NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

Aula 5

- Processos
  - fork/exec e redirecção de canais
- Interação entre processos
  - exemplo com sinais

- OSTEP: cap. 5
- Silberschatz, Operating Systems Concepts, 10th Ed. 4.6.2, C.3.2, C.3.3

2

Alterando os canais

- Onde o ls vai escrever agora?

```
switch (pid = fork()) {
  case 0:
        close(1);
        creat("meu-output", 0666);
        execl("/bin/ls","ls","/bin",NULL);
        perror("ls");
        exit(1);
  case -1:
        perror("fork");
        break;
  default:
        printf("Criei o %d\n", pid);
        wait(&st);
}
```

alterando o standard output

msg se erro no execl e saindo com status 1 (!=0)

onde escreve o ls?

Reconfiguranto o processo

- Várias operações alteram o processo. Exemplos:
  - é normal a execução alterar o cpu, dados e pilha; malloc, mmap podem alterar mapa de memória
  - close/open – alteraram os canais/ficheiros incluindo os *standard*
  - chdir() – altera diretoria corrente
  - setrlimit() – altera os limites dos recursos que pode usar (tempo de cpu, máximo de memória/pilha, máximo de ficheiros abertos, etc)
  - setenv() – altera variáveis de ambiente
- Se alterarmos antes de execve, o novo programa executa nesse ambiente

Exemplo: redireção de canais std

- Equivale a "**`wc -l <in.txt >out.txt`**"

```
switch (pid = fork()) {
  case 0:
        close(0);
        if (open("in.txt", O_RDONLY)==-1) {
            perror("in.txt"); exit(1);
        }
        close(1);
        if (creat("out.txt", 0666)==-1) {
            perror("out.txt"); exit(1);
        }
        execlp("wc","wc","-l",NULL);
        perror("wc");
        exit(1);
  case -1:
        perror("fork");
        break;
  default:
        wait(&st);
}
```

alterando o standard input

alterando o standard output

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

5

Relação libc / chamadas ao SO

- `printf` escreve em `FILE *stdout`
- `stdout` é o *stream* de libc (stdio.h) que representa o descritor 1 (*standard output*) do processo

`printf …` → **`… write(1, …)`**

- *semelhante para* ***`scanf`****,* ***`stdin`*** *e* ***`read`*** *de* ***`0`***

- Pedir o descritor do canal usado por um dado *stream* (FILE*):
  - `int fileno( FILE *stream )`
- Criar um *stream* C para o descritor de um canal já aberto:
  - `FILE *fdopen( int file, char *mode )`

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

6

Relação libc / chamadas ao SO

```
printf("primeira linha");
close(1);
g=creat("f-out", 0666);
printf("segunda linha");
```

onde aparece cada linha?

- *LibC tem buffers*
- *Cuidado ao misturar operações da biblioteca C com as respectivas chamadas ao SO!*

Não há dúvida: Processo ≠ Programa

- Programa é uma definição estática
  - Código, dados iniciais, endereço inicial, ...
- Processo é uma instância de execução com recursos:
  - memória com código e dados (estes vão variando)
  - estado CPU: pilha, registos (IP, SP, …)
  - canais de I/O e outros recursos
  - outros atributos
- O mesmo programa pode ser executado em diferentes processos, mesmo em simultâneo
- Um processo pode vir a executar vários programas

Execução de fork vs. execve

`fork`	`execve`
mapa mem. pai copiado para filho	mapa de mem. substituído pelo novo programa
retorna pid do filho ao pai (filho recebe zero)	nunca retorna (exceto em caso de erro)
pid filho diferente do do pai (processos diferentes!)	o pid mantém-se (mesmo processo!)
filho herda cópia dos canais de I/O do pai	mantém os mesmos canais (mesmo processo!)

Interações entre processos

- Os processos são concorrentes se não temos ordem prédefinida entre as suas ações (estas sobrepõem-se no tempo)
- Os processos são independentes se o que se passa num processo não afeta em nada o que se passa noutro (não interagem)
- Processos podem cooperar para um mesmo objetivo
  - por exemplo, fazem parte de uma aplicação ou sistema constituído por vários processos
  - um atende pedidos do outro (ex. serviço de impressão)
- Vantagens da cooperação entre processos
  - Partilha de informação
  - Aceleração das computações
  - Facilidade no desenvolvimento (Modularidade)
  - Natureza do problema a resolver (ex. Distribuição)

Os processos cooperantes

- Interações: comunicação vs. sincronização
- Comunicação: transferir informação/bytes entre processos.
  - Exemplo: browser / servidor web
- Sincronizar: notificar eventos, garantir ordem
  - Exemplo: aguardar pela terminação de outro processo
  - ou: controlar a ordem de acesso a recursos partilhados
- Pode acontecer que a comunicação envolva sincronização
  - exemplo: aguardar a chegada de dados de outro processo → receção síncrona

Comunicação entre processos?

- Tentativa de filho para pai por ficheiro:

```
p=fork();
if (p==0)
{       f=creat("/tmp/f",0666);
        write(f,"ola",4);
        ...
} else {
        f=open("/tmp/f",O_RDONLY);
        read(f, buf, BFSZ);
        ...
}
```

- Será que o pai lê o que o filho escreve?
- Só se garantir que o pai só lê depois do filho escrever! → sincronização

Comunicando por ficheiro

```
if ((p=fork())==0) {
      f=creat("/tmp/f",0666);
      write(f,"ola",4);
      exit(0);
} else if (p>0){
   waitpid(p,&st,0);//ponto de sincronização
   if(WIFEXITED(st) && WEXITSTATUS(st)==0)
   {    f=open("/tmp/f",O_RDONLY);
        read(f, buf, BFSZ);
    }
}
```


Comunicação entre processos?

- Tentativa de pai para filho por ficheiro:

```
p=fork();
if (p==0)
{       f=open("/tmp/f",O_RDONLY);
        read(f,buf,BFSZ);
        //... etc...
} else {
        f=open("/tmp/f",O_WRONLY|O_CREAT|O_TRUNC,0666);
        write(f, "ola", 4);
        // ...
}
```

- Como garantir que o filho só lê depois do pai escrever?

Notificações assíncronas

- Notificações de eventos ou de situações imprevisíveis/erros
- Assíncronas porque:
  - podem ou não ocorrer durante a execução
  - se ocorrerem, não se sabe quando
  - Exemplos:
    - erros inesperados no programa (violação da proteção de memória, divisão por zero, …)
    - ações desencadeadas pelo utilizador (ex: pede para abortar o processo)
    - um processo notifica outro sobre qualquer evento
- Podem interromper a execução normal
  - Desencadeia a execução de uma função de atendimento (handler)
- Permite a sincronização
  - se aguardando uma notificação

Sinais no sistema Unix

- Mecanismo software semelhante às interrupções do hardware
  - Inclui um identificador (número inteiro)
  - Os sinais são assíncronos. Podem acontecer em qualquer instante.
  - Permitem informar e reagir a situações imprevisíveis (ex. divisão por zero, violação da proteção de memória, Ctrl-C, …).
- Usado pelo SO e oferecido aos processos
  - o SO gera sinais em algumas situações
  - os processos podem enviar sinais a outros processos
  - o SO pode forçar a execução duma função no recetor (*handler*) quando um sinal ocorre (interrompe a execução normal)
    - vetor de *handlers* no descritor do processo (PCB/*task-struct)*

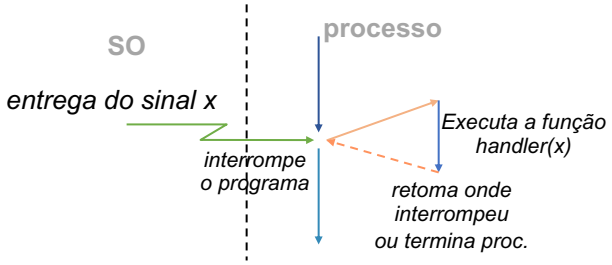
Geração de sinais

- Sinais são gerados pelo SO
- Como consequência de eventos
  - detetados pelo *hardware* :
    - FPU- divisão por zero; MMU- endereço inválido
  - situações detectados pelo SO:
    - terminação de um processo
- Ou por chamada ao sistema para gerar sinal:
  - `kill( )`
  - provocado pelo próprio processo:

```
alarm(int sec)
abort( )
```


Tratamento de sinais (recepção)

- Cada sinal tem um *handler.* Alguns podem ser alterados.



Tratamento de sinais (recepção)

- Interface mais simples para declarar *handler*:

```
typedef void (*sighandler_t)(int);
sighandler_t
    signal(int sig, sighandler_t handler);
```

- outra interface: **`sigaction()`**

- *"Handlers"* especiais (pré-definidos):
  - SIG_IGN: ignorar sinal (nota: nem todos os sinais podem ser ignorados)
  - SIG_DFL: o *handler default* oferecido pelo SO
- A ideia do *handler* é permitir alguma reação no destino:
  - p.ex. alterar *flags*; mudar contadores; etc.
  - Muitas chamadas ao SO podem ser interrompidas

Exemplo: tratar SIGCHLD

- Quando um processo termina o seu pai recebe o sinal SIGCHLD.
  - SIG_DFL: não fazer nada
  - Tratando sinal no pai, permite não bloquear nem ter zombies:

```
void tratafilho(int sig)
{ int pid=wait(NULL); // ignora status
  printf("%d terminou\n", pid);
}
...
int main(…)
{
  signal(SIGCHLD, tratafilho);
    ...
```

Ou

```
  signal(SIGCHLD, SIG_IGN); // ignorar filhos
```

Exemplo: tratar SIGINT

- Sinal SIGINT é entregue quando Ctrl-C
- SIG_DFL é terminar o processo
- Para ignorar:

```
signal( SIGINT, SIG_IGN );
```

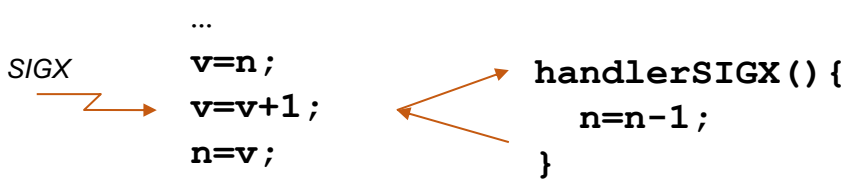
- Para definir um *handler:*

```
void cleanup(int s){
   printf("Received CTRL-C\n");
}

int main(){
   signal( SIGINT, cleanup );
...
```


Possíveis problemas …

- Os sinais são assíncronos, logo, podemos ter **problemas de concorrência** interna ao processo. Exemplo:



- Este tipo de problemas pode ocorrer sempre que há ações concorrentes **alterando** o mesmo recurso.
- Outro exemplo: um sinal interrompe printf() e no *handler* chama-se printf() … o que acontece?

Tratamento *default* (pré-definido)

- Para a maior parte dos sinais, o *default* é **terminar** anormalmente o processo
  - ex: os sinais para situações anómalas:
    - SIGFPE – erro detectado pela FPU
    - SIGSEGV – erro detectado pela MMU
    - SIGILL – erro detectado pelo CPU
    - SIGINT – pedido de terminação (p.ex. Ctrl-C no terminal)
    - SIGKILL – terminação (não pode ser mudado!)
- A terminação por um sinal de erro, pode gerar um *core dump* (ficheiro com a imagem binária do mapa de memória do processo, para *debug*)

Tratamento *default* (pré-definido)

- Para outros é **ignorar**
  - ex: SIGCHLD – um processo filho terminou
- Ou ainda parar/continuar o processo:
  - SIGSTOP – parar (não pode ser mudado!)
  - SIGCONT – continuar

  (Normalmente Ctrl-S, Ctrl-Q no terminal)

Enviar sinais

`int kill( pid_t pid, int signal )`

- Chamada ao sistema pedindo para enviar *signal* ao processo *pid*
- O SO verifica permissões (UID)

- A partir da *shell* existe o comando **`kill`**
  - Que faz a chamada anterior
  - Exemplo:

    Para terminar o processo 234

    `kill -9 234`

    `kill -KILL 234`

Terminação: wait e sinais

- Um processo que termina por um sinal não faz exit()
- O SO indica um estado especial de terminação para o pai e o número do sinal que o terminou
  - no pai este pode ser obtido assim:

```
pid_t p = wait( &status );
if ( WIFSIGNALED(status) )
   printf("o filho %d morreu com sinal %d\n",
                        p, WTERMSIG(status) );
```

Exemplo: aguardar um intervalo de tempo

- A função **`sleep(unsigned int nsec )`**
  - bloqueia o processo até que decorra o intervalo **`nsec`** segundos.
- Mas se quisermos que o processo seja avisado de que decorreu um certo intervalo de tempo, mas sem ter de ficar bloqueado à espera?
  - recorre-se a um sinal: SIGALRM

Exemplo de sinal de alarme

- Chamada ao SO:

  `int alarm(int nseg)`
  - pedir para gerar um sinal SIGALRM, ao próprio processo, após decorridos `nseg` segundos…
  - devolve o número de segundos que faltavam até o alarme anterior ocorrer ou 0 se nenhum estava definido
  - `alarm(0)`: desliga o alarme

Exemplo de sinal de alarme (2)

- O processo é notificado da ocorrência do alarme:
  - se o programa definiu um *handler*, esse é invocado, senão termina
- Isto permite invocar ações periodicamente
- E permite realizar *TIME-OUTs:*
  - interrompendo algumas chamadas ao sistema
    - por exemplo: se o processo estava bloqueado num read, a entrega do sinal de alarme permite desbloqueá-lo.

Exemplo: meuSleep

- Como obter o efeito do `sleep`?

```
void alarmHandler(int s)
{ }

void meuSleep( int s )
{  signal(SIGALRM, alarmHandler);
   alarm(s);
   pause(); /*bloqueia até sinal*/
   signal(SIGALRM, SIG_IGN);
}
```

fork/exec e os sinais

- Após fork():
  - o filho herda as definições dos sinais do pai;
  - mas não recebe um SIGALRM que o pai tenha pedido
- Após exec*():
  - todos os sinais que tiverem um *handler* definido pelo programa, passam à ação *default* do SO;
  - todos os outros sinais ficam com a mesma ação antes definida.

Comunicando por ficheiro+sinal

- Tentativa de pai para filho por ficheiro:

```
signal( SIGUSR1, nop );
if ((p=fork())>0){  // pai
       int f=creat("/tmp/f",0644);
       write(f,"ola",4);
       kill( p, SIGUSR1 );
       ...

} else if (p==0){  // filho
       pause();  // ponto de sincronização
       int f=open("/tmp/f",O_RDONLY);
       read(f, buf, BFSZ);
       printf("%s\n", buf);
       ...
} }
```

```
void nop(int s) {
}
```

e se o sinal pode chegar ao filho antes de pause()?

Comunicando por ficheiro?

- Pode ser uma solução
- Os ficheiros são tipicamente usados entre processos **não concorrentes**
  - a informação a partilhar fica guardara em ficheiro
- **O SO oferece mecanismos melhores e mais eficientes para processos concorrentes comunicarem...**