

Sistemas Operativos

2021 – 2022

**Modelo de programação Unix
Processos, sinais, pipes anónimos, redireccionamento**

DE/S/ISEC

Sistemas Operativos – 2021/22

João Durães

Tópicos

Funções sistema principais
Criação e gestão de processos
Execução de programas
Sinais
Redireccionamento

Bibliografia específica:

- *Fundamentos de Sistemas Operativos*; 3^a Ed.; Marques & Guedes
Capítulos 6 e 11
- *Beginning Linux Programming*; Mathew & Stones
Capítulos 10,11,12

DE/S/ISEC

Sistemas Operativos – 2021/22

João Durães

Modelo de programação UNIX

Conceito de processo e programa

- Processos e programas são duas entidades bastante diferentes

Programa

- Trata-se de um conjunto de instruções. Descreve como o computador deve proceder para cumprir um determinado algoritmo.
- Essencialmente é apenas uma espécie de receita sem via própria

Processo

- Trata-se de um ambiente de trabalho contendo recursos necessários à execução de um programa. O programa é apenas um desses recursos.
- Cada processo tem sempre exactamente um programa, mas pode mudar o programa por outro
- Aquilo que se diz que está em execução são os processos e não os programas

Modelo de programação UNIX

Atributos de processos e programas

Programa

- Essencialmente um ficheiro executável em disco que é transferido para o interior de um processo para ser executado
- Contém instruções e valores iniciais de trabalho
- Para executar um programa é sempre necessário ter um processo

O mesmo programa pode ser executado por diversos processos em simultâneo.

- Corresponde à situação de executar um programa várias vezes em simultâneo
- Cada execução é independente uma da outra, tendo cada execução dados e evolução próprios

Modelo de programação UNIX

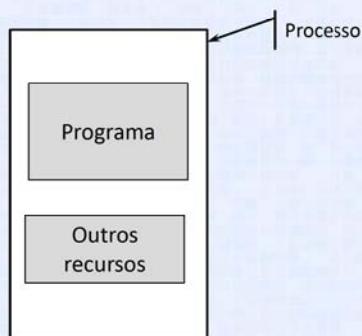
Atributos de processos e programas

Processo

- Conjunto de atributos e recursos
- Espaço de memória organizada em zonas específicas (código, dados, pilha, *heap*)
- Identificação (PID), Prioridade, Directória de trabalho, etc.
- Programa em execução no processo
- Tabela de ficheiros abertos
- Muitos dos atributos do processo podem mudar ao longo da sua vida (inclusive o programa a executar)
- Um processo pode executar vários programas, um de cada vez.
 - A qualquer altura um processo pode mudar o programa por outro. O espaço de memória é adaptado ao novo programa.
- Processos diferentes podem executar o mesmo programa. Mas cada processo é independente dos restantes e as várias “cópias” do programa podem seguir caminhos e ter dados diferentes uns dos outros

Modelo de programação UNIX

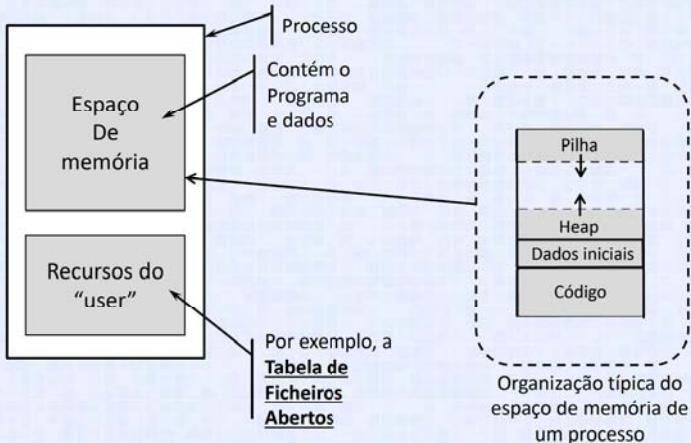
Aspecto conceptual de um processo



- O conceito de processo e a diferença entre processo e programa aplica-se à generalidade de sistemas operativos e não apenas a Unix
- O tema de Processo vs programa será abordado novamente mais adiante

Modelo de programação UNIX

Aspecto concreto de um processo (Unix e não só)



Modelo de programação UNIX

Espaço de memória de um processo

Código (designado como *text* “segment”)

- Contém as instruções do programa
- Normalmente marcada como só leitura para execução (não é suposto as instruções do programa mudarem)

Dados iniciais

- Contém os valores iniciais do programa.
- Exemplos: valores iniciais de variáveis globais, strings fixas (ex., o “olá mundo” em `printf("olá mundo")`), etc.
- Normalmente marcada como apenas de leitura e sem permitir execução. Nem sempre a característica de apenas-leitura é respeitada em todos os sistemas nesta zona de memória (o Unix tipicamente respeita)

Modelo de programação UNIX

Espaço de memória de um processo

Heap

- Contém variáveis e blocos de memória dinâmica (por exemplo, alocadas por malloc)
- O tamanho máximo desta zona não pode ser determinado à partida: depende da execução do programa. O tamanho é dinâmico e pode ser expandido pelo sistema em caso de necessidade
- Normalmente marcada como leitura/escrita e não permite execução porque se destina a ter dados

Pilha

- Contém as variáveis locais e os endereços de retorno das funções
- O tamanho máximo desta zona não pode ser determinado à partida: depende da execução do programa. O tamanho é dinâmico e pode ser expandido pelo sistema em caso de necessidade
- Normalmente marcada como leitura/escrita e não permite execução porque se destina a ter dados (isto impede, por exemplo, ataques *stack smashing*)

Modelo de programação UNIX

Espaço de memória de um processo

- As zonas de memória do processo adaptam-se ao programa que foi carregado para o processo.
- Normalmente cada zona é “marcada” com tipos de acesso específico (só leitura, execução: S/N, etc.) para aumentar a estabilidade do programa e sistema e impedir certos tipos de ataques e vírus.
- As zonas *heap* e pilha são dinâmicas e expandem-se consoante o necessário.
 - Expandem-se na direcção uma da outra: *heap* expande para cima (para endereços superiores) e a pilha para endereços inferiores
 - Desta forma encontram-se (colidam) o mais tarde possível. Tendo em atenção que não se sabe à partida qual das duas irá necessitar de mais memória, evita-se assim impor limites prévios arbitrários a cada uma – esta “colisão” apenas acontece quando efectivamente não houver mais memória disponível

O espaço de memória de processos será abordado novamente, por exemplo (mas não só) no capítulo e gestão de memória

Modelo de programação UNIX

Implementação em Linux (semelhante à dos outros sistemas)

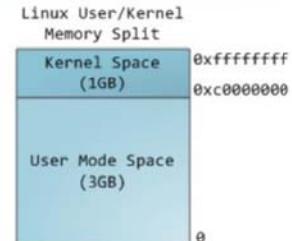
- O sistema atribui uma zona para uso exclusivo do processo.
- Cada processo tem uma zona de memória com aspecto semelhante e com gamas de endereços iguais
- Não há *colisão de memória* entre os vários processos porque os endereços vistos por cada um são mapeados para outros endereços pelo processador em run time de forma a não colidirem
 - Trata-se do **mecanismo de endereçamento** virtual a ver mais adiante
 - Endereços virtuais: Endereços vistos pelas instruções dentro do processo
 - Endereços reais: endereço que realmente estão a ser usados na memória. OS processos não se apercebem desta tradução. É feita por hardware

Modelo de programação UNIX

Implementação em Linux

Exemplo em arquitectura de 32 bits

- O conteúdo relativo ao processo fica nos endereços mais baixos.
 - Cada processo tem o seu próprio conteúdo (diferente)
- O sistema fica mapeado no último Gb
- Este último Gb é comum a todos os processos através de um esquema de memória partilhada simples, baseado no mecanismo de endereçamento virtual.
 - O sistema é o mesmo para todos os processos e por isso só existe uma cópia



Modelo de programação UNIX

Implementação em Linux (semelhante à dos outros sistemas)

Estrutura e nomenclatura das zonas de memória do processo

- Pilha -> **Stack**
Variáveis locais não estáticas. Endereços de retorno
- Heap -> **Heap**
Memória alocada dinamicamente
- Variáveis não inicializadas (podem ser modificadas) -> **BSS segment**
Exemplo: variável global `int dados[10]; /* onde está o conteúdo de dados */`
- Variáveis inicializadas no código -> **Data segment**
Exemplo: `char * p = "olá mundo"; /* onde está o "olá mundo"?`
- Código (instruções) -> **Text Segment**

Modelo de programação UNIX

Implementação em Linux (semelhante à dos outros sistemas)

Estrutura e nomenclatura das zonas de memória do processo

- **Stack, Heap, BSS segment, Data segment, Text Segment**

Estes nomes podem ser observados usando a ferramenta ***objdump*** (inspecciona ficheiros objectos e ficheiros executáveis)

- Existem em Linux e em Windows

Esta organização é directamente transcrita do ficheiro executável

- O formato do ficheiro executável é, assim, muito importante e fortemente dependente do Sistema
- Em Linux: formato “**ELF**” (Executable and Linking Format)
- Windows: formato “**PE**” (Portable Executable) – parecido, mas diferente

Modelo de programação UNIX

Criação de processos e execução de programas em Unix

Em Unix, a criação de processos e a execução de programas são duas ações distintas e independentes uma da outra

Criação de um processo.

- É feito com base na duplicação de um processo que já existente
 - > Mecanismo **fork**
- O novo processo fica com uma cópia do conteúdo do original
- O novo processo fica a executar uma cópia do programa que estava a executar no original, com o mesmo contexto e valores de variáveis (copiadas)
- No novo processo a execução do programa prossegue no ponto em que já ia no contexto do processo original.

- O processo original (*pai*) e o seu clone (*filho*) são praticamente indistinguíveis. Terão identificação distintas e podem usar essa informação para decidirem fazer coisas diferentes um do outro.

Modelo de programação UNIX

Criação de processos e execução de programas em Unix

Execução de um novo programa

- Trata-se de uma operação desencadeada a partir de um processo já existente
 - > Mecanismo **exec**
- O processo passa simplesmente a executar o novo programa, perdendo-se o anterior.
- O novo programa começa a sua execução no inicio (exemplo, função *main*)

- Se se quiser executar um programa novo sem perder o que já estava em execução, terá que se criar um novo processo e nele executar o programa pretendido (ou seja, *fork + exec*)

Modelo de programação UNIX

Criação de processos

```
pid_t fork(void)
```

Cria um processo novo

Devolve:

- 1 : houve erro
- 0 : contexto do processo filho
- outro valor: contexto do processo pai

A criação do novo processo é feita através de uma forma optimizada baseada em partilha de memória, do mecanismo *copy-on-write* e apoiada nos mecanismos de endereçamento virtual do processador

Inicialmente os processos têm conteúdo igual e partilham a mesma zona de memória RAM. À medida que vão divergindo e tendo valores diferentes, o sistema vai duplicando a memória de forma a que cada processo tenha zonas de RAM diferentes e independentes. Isto é completamente transparente para os processos e será visto novamente no capítulo de gestão de memória

Modelo de programação UNIX

Identificação de processos

```
pid_t getpid(void)
```

Obtém identificação do processo

```
pid_t getppid(void)
```

Obtém identificação do processo pai (processo que criou o processo que invoca a função)

Modelo de programação UNIX

Execução de programas

```
int exec1(char * path, char * arg, ...)  
int execlp(char * file, char * arg, char * arg, ...)  
int execle(char * path, char * arg, ..., char * envp[])  
int execv(char * path, char * argv[])  
int execvp(char * file, char * argv[])  
int execve(char * file, char * argv[], char * envp[])
```

- Executa o programa indicado no contexto do processo que invoca a função
- Em caso de sucesso não devolve nada.
- O programa em execução é substituído pelo novo. O processo e os recursos mantém-se (exemplo, ficheiros abertos)

Modelo de programação UNIX

Sincronização simples de processos

```
pid_t wait(int * status)
```

- Espera que um processo filho termine
- Armazena o *exit code* em *status*

```
pid_t waitpid(pid_t pid, int * status, int options)
```

Espera que um processo filho específico termine

pid pode ser (exemplos):

- -1 ► espera pelo primeiro processo filho que terminar
- > 0 ► espera pelo processo cujo PID é o indicado (em *pid*)

options pode ser :

- | | |
|-------------|--|
| • WNOHANG | ► retorna imediatamente se nenhum processo filho tiver terminado |
| • WUNTRACED | ► retorna também para processos filho que tenham sido parados |

Modelo de programação UNIX

Macros para obter informação a partir do *exit status*:

WIFEXITED(*status*)

- *true* se o processo filho terminou normalmente

WEXISTATUS(*status*)

- Obtém o valor indicado no *return* ou *exit* do processo filho (8 bits)

WIFSIGNLLED(*status*)

- *true* se o processo filho terminou como resposta a um sinal

WTERMSIG

- Obtém o sinal que causou o término do processo filho

ETC – consultar páginas man

Modelo de programação UNIX - Sinais

Sinais em Unix

Existem actualmente duas formas de funcionamento de sinais

- 1) signal/kill
- 2) sigaction/sigqueue

A primeira forma é mais simples mas menos poderosa

Os sinais correspondem a um mecanismo de notificação.

- Permitem a um processo assinalar a outro que algo ocorreu
- Não são mecanismos de comunicação: não transportam informação nenhuma para além do facto que algo ocorreu (forma 2 permite passar um inteiro)
- Existem diversos sinais. Cada sinal tem um código diferente, permitindo que se possa usar sinais diferentes para objectivos diferentes

Próximos slides: forma 1) = signal/kill

Modelo de programação UNIX - Sinais

O mecanismo de sinais é completamente assíncrono

- O processo **A** que envia um sinal ao processo **B** fá-lo quando bem entender, não sendo necessário que o processo alvo **B** esteja previamente à espera, ou que seja acordado entre ambos uma altura para fazer o envio.
- O processo **B** que vai receber um sinal não tem que estar explicitamente à espera de um sinal. Estará simplesmente a executar o seu algoritmo e o sinal é entregue a qualquer altura (*)

Usando uma analogia com a vida real, o mecanismo de sinais é semelhante às campainhas dos prédios com duas alterações:

- Existem várias campainhas para cada apartamento, cada uma com um som diferentes (há vários sinais), e
- Não há intercomunicador – apenas campainhas (os sinais não são mecanismos de comunicação)

(*) Os sinais são detectados e entregues apenas quando a execução está a passar de código do núcleo para código do processo alvo. Cada sinal corresponde a um bit num vector: foi recebido/não foi recebido. É colocado a 1 quando é enviado e a 0 quando é recebido (não se acumulam enquanto não são tratados)

Modelo de programação UNIX - Sinais

Sincronização simples de processos com sinais

```
typedef void (*sghandler_t)(int)      ► função para tratar um sinal
```

```
sighandler_t signal(int signum, sghandler_t handler)
```

Associa uma função *handler* ao sinal *signum*

(Significa que o sinal vai ser *tratado* pelo programa)

Valores especiais para *handler*:

SIG_IGN – O sinal é ignorado

SIG_DFL – O sinal causa o tratamento por omissão

Devolve a situação anterior (*handler*) relativa a esse sinal

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

Envia o sinal *sig* a um processo *pid*

- Os sinais apenas são atendidos na transição de código do núcleo para código do processo. O método bascia-se na manipulação da pilha utilizador de forma a criar um registo fictício de chamada a função (a função de tratamento do sinal)

Modelo de programação UNIX - Sinais

Exemplo de tratamento de sinais (programa que apanha o ^C)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
int a = 0;
void trata_sinal(int s) {
    a++; printf("ouch %d ",a); /* fflush(stdout); */
    if (a==5) { printf("até logo "); exit(0); }
}

int main() {
    setbuf(stdout, NULL);
    signal(SIGINT,trata_sinal);
    while(1) pause();
    return 0;
}
```

Modelo de programação UNIX - Sinais

Alguns dos sinais mais relevantes

Sinal	Valor	Acção por omissão
SIGHUP	(1)	Termina processo
SIGINT	(2)	termina processo
SIGQUIT	(3)	Termina processo + <i>dump core</i>
SIGKILL	(9)	Termina processo ► Não pode ser tratado
SIGALRM	(14)	termina processo
SIGUSR1	(30)	Termina processo ► Para uso do programador
SIGUSR2	(32)	Termina processo ► Para uso do programador
SIGCHLD	(20)	Ignorado

Há mais sinais – consultar páginas man

Nota: os números dos sinais podem variar de sistema para sistema

Modelo de programação UNIX - Sinais

Os sinais **desbloqueiam** a execução do processo que os recebem

Processo bloqueado:

- Significa que a execução do processo está parada num determinado ponto à espera de algo.
- Por exemplo: está à **espera de caracteres numa leitura desencadeada por um scanf** (há muitos mais exemplos)
- **Em SO, “blockeado” não significa “encalhado” nem “crashado”.**
Trata-se se uma situação temporária normal e vulgar.

A recepção de um sinal pode interromper, por exemplo

- Uma leitura de caracteres desencadeada por scanf
- Um operação pause() (aver mais adiante)
- Uma operação sleep() (a ver mais adelante)
- Etc.

Modelo de programação UNIX - Sinais

Interrupção de uma operação por um sinal

Como agir?

A recepção de um sinal pode interromper, por exemplo, uma leitura de *scanf*.

- O que fazer? Repetir o *scanf*? O *scanf* recomeça automaticamente?
- O comportamento do *scanf* (e outros mecanismos) nesta situação pode variar de sistema para sistema. **É necessário verificar o comportamento** (o *scanf* retorna um inteiro que é o número de campos lidos. Se retornar EINTR significa que foi interrompido)
- No Linux habitual, o *scanf* continua a leitura no ponto onde tinha parado. **O próximo exemplo ilustra este comportamento**

Modelo de programação UNIX - Sinais

Exemplo: sinais a meio de interacção com utilizador (1/3)

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
int RecebeuSinal; // flag para uso do resto do programa

// função de atendimento de sinal
void atendeSinal(int snum) {
    // executa a acção assíncrona associada ao sinal
    // por exemplo, lê uma mensagem de um named pipe
    printf("\nSinal recebido.\n");
    printf("\nProcessar acontecimento, ler named pipes, etc\n");
    // assinala resto do programa que foi recebido o sinal,
    // para o caso de ser relevante, usando a flag
    RecebeuSinal = 1;
}
```

Modelo de programação UNIX - Sinais

Exemplo: sinais a meio de interacção com utilizador (2/3)

```
int main(int argc, char * argv[]) {
    char buffer[50];
    int res;
    printf("\n\nPID=%d\n",getpid()); // apresenta PID
    RecebeuSinal = 0; // Nenhum sinal recebido
    signal(SIGUSR1, atendeSinal); // associa função a SIGUSR1

    while (1) {
        printf("Favor escrever algo (\\"fim\\" para sair)\n--> ");
        res = scanf("%s", buffer);
        printf("scanf leu isto: %s\n", buffer);
        printf("resultado do scanf = %d\n", res);
    }
}
```

Modelo de programação UNIX - Sinais

Exemplo: sinais a meio de interacção com utilizador (3/3)

```

if (RecebeuSinal == 1) {
    RecebeuSinal = 0; // reset à flag
    printf("Parece que entretanto foi atendido ");
    printf("um sinal qualquer\n");
    printf("Lamentamos a eventual confusão na ");
    printf("interface com o utilizador\n");
}
if (strcmp(buffer,"fim")==0)
    break;
}
printf("ok\n\n");
}
  
```

Modelo de programação UNIX - Sinais

Utilização do exemplo anterior

```

PID=2954
favor escrever algo ("fim" para sair)
--> ola
scanf leu isto: ola
resultado do scanf = 1
favor escrever algo ("fim" para sair)
--> asd
Sinal recebido. Processar acontecimento, ler named pipes, etc
fgh
scanf leu isto: asdfgh
resultado do scanf = 1
Parece que entretanto foi atendido um sinal
Lamentamos a eventual confusão na user interface
favor escrever algo ("fim" para sair)
  
```

Foi escrito **asdfgh** seguido, mas ocorreu um sinal a meio.

Após o sinal, o **scanf** continuou e não perdeu os caracteres já lidos.

Envio do sinal (noutra consola foi escrito): **kill -s SIGUSR1 2954**

Modelo de programação UNIX - Sinais

Acerca do exemplo anterior

A capacidade dos sinais de interromper algo (tarefa 1), desencadeando a execução de uma função associada ao sinal (tarefa 2) pode ser muito útil em cenários em que é necessário a um processo (aparentemente) executar duas tarefas em simultâneo.

Situação exemplo: um processo precisa de fazer em simultâneo:

- Tarefa 1: interagir com o utilizador, (exemplo, `scanf`s e semelhantes)
- Tarefa 2: dar atenção a mensagens provenientes de um outro processo "servidor" (exemplo, ler dados de mecanismos de comunicação inter-processo *named pipes*)

Problema: se está a fazer uma coisa, não está a fazer a outra

Como resolver isto?

-> Resolução com **sinais**: Próximos slides

Obs:

Resolução com **select** -> Mais adiante na matéria

Resolução com **threads** -> Mais adiante na matéria

Modelo de programação UNIX - Sinais

Soluções

(para quando se quer fazer mais do que uma coisa ao mesmo tempo)

Genérica e melhor:

- Usar **threads** (**threads** são discutidas mais adiante e depois aprofundadas em SO2)

Específica ao caso de I/O:

- Usar o mecanismo **select** (a ver mais adiante)

"Desenrasque":

- Usar **sinais**: próximo slide
- "*desenrasque*" porque há formas melhores de resolver essa questão (as duas anteriores) e porque só funciona em sistemas onde houver sinais (Unix), e porque os sinais não são (nem deixam de ser) especificamente para esse objectivo

Modelo de programação UNIX - Sinais

Solução do cenário do slide anterior usando sinais

- Trata-se de um cenário muito semelhante ao exemplo do scanf/sinal apresentado atrás
- O processo **A** está a interagir com o utilizador. Quando o tal outro processo “servidor” **B** tem algo a dizer a este processo **A**, faz:
 - 1º escreve a mensagem no tal mecanismo *named pipe*, e
 - 2º envia um sinal ao processo **A**.O processo **A**, ao receber o sinal, executa uma função (previamente associada ao sinal) que lê a mensagem no *named pipe*, e volta ao que estava a fazer (interagir com o utilizador).

Este cenário/exemplo pode ter aplicação nos trabalhos práticos de SO. No entanto há formas melhores de fazer isto (exemplo: mecanismo **select**)

Modelo de programação UNIX – Sinais 2: sigaction/sigqueue

Segunda forma de lidar com sinais: **sigaction** / **sigqueue**

Podem-se usar duas outras funções para lidar com sinais:

- **sigaction**: define o comportamento dos sinais (em vez de **signal**)
- **sigqueue**: envia sinais (em vez de **kill**)

Esta forma de uso permite funcionalidade adicional:

- Configurar detalhes adicionais de comportamento dos sinais
- Enviar um valor juntamente com o sinal
- Saber o PID do processo que enviou o sinal

Forma mais recente e mais portável que signal/kill

Modelo de programação UNIX – Sinais → 2)

Segunda forma de lidar com sinais: sigaction/sigqueue

Função para ver/modificar comportamento associado a um sinal

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

signum

Sinal a tratar

struct sigaction * act

Ponteiro para estrutura que descreve o que fazer com o sinal (como passa a ser tratado o sinal)

struct sigaction * oldact

Ponteiro para estrutura que é preenchida com a descrição de como era feito o tratamento do sinal

Modelo de programação UNIX – Sinais -> (2)

Estrutura sigaction

```
struct sigaction {
    void      (*sa_handler)(int);
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t  sa_mask;
    int       sa_flags;
    void      (*sa_restorer)(void);
};
```

Parâmetro	Significado
handler	Ponteiro para a função que atende o sinal, na forma void func(int)
sigaction	Ponteiro p/ função que atende o sinal, forma: void func (int, siginfo_t *, void *)
-->	Apenas um destes ponteiros deve estar definido (é uma union)
sa_mask	Bit mask com os sinais que ficam bloqueados durante o atendimento
sa_flags	Exemplo: SA_NODEFER (não bloquear o sinal durante o atendimento)
sa_restorer	Uso reservado. Não é para uso da aplicação

Modelo de programação UNIX – Sinais → (2)

Segunda forma de lidar com sinais: sigaction/sigqueue

Função para enviar um sinal

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

pid

PID do processo alvo

sig

Sinal a enviar

value

Valor a passar ao processo alvo juntamente com o sinal

Modelo de programação UNIX - Sinais

Segunda forma de lidar com sinais: sigaction

```
union sigval {  
    int sival_int;  
    void *sival_ptr;  
};
```

Trata-se de uma **union**: os campos estão sobrepostos

-> Ou se usa um campo, ou se usa o outro

O programador envia ou um inteiro, ou um ponteiro

- O significado do valor é o programador que o decide.
 - > **Nota** (no caso de se enviar um ponteiro): **um ponteiro só tem significado no contexto do processo onde esse ponteiro foi obtido**
 - > O uso do ponteiro tem a ver com ser **um tipo de dados genérico** (como o void) e não é propriamente para “apontar”

Modelo de programação UNIX – Sinais → 2)

Segunda forma de lidar com sinais: sigaction/sigqueue

Função para atender um sinal

```
void funcao(int sig, siginfo_t * siginfo, void * ucontext)
```

pid

Sinal recebido

siginfo

Ponteiro para estrutura siginfo_t com informação acerca do sinal recebido

ucontext

Ponteiro para estrutura ucontext_t com informações de baixo nível acerca do contexto do processo (estado interno do processo, incluindo valores dos registos do processador salvaguardados, máscara dos sinais, etc.)
→ ucontext.h para ver os campos (dependem da implementação do sistema)

Modelo de programação UNIX - Sinais

Estrutura siginfo_t (os campos mais úteis estão assinalados)

```
typedef struct {
    int si_signo;    -> número do sinal
    int si_code;     -> Informação extra acerca do sinal
    union sigval si_value; -> valor enviado com o sinal
    int si_errno;   -> errno (normalmente não usado em Linux)
    pid_t si_pid;   -> PID de quem enviou o sinal
    uid_t si_uid;   -> UID (User ID) do processo que enviou o sinal
    void *si_addr;  -> (SIGBUS/SGSEV/SIGILL/SIGFPE) endereço onde foi
                      originado o problema que causou o sinal
    int si_status;  -> (SIGCHLD) Código do sinal enviado ao filho
    int si_band;    -> Código de evento de controlo de I/O
} siginfo_t;
```

Esta estrutura tem mais campos consoante as diversas implementações do sistema

Modelo de programação UNIX - Sinais

Informação em [si_code \(excerto\)](#)

Geral

SI_ASYNCIO	Completion of an asynchronous I/O (AIO) operation
SI_KERNEL	Sent by the kernel (e.g., a signal from terminal driver)
SI_MESGQ	Message arrival on POSIX message queue (since Linux 2.6.6)
SI_QUEUE	A realtime signal from a user process via sigqueue()
SI_SIGIO	SIGIO signal (Linux 2.2 only)
SI_TIMER	Expiration of a POSIX (realtime) timer
SI_TKILL	A user process via tkill() or tgkill() (since Linux 2.4.19)
SI_USER	A user process via kill() or raise()

Para SIGCHLD

CLD_CONTINUED	Child continued by SIGCONT (since Linux 2.6.9)
CLD_DUMPED	Child terminated abnormally, with core dump
CLD_EXITED	Child exited
CLD_KILLED	Child terminated abnormally, without core dump
CLD_STOPPED	Child stopped
CLD_TRAPPED	Traced child has stopped

SIGFPE

FPE_FLTDIV	Floating-point divide-by-zero
FPE_FLTINV	Invalid floating-point operation

[ETC -> ver man pages](#)

Modelo de programação UNIX - Sinais

signal/kill vs. sigaction/sigqueue

- **sigaction/sigqueue é mais recente e mais versátil**
 - Em princípio será mais vantajoso usar o API mais recente sigqueue/sigaction

- **O comportamento não é exactamente igual entre ambos**

- Isto deve-se ao facto das *flags default* em sigaction() não corresponderem ao comportamento *default* em signal()

Exemplo:

- O exemplo do comportamento de *scanf* apresentado atrás tem um resultado diferente se for usado sigaction() em vez de signal(). Com sigaction() o scanf é mesmo interrompido e os caracteres anteriormente lidos perdem-se
- No entanto, este comportamento pode ser configurado de forma a ficar igual ao que se tem com signal. Bastara especificar a flag SA_RESTART no campo sa_flags
- Em conclusão: sigaction é mais versátil e permite especificar melhor o comportamento desejado

Modelo de programação UNIX - Sinais

Funções relacionadas com o mecanismo de sinais

`int pause(void)`

- Aguarda que o processo receba um sinal

Esta função pode ser usada para sincronização simples entre dois processos. Um processo **A** que dependa de uma determinada acção **x** por parte de outro processo **B** poderá invocar **pause()**, ficando adormecido. O processo **B**, após concluir a acção **x**, enviará um sinal ao processo **A**, acordando-o.

Importante: a função **pause()** faz com que o processo que a invocou fique adormecido (o termo correcto é “bloqueado”). O processo não fará mais nada até receber um sinal, altura em que prosseguirá normalmente

Modelo de programação UNIX - Sinais

Funções relacionadas com o mecanismo de sinais

`unsigned int alarm(unsigned int seconds)`

- Faz com que seja enviado um sinal SIGALRM ao processo que invoca a função dentro de **seconds** segundos.
- Retorna o número de segundos que ainda faltavam referentes a um eventual pedido de alarme anterior (0 se nenhum)
- Após a invocação, o processo prossegue a sua execução normalmente

Esta função pode ser usada para construir um mecanismo de despertador. Um processo **A** pretende ser notificado quando se tiver passado **x** segundos, mas precisa de fazer várias acções e não quer ficar adormecido (“bloqueado”) durante esses **x** segundos.

- Ao invocar a função **alarm()**, o sistema vai enviar-lhe um sinal SIGALRM passados esses **x** segundos, durante os quais o processo prossegue a sua actividade normal.
- Em princípio, o processo terá associado uma função à recepção do sinal para efectuar algo no final dos **x** segundos

Modelo de programação UNIX - Sinais

Funções relacionadas com o mecanismo de sinais

`unsigned int sleep(unsigned int seconds)`

- Faz com que o processo adormeça (não executa) até que um sinal chegue ou que o número de segundos especificado passe
- Retorna o número de segundos que ainda faltavam no caso do `sleep` ter sido interrompido (por exemplo, pela recepção de um sinal)

Esta função pode ser usada para concretizar pausas de duração conhecida. Tal como a generalidade dos restantes mecanismos “bloqueantes”, a pausa pode ser interrompida pela recepção de um sinal

Modelo de programação UNIX – Sinais - Implementação

A informação acerca do estado e configuração dos sinais faz parte de cada processo

Isto significa que:

fork → O novo processo “herda” a configuração de sinais feita pelo processo pai (o processo filho obtém uma cópia da informação relativa aos sinais existente nos processo pai)

exec → O processo adquire o comportamento *default* para cada sinal uma vez que as funções de atendimento eventualmente programadas deixaram de existir (o código anterior foi substituído por outro)

Modelo de programação UNIX – Sinais - Implementação

Implementação do atendimento de sinais

- Cada processo tem uma tabela de *flags de sinais*. Para cada sinal (SIGINT, SIGALRM, SIGUSR1, etc.) existe a informação “sinal chegou / sinal não chegou”. É um bit: sim/não chegou. Não indica quantas vezes chegou “entretanto” um sinal.
- A flag é colocada a 1 quando o sinal é enviado ao processo, e colocada a 0 quando o sinal é atendido.
- Os sinais não acumulam: a informação respeitante a cada sinal é apenas chegou/não chegou (1 flag por cada sinal). Se um processo enviar muitos sinais iguais repetidos enquanto o processo alvo não está a executar, este apenas terá a percepção de um sinal X

Modelo de programação UNIX – Sinais - Implementação

Implementação do atendimento de sinais

- Os sinais são verificados (e atendidos, se for essa a configuração) quando a execução está a **regressar do sistema para dentro do código do processo**

Exemplos

- Retorno de uma função sistema
- Reposição do processo após preempção pelo sistema

Regresso ao código do processo:

- **Situação normal** (sem sinais a atender): o regresso ao código do processo limita-se a saltar para o endereço onde este estava quando perdeu a execução
- **Situação com um sinal associado a uma função**: O sistema “falsifica” uma chamada à função manipulando a pilha do processo. Detalhes -> próximo slide

Modelo de programação UNIX – Sinais - Implementação

Implementação do atendimento de sinais

Como é feita a invocação da função de atendimento do sinal no contexto do processo alvo do sinal, não havendo nenhuma chamada explícita feita pelo programador :

Ao regressar do sistema para dentro do processo o sistema faz:

- Coloca na pilha do processo um registo “artificial” referente ao endereço onde o processo estava actualmente.
- O sistema salta para o endereço da função de atendimento e não para o endereço onde o processo estava.
- Quando função de atendimento termina, a sua instrução final “RET” naturalmente retira o endereço da pilha que corresponde ao ponto onde o processo ia e a execução passa para esse ponto *tal e qual como se a função tivesse sido chamada explicitamente a partir desse ponto*

Modelo de programação UNIX – Sinais

Notas

O comportamento de sinais pode variar de versão de Unix para versão de Unix

(Nota: os vários Linuxes são todos a mesma versão de Unix)

O uso de *signal/kill* em vez de *sigaction/sigqueue* pode fazer variar alguns pormenores do seu funcionamento

-> É importante testar estes aspectos em pequenos programas de *prova-de-conceito*

Desafio

(foi falado na aula, portanto é mais **revisão** que outra coisa)

-> *O que acontece ao atendimento de sinais quando*

- é feito *fork()*
- é feito *execxx()*

Modelo de programação UNIX – Redireccionamento

Redireccionamento de input/output

- Entre um programa e um ficheiro
- Entre dois programas

O assunto de redireccionamento envolve os seguintes assuntos

- Funcionamento de ficheiros em Unix
- Funcionamento das funções de entrada e saída em Unix
- Tratamento de input/output standard
- Pipes anónimos

Este tópico está fortemente relacionado com os seguintes assuntos

- Processos em Unix e tabela de ficheiros abertos
- Percurso das operações de I/O desde as funções biblioteca até ao sistema
- Mecanismo *fork*
- Mecanismo *exec*

OS slides seguintes abordam este assunto e no final apresentam três exemplos

Modelo de programação UNIX – Redireccionamento

Funcionamento de ficheiros em Unix

Tabela de ficheiros abertos

- Cada processo em Unix tem uma **tabela de ficheiros abertos**
- Cada posição nessa tabela contém um **descriptor**. Este descriptor tem a informação de controlo acerca do uso do ficheiro (modo de abertura, posição actual no ficheiro etc.)
- Tudo o que o programa precisa de fazer para operar sobre um ficheiro é referir a posição nesta tabela que controla o ficheiro em questão. O sistema irá buscar a essa posição os dados que precisa.
- A função de abertura de ficheiro é responsável por colocar na tabela (primeira posição livre) a informação de controlo acerca do ficheiro
- O programador geralmente não interage directamente com a tabela de ficheiros abertos.

Modelo de programação UNIX – Redireccionamento

Funcionamento de ficheiros em Unix

Funções de manipulação de ficheiros

-> **Funções biblioteca standard C**

- *fopen, fclose, fread, fwrite, etc (com "f")*
- Usam FILE * para identificar o ficheiro em causa
- Estas funções **usam internamente as funções sistema.**

-> **Funções sistema Unix para manipulação de ficheiros**

- *open, close, read, write, etc (sem "f")*
- São semelhantes às funções biblioteca C.
- Todo o acesso a ficheiros **passa obrigatoriamente** por estas funções
- Uma diferença muito visível é o facto de usarem um **inteiro** para identificar um ficheiro em vez de um FILE *
- **Esse inteiro é simplesmente o índice (posição) dentro da tabela de ficheiros abertos do processo.**

Modelo de programação UNIX – Redireccionamento

Funcionamento de ficheiros em Unix

Funções de manipulação de ficheiros

-> **Funções biblioteca standard C**

- O uso de *fwrite* (exemplo) implica internamente o uso da função sistema *write*
- *printf* e *scanf* são operações de entrada/saída: passam necessariamente pelo uso das funções de manipulação de ficheiros
- **Exemplo:** *printf --> fprintf --> fwrite --> write*
 - Neste caso, a função *printf* desencadeia o uso de um ficheiro: **stdout** trata-se do pseudo-ficheiro que corresponde ao dispositivo “standard output” (consola). Este ficheiro está sempre na posição **1** da tabela de ficheiros abertos, identificado por **STDOUT_FILENO**
 - De igual forma, *scanf* (usando **stdin**) implica uma operação **read** sobre o pseudo-ficheiro na posição **0** ou **STDIN_FILENO** que deve estar sempre na posição **0** da tabela de ficheiros abertos

Modelo de programação UNIX – Redireccionamento

Redireccionamento – Ideia base

- printf traduz-se em operações write sobre o ficheiro descrito pela **posição 1** da tabela de ficheiros abertos do processo
 - scanf traduz-se em operações read sobre o ficheiro descrito pela **posição 0**
- O que acontece se se modificar o conteúdo das posições 0 e 1 da tabela de ficheiros abertos de forma a “apontar” para outros ficheiros?
- As operações read e write (relativas aos scanf e printf) serão desviadas (“redireccionadas”) para os ficheiros em vez de teclado/ecrã
 - Esta ideia corresponde ao redireccionamento tal como em (exemplos)
 - ls > etc.txt
 - sort < abc.txt
 - Também se pode redireccionar a saída de erro standard (stderr), cujo descritor corresponde à posição 2 (STDERR_FILENO) da tabela de ficheiros abertos

Modelo de programação UNIX – Redireccionamento

Redireccionamento para ficheiros – Concretização

->> Substituir os descritores na tabela de ficheiros abertos correspondentes às posições *stdin* (posição 0 = STDIN_FILENO) e *stdout* (posição 1 = STDOUT_FILENO) por descritores para outro destino

- Os novos descritores podem referir ficheiros regulares ou mecanismo que seja compatível com a ideia de ficheiro (exemplo, pipes)
- Tudo o que é necessário é que os descritores estejam nas posições standard (0 para *stdin*, 1 para *stdout*, 2 para *stderr*)
- Exceptuando eventuais pormenores de sincronização, as operações iniciadas pelos printf/scanf continuam válidas apesar de terem sido redireccionadas para outros destino

Modelo de programação UNIX – Redireccionamento

Redireccionamento para ficheiros – COncretização

Redireccionamento feito pela *Shell* na linha de comandos:

->>A substituição dos descritores é feita pela *Shell* antes de lançar a execução do comando alvo do redireccionamento, e já dentro do contexto do processo criado para a sua execução (para não afectar as operações d I/O da própria *Shell*)

- Essa substituição ocorre **depois do fork e antes do exec**, na sequência habitual de acções da shell no lançamento de um comando
 - Depois do fork, já no processo filho -> de forma a não afectar a tabela de ficheiros abertos da própria shell (o redireccionamento é apenas para o comando em questão)
 - Depois do fork, já no processo filho mas antes do exec, no processo filho -> a invocação de exec substitui o código pelo do programa a executar mas o processo é o mesmo e mantém-se a tabela de ficheiros: o novo código irá usar a tabela com a alteração feita e o redireccionamento é transparente para o programa executado

Modelo de programação UNIX – Redireccionamento

Redireccionamento para ficheiros

Exemplo para redireccionamento de stdin (exemplo: **sort < abc.txt**)

- No contexto da shell, processo filho

```
close(STDIN_FILENO); // liberta posição 0 (stdin)
open("abc.txt", O_RDONLY); // abre o ficheiro (fica na pos. 0)
execlp("sort", "sort", NULL); // executa o programa ls
```

Exemplo para redireccionamento de stdout (exemplo: **ls > fich.txt**)

- No contexto da shell, processo filho

```
close(1); // liberta posição 0 (stdout)
open("fich.txt", O_WRONLY); // abre o ficheiro (fica na pos. 1)
execlp("ls", "ls", NULL); // executa o programa sort
```

Nota: podem-se usar ambos os redireccionamentos em simultâneo

É com que acontece com, po exemplo: **sort < abc.txt > ord.txt**

Modelo de programação UNIX – Redireccionamento

Redireccionamento entre programas com *pipes* anónimos

O redireccionamento entre dois programas (exemplo: ls | sort) é mais complexo do que o redireccionamento para ficheiros. No entanto, o mecanismo básico é o mesmo

- Usa-se um mecanismo de comunicação entre processos: **pipe anónimo**
- Um pipe anónimo comporta-se como dois ficheiros e produz **dois descritores** de ficheiros abertos: num escreve-se (extremidade de escrita) e no outro lê-se aquilo que foi escrito (extremidade de leitura)
- Num processo redirecciona-se o output (stdout) para a extremidade de escrita do pipe
- No outro processo redirecciona-se o input (stdin) para a extremidade de leitura do mesmo pipe
- Assim, aquilo que um processo escreve (stdout) é conduzido para a entrada (stdin) do outro processo.

Modelo de programação UNIX – Redireccionamento

Redireccionamento entre programas com *pipes* anónimos

Lógica geral (assumir exemplo: ls / sort)

- A shell irá criar dois processos: um para um processo (exemplo: ls), outro para o segundo processo (exemplo: sort)
- Antes de criar ambos é necessário criar um pipe.
- Após a criação de cada filho, e já no contexto de cada filho é feita a substituição dos descritores na tabela de ficheiros abertos

O slide seguinte acrescenta os pormenores

Modelo de programação UNIX – Redireccionamento

Redireccionamento entre programas com *pipes* anónimos

- *Antes de criar ambos é necessário criar um pipe.*
 - Este passo cria na tabela de ficheiros abertos dois descritores: uma para a extremidade de leitura e outro para a extremidade de escrita
 - Este passo é feito antes da criação dos filhos para que ambos herdem a tabela de ficheiros já com os descritores para o pipe e assim tenham visibilidade sobre o pipe. *Esta é a única forma de partilhar o pipe anónimo em ambos os processos dado que este mecanismo não tem um identificado associado*
- *Após a criação de cada filho, e no contexto de cada filho:*
 - No processo que escreve (exemplo: ls), redirecciona-se o stdout para a extremidade de escrita do pipe. Isto corresponde a mover (com a função **dup**) o descrito do pipe para a posição do stdout na tabela de ficheiros abertos. Isto é feito antes do exec
 - No processo que escreve redirecciona-se o stdin para a posição do stdin de forma análoga

Modelo de programação UNIX – Redireccionamento

Redireccionamento entre programas com *pipes* anónimos

Código exemplo (assumir exemplo: *ls / sort*)

Antes de criar ambos os filhos

```
int mat[2]; // matriz para os descritores das extremidades do pipe
pipe(mat); // cria o pipe.
            // mat[0] --> descritor do lado de leitura
            // mat[1] --> descritor do lado de escrita
// ... fork

// a matriz é preenchida com os índices na tabela de ficheiros abertos
// onde foram colocados os descritores das extremidades do pipe
// filhos herdam a tabela de descritores => herdam o acesso ao pipe
// assim podem comunicar um com o outro através desse pipe
```

Modelo de programação UNIX – Redireccionamento

Redireccionamento entre programas com *pipes* anónimos

Código exemplo (assumir exemplo: *ls* / *sort*)

No contexto do processo filho “que escreve” (exemplo = para executar o *ls*)

```
close(1); // liberta posição de stdout (pos. 1 = STDOUT_FILENO)
dup(mat[1]); // duplica extremidade de escrita do pipe para pos. 1
close(mat[1]); // fecha extremidade de escrita porque já foi duplicada
close(mat[0]); // fecha extrem. de leitura do pipe porque não a vai usar
//execlp("ls","ls",NULL);
```

- Deve ser usado e STDIN_FILENO em vez de 0 e STDOUT_FILENO em vez de 1
 - (este slide e seguinte não o fazem por razões de espaço)
- **dup()** é necessário para duplicar o descritos colcando a cópia na posição recém libertada pelo **close()** anterior, ficando na posição standard esperada
 - **Evitar** o recurso a **dup2()** pois oculta parte do funcionamento desta matéria

Modelo de programação UNIX – Redireccionamento

Redireccionamento entre programas com *pipes* anónimos

Código exemplo (assumir exemplo: *ls* / *sort*)

No contexto do processo filho “que lê” (exemplo = para executar o *sort*)

```
close(0); // liberta posição de stdin (pos. 0 = STDIN_FILENO)
dup(mat[0]); // duplica extremidade de leitura do pipe para pos. 0
close(mat[0]); // fecha extremidade de leitura porque já foi duplicada
close(mat[1]); // fecha extrem. de escrita do pipe porque não a vai usar
//execlp("sort","sort",NULL);
```

Slides seguintes: código dos exemplos mostrados na aula com a máquina linux

Modelo de programação UNIX – Redireccionamento

Exemplo 1 (print2file.c) – Encaminha os seus “printfs” para um ficheiro

```
int main(int argc, char * argv[]) {
    int f;
    if (argc<3) {
        printf("\n\nfaltam parâmetros: texto ficheiro\n\n");
        exit(1);
    }
    close(STDOUT_FILENO);
    f = open(argv[2], O_WRONLY | O_CREAT,
             S_IRWXU | S_IRGRP | S_IROTH);
    if (f==-1) {
        perror("\n\nnão foi possível criar o ficheiro");
        exit(2);
    }
    printf("%s\n", argv[1]);
    return 0;
}
```

DE/S/ISEC

Sistemas Operativos – 2021/22

João Durães

Modelo de programação UNIX – Redireccionamento

Exemplo 2 (send2file.c) – Encaminha “printfs” de outro programa p/ fich.

```
int main(int argc, char * argv[]) {
    int f;
    if (argc<3) {
        printf("\n\nfaltam parâmetros\n\n"); exit(1);
    }
    close(STDOUT_FILENO);
    f = open(argv[2], O_WRONLY | O_CREAT,
             S_IRWXU | S_IRGRP | S_IROTH);
    if (f==-1) {
        printf("\n\nnão foi possível criar o ficheiro");
        exit(2);
    }
    execl(argv[1], argv[1],NULL);
    perror("\n\ncomando não encontrado ");
    close(f);
    return 3;
}
```

DE/S/ISEC

Sistemas Operativos – 2021/22

João Durães

Modelo de programação UNIX – Redireccionamento

Exemplo 3 (send2exe.c) – Encaminha stdout (“printfs”) de um programa

Parte 1 para stdin (“scansfs”) de outro programa

```
int main(int argc, char * argv[]) {
    int fpid;
    int mat[2];
    if (argc<3) {
        printf("\n\nfaltam parâmetros\n\n"); exit(1);
    }
    if ( pipe(mat) == -1 ) {
        printf("\n\nnão foi possível criar o pipe\n\n"); exit(2);
    }
    fpid = fork();
    if (fpid== -1) {
        printf("\nfork falhou\n\n"); exit(3);
    }
    // filho criado. A seguir: fazer o redir pai -> filho
```

Modelo de programação UNIX – Redireccionamento

Exemplo 3 (send2exe.c) – parte 2

```
if (fpid > 0) { // PAI - vai executar programa que escreve
    close (STDOUT_FILENO); // liberta stdout: STDOUT_FILENO=1
    dup(mat[1]); // duplica mat[1] (write) p/ lugar libertado
    close(mat[1]); // fecha este pq trabalha com o duplicado
    close(mat[0]); // fecha lado entrada pq n vai usar no pai
    execlp(argv[1], argv[1], NULL);
    // ocorreu erro
    perror("\n\nprograma 1 não encontrado : ");
}
```

Modelo de programação UNIX – Redireccionamento

Exemplo 3 (send2exe.c) – parte 3

```
if (fpid == 0) { // FILHO - vai executar programa que le
    close (STDIN_FILENO); // liberta stdin: STDOUT_FILENO = 1
    dup(mat[0]); // duplica mat[0] (read) p/ lugar libertado
    close(mat[0]); // fecha este pq trabalha com o duplicado
    close(mat[1]); // fecha lado saída pq não vai usar no pai
    execvp(argv[2], argv[2], NULL);
    // ocorreu erro
    perror("\n\nprograma 2 não encontrado : ");
}

return 4;
}
```

->> Nota: este exemplo não é representativo da Shell no sentido que o processo pai se “sacrifica” para executar o código do programa 1