Sistemas Operativos: Processos

Pedro F. Souto (pfs@fe.up.pt)

March 2, 2012

Sumário: Processos

Conceito de Processo Sequencial

Multiprogramação

Chamadas ao Sistema

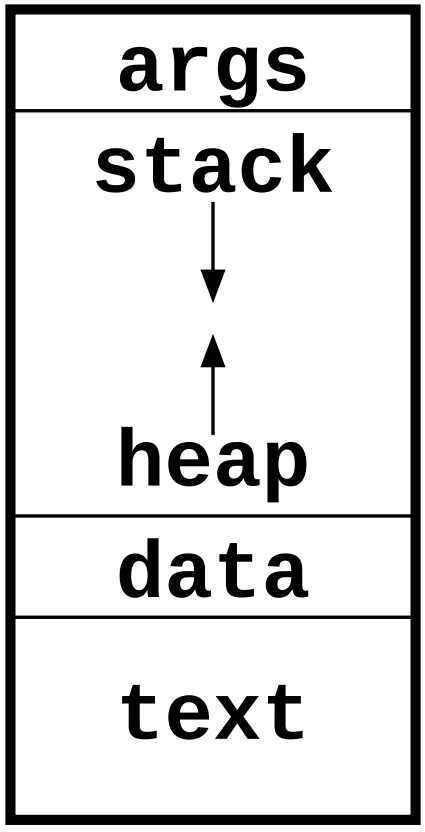
Notificação de Eventos

Leitura Adicional

Processo (Sequencial)

Representa um programa em execução

```
int main(int argc, char *argv[], char* envp[])}
```



`args` Argumentos passados na linha de comando e variáveis de ambiente.

`stack` *Registos de activação* correspondente à invocação de funções.

`heap` Dados alocados dinamicamente usando `malloc`.

`data` Dados alocados estaticamente pelo compilador (p.ex. a *string* "`Hello, World!`")

`text` Instruções do programa.

- O SO mantém ainda várias estruturas de dados com informação associada a cada processo

Stack

- Região de memória que *"pode ser acedida apenas"* numa das suas extremidades, usando as operações `push` e `pop` (Fig. 1).

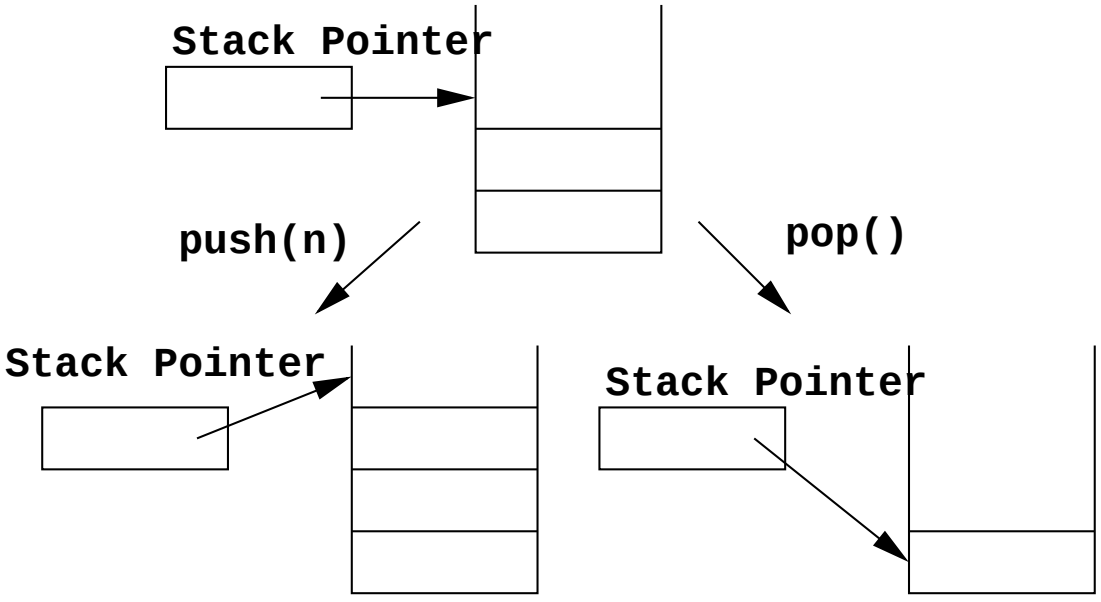


Fig. 1

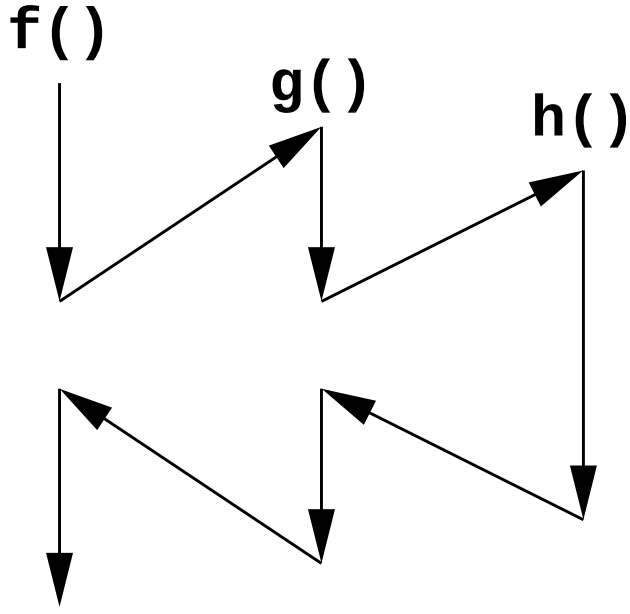


Fig. 2

- Este tipo de acesso é particularmente adaptado para implementar chamada a funções (Fig. 2).

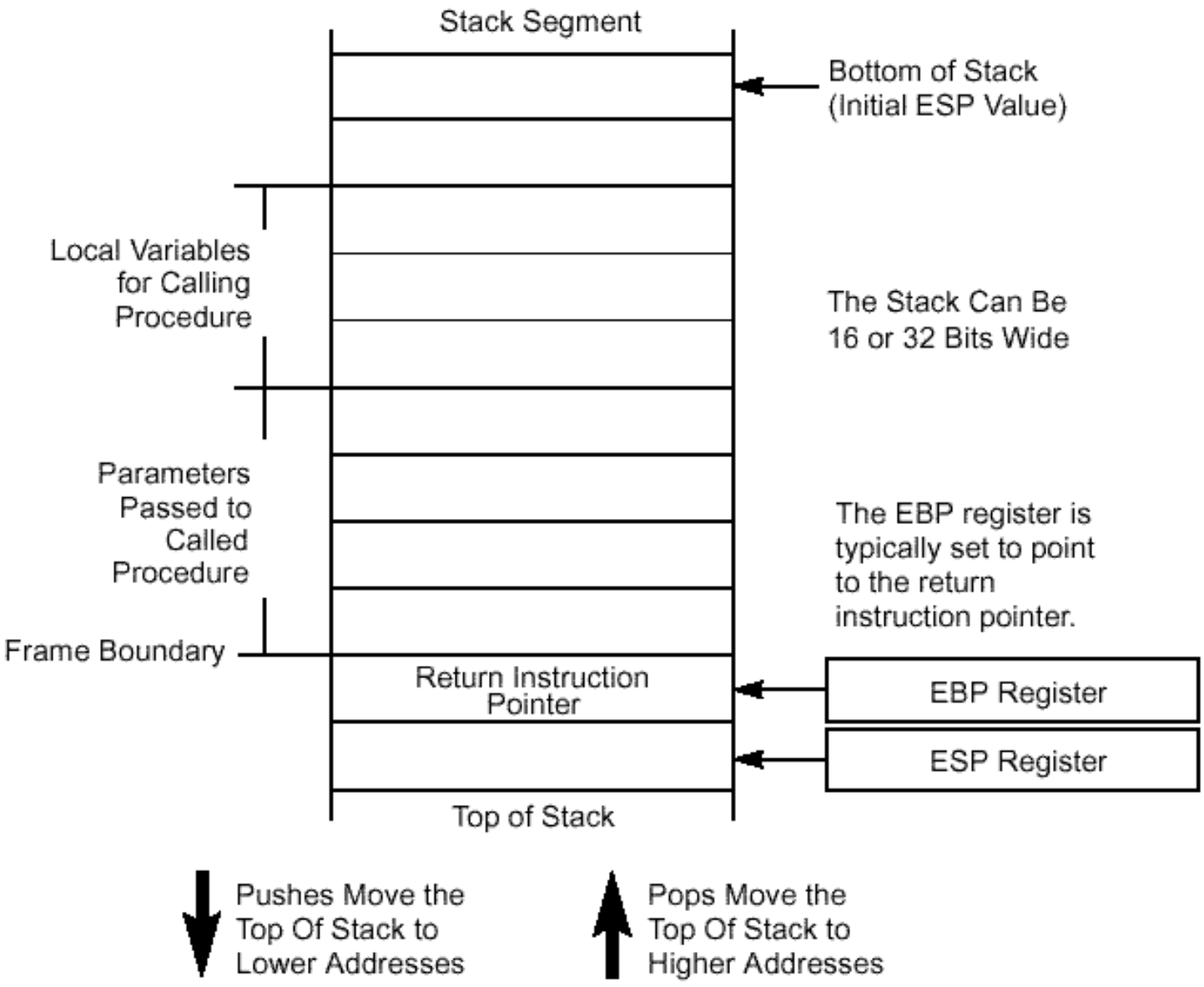
Uso da *stack* em funções de C

- Em C, o compilador usa a *stack* para:
  - passar os argumentos;
  - guardar o endereço de retorno;
  - implementar as variáveis locais;
  - passar o valor de retorno;

  duma função. E ainda para temporariamente:
  - guardar registos do processador

  usados por uma função.
- Cada compilador estabelece uma convenção sobre como esta informação é guardada na *stack*: a estrutura correspondente designa-se por *stack frame* ou *activation record*.

Stack Frame do 386



Sumário: Processos

Conceito de Processo Sequencial

Multiprogramação

Chamadas ao Sistema

Notificação de Eventos

Leitura Adicional

Unix/Linux são SOs multiprocesso (XP,Vista,...)

```
$ ps ax | more
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:04 /sbin/init
    2 ?        S      0:00 [kthreadd]
    3 ?        S      0:09 [ksoftirqd/0]
    6 ?        S      0:00 [migration/0]
   11 ?        S<     0:00 [cpuset]
   12 ?        S<     0:00 [khelper]
   13 ?        S<     0:00 [netns]
   15 ?        S      0:00 [sync_supers]
   16 ?        S      0:00 [bdi-default]
   17 ?        S<     0:00 [kintegrityd]
   18 ?        S<     0:00 [kblockd]
   19 ?        S<     0:00 [kacpid]
   20 ?        S<     0:00 [kacpi_notify]
   21 ?        S<     0:00 [kacpi_hotplug]
   22 ?        S<     0:00 [ata_sff]
--More-- (238 in all)
```

SOs suportam múltiplos processos (multiprogramação) por razões de **eficiência**, **conveniência** e **impaciência**.

Multi-processo e eficiência

Problema os dispositivos periféricos de entrada e saída de dados (consola, i.e. o monitor e o teclado, rato, disco, modem, placa de rede, etc) são muito mais lentos do que o processador (e a memória)

Parâmetro	Tempo
Ciclo do CPU	1 ns (1 GHz)
Acesso à *cache*	~ 2ns
Acesso à memória	~10 ns
Acesso ao disco	~10 ms

Solução quando um processo inicia uma operação de entrada/saída de dados e fica à espera que ela termine, o sistema operativo atribui o processador a outro processo:

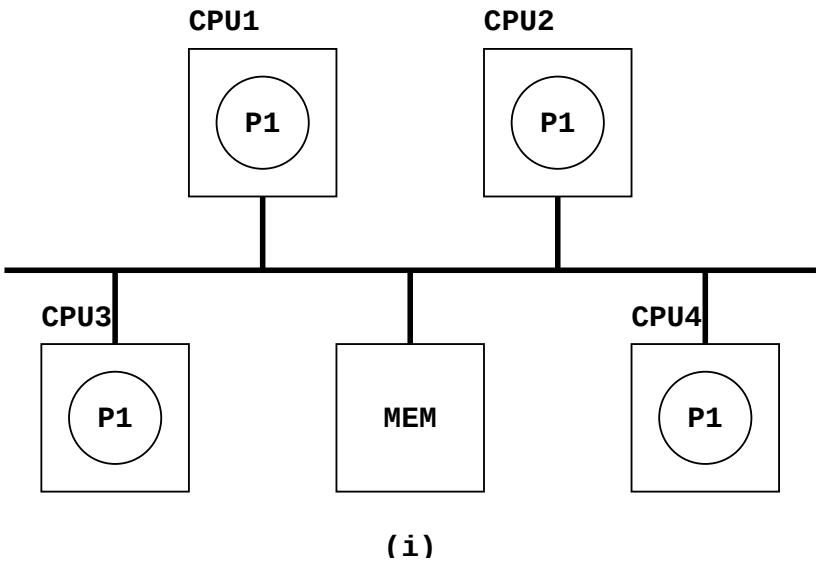
quando a operação terminar o periférico gera uma interrupção

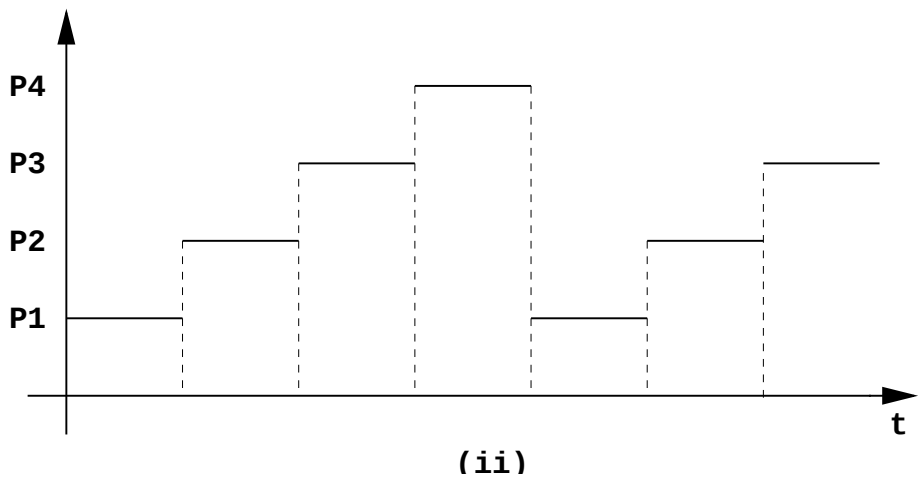
Multi-processo e conveniência

- Utilizadores frequentemente necessitam de usar diferentes programas “ao mesmo tempo”.
- A “especialização” dos programas facilita:
  - o seu desenvolvimento;
  - a sua reutilização.
- Quanto a **impaciência** . . .

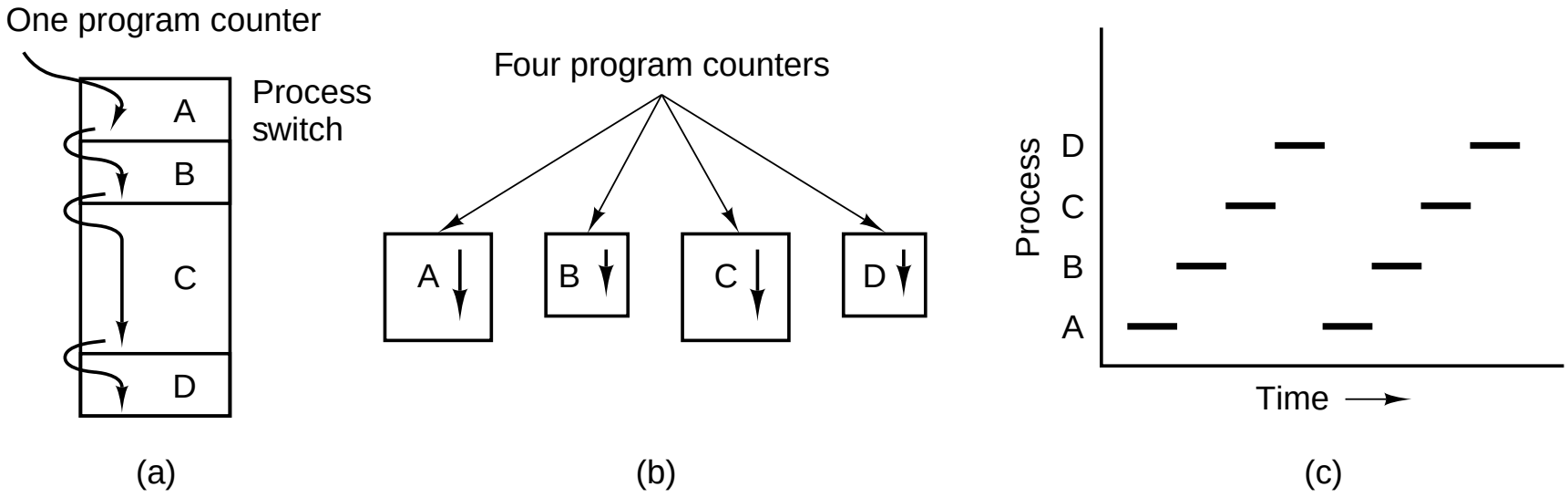
Execução multi-processo (1/2)

- Em sistemas multiprocessador (i), ou *multicore*, vários processos podem executar ao mesmo tempo, um em cada processador: **paralelismo real**
- Num sistema uniprocessador (ii), o sistema operativo gere a atribuição do processador aos diferentes processos (o processador é um recurso partilhado pelos diferentes processos): **pseudo-paralelismo**





Execução multi-processo (2/2)



- O computador é partilhado por 4 processos;
- O SO dá a ilusão de que cada processo executa isoladamente num CPU, i.e. cada processo executa num CPU virtual.

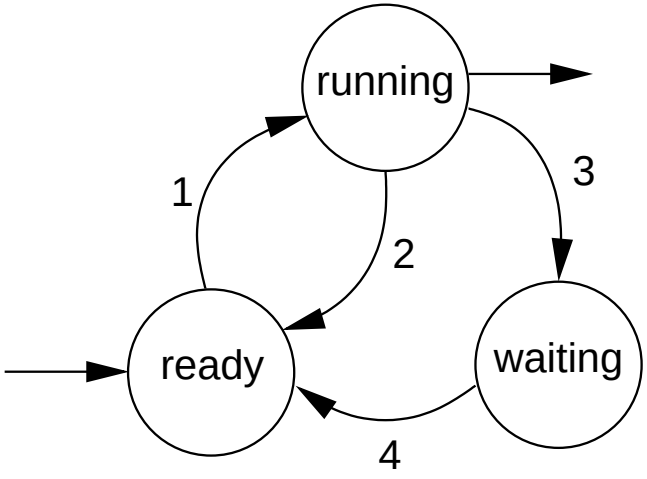
Estados dum Processo

- Ao longo da sua existência um processo pode estar em 1 de 3 estados:

  execução *(running)*: o CPU está a executar as instruções do processo;

  bloqueado *(waiting)*: o processo está à espera de um evento externo (tipicamente, o fim de uma operação de E/S) para poder prosseguir;

  pronto *(ready)*: o processo está à espera do CPU, o qual está a executar instruções de outro processo.



1. CPU atribuído ao processo (pelo SO);
2. CPU removido do processo (pelo SO);
3. processo bloqueia à espera dum evento;
4. ocorrência do evento esperado.

Processos: Segurança

Problema Como proteger:

- Os processos um dos outros;
- O SO dos processos

Solução

Espaço de endereçamento disjunto Por omissão o espaço de endereçamento dum processo é disjunto do dos restantes processos, e do espaço de endereçamento do *kernel*

Associação a um utilizador Cada processo está associado a um utilizador, o seu *dono*, podendo executar *apenas* as operações permitidas ao seu dono

Chamadas ao sistema O acesso a recursos do SO e ao HW é mediado pelo *kernel*.

Sumário: Processos

Conceito de Processo Sequencial

Multiprogramação

Chamadas ao Sistema

Notificação de Eventos

Leitura Adicional

Criação de Processos

- Durante o arranque do SO:
  - Normalmente estes processos são *não -interactivos* e designam-se por *daemons*:
    - Alguns executam sempre em *kernel space*, p.ex. `kswapd`;
    - Outros executam normalmente em *user space*, p.ex. o servidor de HTTP (Web) e o servidor de impressão.
  - Em SOs baseados *microkernel*, diferentes serviços do SO são fornecidos por processos especializados.
- Por invocação da chamada ao sistema apropriada.

Criação de Processos em Unix/Linux

```
#include <unistd.h>
pid_t fork(void);   /* clones the calling process *
```

- O processo criado (filho):
  - executa o mesmo programa que o programa pai;
  - inicia a sua execução na instrução que segue a `fork()`.
- O processo filho herda do pai:
  - o ambiente e “privilégios de acesso a recursos”;
  - alguns recursos, incluindo ficheiros abertos.
- Contudo, o processo filho tem os seus próprios
  - identificador;
  - espaço de endereçamento:

    após a execução de `fork()`, alterações à memória pelo *pai* não são visíveis ao *filho* e vice-versa.

Criação de Processos em Unix/Linux (cont.)

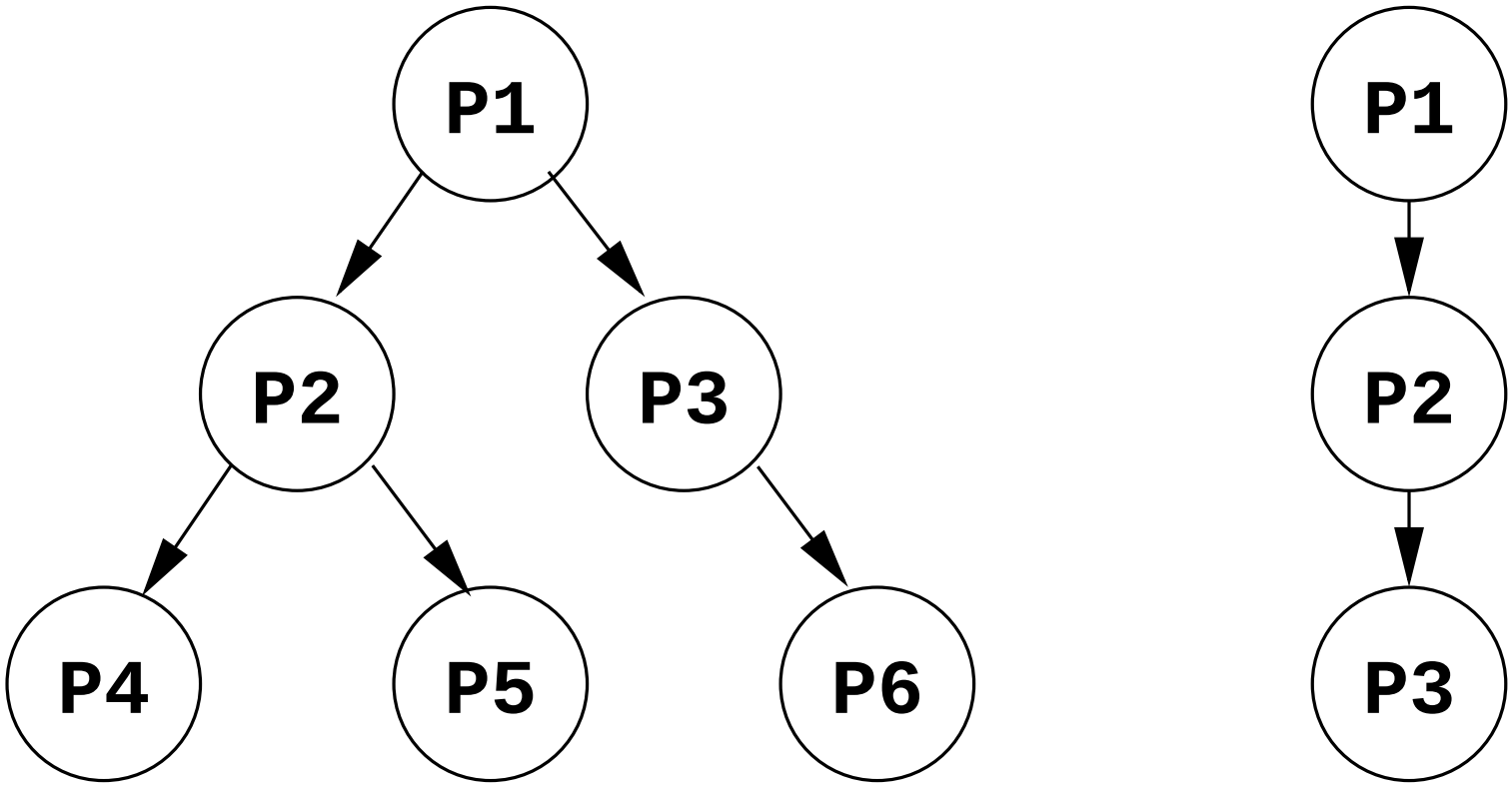
Problema: Como é que se distingue o processo pai do processo filho?

Solução: `fork()` retorna:

- o `pid` do filho ao processo pai;
- `0` ao processo filho.

```
if( (pid = fork()) > 0 ) {
   parent(pid); /* this is executed by the parent
} else if (pid == 0) {
   child();  /* and this by its child */
} else { /* pid == -1 */
   ... /* (parent) handle error */
}
```

Hierarquia de processos em Unix/Linux



- Em Unix/Linux há uma *relação especial*:
  - entre o processo pai e os seus filhos;
  - entre processos que têm um pai comum (*grupo de processos*).

pstree

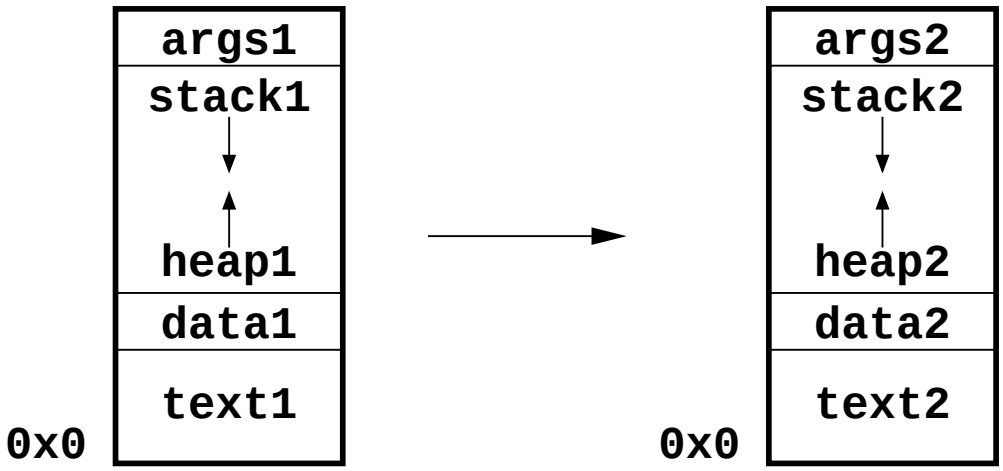
```
init-+--acpid
     +--atd
     +--battstat-applet
     +--bonobo-activati---{bonobo-activati}
     +--cron
     +--cupsd
     +--2*[dbus-daemon]
     +--dbus-launch
     +--dd
     +--events/0
     +--firefox-bin-+-acroread
     |              +-7*[{firefox-bin}]
     +--gconfd-2
     +--gdm---gdm-+-Xorg
     |            +-x-session-manag-+-gnome-cups-icon
     |                               +-gnome-panel---{gnome-pan
     |                               +-gnome-terminal-+-6*[bash
     |                               |                +-bash---
   [...]                           [...]             [...]
```

Execução de programas

Problema: Como é que um processo pode executar um programa diferente do do pai?

Solução: Usando a chamada ao sistema `execve()`

```
#include <unistd.h>
int execve(const char* filename, char *const argv[]
           char *const envp[])
```



- Substitui o programa em execução pelo contido em `filename`;
- `argv` e `envp` permitem especificar os argumentos a passar à função `main()` do programa a executar.

Terminação de Processos

- Um processo pode terminar por várias causas:
  1. decisão do próprio processo, executando a chamada ao sistema apropriada (directa ou indirectamente, p.ex. por retorno de `main()`);
  2. erro causado pelo processo, normalmente devido a um *bug*, p.ex. divisão por zero ou acesso a uma região de memória que não lhe foi atribuída;
  3. decisão de outro processo, executando a chamada ao sistema apropriada (`kill` em `POSIX`);
  4. decisão do SO (falta de recursos).
- Note-se que um processo pode terminar voluntáriamente quando detecta um erro, p.ex. um compilador não pode compilar um ficheiro que não existe.

Mais chamadas ao sistema

```
#include <unistd.h>
void exit(int status)
void _exit(int status)
pid_t getpid()
pid_t getppid()
```

- `_exit()` termina o processo que a invoca;
- `exit()` é uma função da biblioteca C (deve usar-se com `#include <stdlib.h>`):
  - invoca as funções registadas usando `at_exit()`;
  - invoca a chamada ao sistema `_exit()`.
- Após retornar de `main()`, um processo invoca `exit()`.
- `getpid()` retorna o `pid` do processo que o invoca.
- `getppid()` retorna o `pid` do processo pai.

Sincronização com `wait()`

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

- Suspende a execução do processo até:
  - um processo filho terminar (`wait()`);
  - o processo filho especificado em `pid` terminar (`waitpid()`);

  Pode retornar ainda quando da ocorrência outros eventos (ver *man page*)
- `status` permite obter informação adicional sobre o processo que terminou, incluindo:
  - O evento que causa o retorno
  - Se o retorno tiver sido via `_exit()`, o argumento que lhe foi passado
- `options` controla o modo de funcionamento de `waitpid()`.

Sincronização com `wait()`: exemplo

```
...
int status;
pid_t pid;
...
if( (pid = wait(&status)) != -1 ) {
   if( WIFEXITED(status) != 0 ) {
      printf(``Process %d exited with status %d\n'',
             pid, WEXITSTATUS(status));
   } else {
      printf(``Process %d exited abnormally\n'', pid);
   }
}
```

- `status` é um inteiro que codifica informação relativa ao evento que causou o retorno de `wait()`/`waitpid()`
- `WIFEXITED()` e `WEXITSTATUS()` são macros que permitem descodificar a informação contida em `status`
  - A *man page* lista e explica estas e outras macros

Sumário: Processos

Conceito de Processo Sequencial

Multiprogramação

Chamadas ao Sistema

Notificação de Eventos

Leitura Adicional

Eventos em Processos (Sequenciais)

Eventos (Sinais em Unix/Linux)

Excepções p.ex. divisão por 0, tentativa de acesso a uma posição de memória não válida

Notificação de eventos assíncronos p.ex., fim duma temporização ou da terminação dum processo filho, pressão dum conjunto de teclas

Comunicação entre processos Em Unix/Linux é possível um processo notificar um outro processo do mesmo dono

Processamento de Eventos

Signal Handlers são as rotinas de processamento de sinais

- Têm que ser registadas no *kernel* usando uma chamada ao sistema
- Podem executar assincronamente ao código do processo (tal como *interrupt handlers*)
  - Excepto no caso de **excepções**

Exemplo de Sinais POSIX (`man 7 signal`)

- Definidos no ficheiro `<signal.h>`
- Para cada sinal existe um processamento por omissão:
  Ignorar
  Terminar o processo notificado
  Suspender/retomar execução do processo notificado

Excepções

Nome simbólico	Descrição	Proc. por omissão
SIGILL	Instrução ilegal	Terminar
SIGSEGV	Acesso inválido à memória	Terminar
SIGFPE	Excepção na FPU	Terminar

Notificação de eventos assíncronos

Nome simbólico	Descrição	Proc. por omissão
SIGALRM	Fim de tempor. (`alarm()`)	Terminar
SIGCHLD	Filho terminou/foi suspenso	Ignorar

Comunicação entre processos

Nome simbólico	Descrição	Proc. por omissão
SIGSTOP	Suspender processo	Suspender
SIGUSR1	“Definível pelo utilizador”	Terminar

Conjuntos de Sinais em POSIX

- As chamadas ao sistema relativas a sinais usam frequentemente o tipo:

  `sigset_t` o qual abstrai um conjunto de sinais
  - Tipicamente, implementado como uma máscara de bits
  - Por isso, é frequente usar o termo ***signal mask***, em vez de ***signal set***
- POSIX especifica um conjunto de operações sobre conjuntos de sinais

```
int sigemptyset(sigset_t *set)
int sigfillset(sigset_t *set)
int sigaddset(sigset_t *set, int signo)
```

  - Para evitar erros, o 2º argumento deve ser uma constante simbólica definida em `<signal.h>` e não um inteiro

```
int sigdelset(sigset_t *set, int signo)
int sigismember(const sigset_t *set, int signo)
```

Alterar Processamento de Sinais POSIX com `sigaction()` (1/2)

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
struct sigaction {
              void     (*sa_handler)(int);
              sigset_t   sa_mask;
              int        sa_flags;
              ...
};
```

`sa_handler` *handler* para processamento de sinal. Nome duma função
cujo único argumento é um `int`, usado para passar o número/identificador do sinal a ser processado
que retorna `void` i.e. não retorna qualquer valor
Pode ainda ser uma das 2 constantes simbólicas:
`SIG_IGN` para ignorar o sinal
`SIG_DFL` para repôr o processamento por omissão

Alterar Processamento de Sinais POSIX com `sigaction()` (2/2)

`sa_mask` conjunto de sinais a **bloquear** durante execução do *handler*

- Enquanto um sinal está bloqueado, o SO não notifica o processo da sua ocorrência

`sa_flags` conjunto de *flags* que permitem modificar o comportamento dum sinal. P.ex. instalar a ação por omissão após a primeira execução do *handler*

- `sigaction()` pode ser usada:
  - não só para **alterar** o processamento, se `act` for diferente de `NULL`
  - mas também para **examinar** o processamento em vigor, se `oact` for diferente de `NULL`

  Se ambos os argumentos forem `NULL` permite alterar e examinar o processamento em vigor

Sincronização entre Processos com Sinais

- Para permitir a sincronização entre processos via sinais, o SO oferece uma chamada ao sistema para enviar sinais:
  `int kill(int pid, int sig)` envia o sinal `sig` para o processo cujo identificador é `pid`
  - Porquê chamar a esta chamada ao sistema `kill`?
- Um processo não pode enviar um sinal a qualquer outro processo
  - Só a processos cujo dono é o mesmo (i.e. "com o mesmo *user id*" que o seu)
- Um processo pode enviar qualquer sinal
  - Contudo fazê-lo não é muito boa ideia
    - Alguns sinais são enviados pelo SO quando ocorrem determinados eventos
  - POSIX reserva os sinais `SIG_USR1` e `SIG_USR2` para fins a determinar pela aplicação (*user defined*)

Outras Chamadas ao Sistema

`sigprocmask()`

- Permite examinar/alterar os sinais bloqueados
- Útil para impedir ***race conditions*** resultantes da execução assíncrona:
  - Do processo que envia um sinal
  - Do processo que o recebe
- Para analisar quando discutirmos concorrência

`sigsuspend()`

- Permite que um processo passe para o estado `WAIT` até receber um sinal que:
  - ou cause a execução dum *handler*
  - ou cause a sua terminação
- No primeiro caso, `sigsuspend()` retorna assim que o processo termine a execução do *handler*:
  - Retorna sempre -1
  - `errno` é inicializado com `EINTR`

Sumário: Processos

Conceito de Processo Sequencial

Multiprogramação

Chamadas ao Sistema

Notificação de Eventos

Leitura Adicional

Leitura Adicional

- Secções 3.1, 3.2, 3.3, 3,4 e 3.6 de
  José Alves Marques e outros, *Sistemas Operativos*,
  FCA - Editora Informática, 2009
- Secções 2 e 2.1 de
  A. Tanenbaum, *Modern Operating Systems*, 2nd Ed.
- Secções 3.1, 3.3 e 3.2 de
  Silberschatz e outros, *Operating System Concepts*,
  7th Ed.
- Outra documentação (transparências e enunciados dos TPs):
  via `http://web.fe.up.pt/~pfs/`