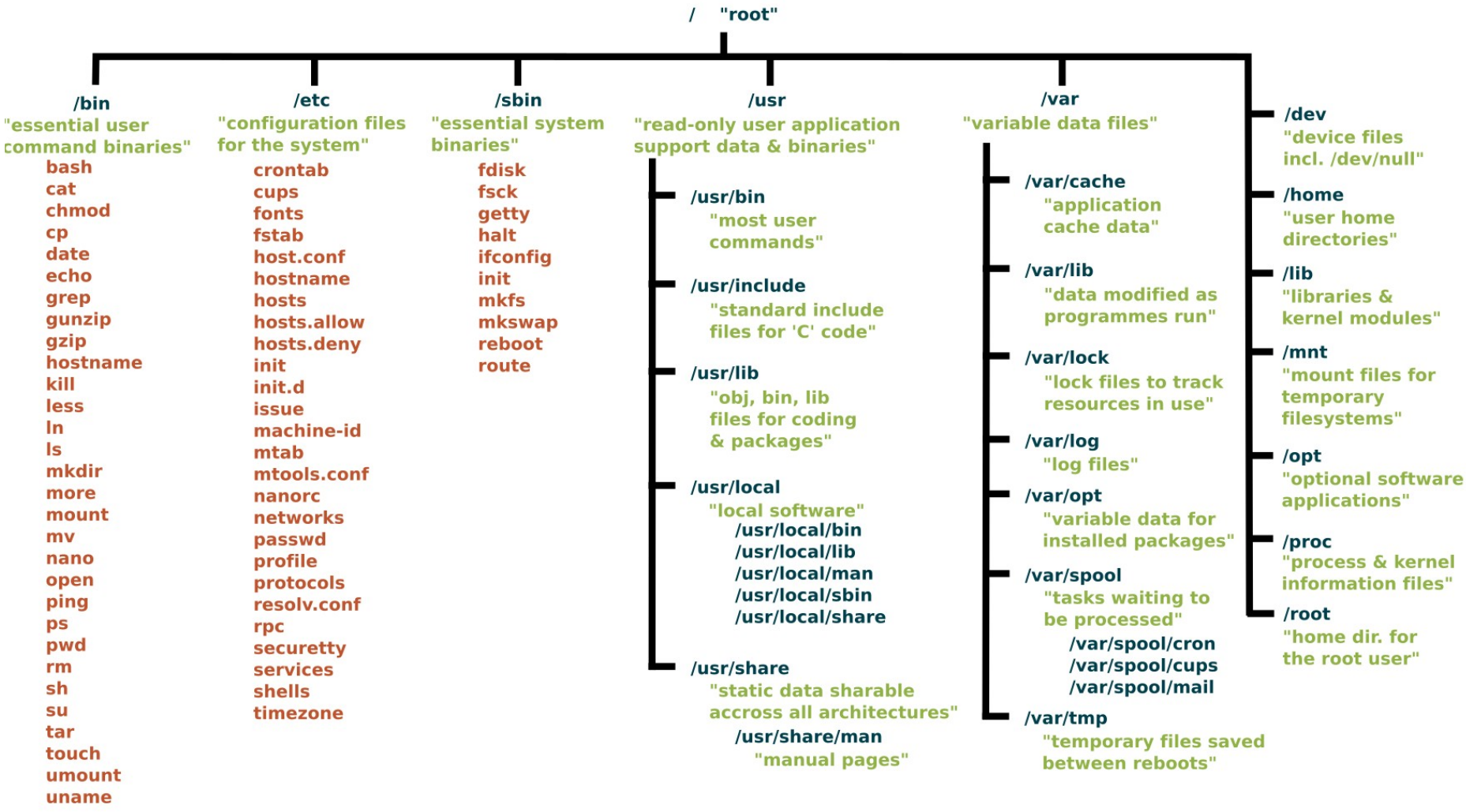
Essencial da Linha de Comando Linux

Sistema de Ficheiros (Exemplo)



Sistema de Ficheiros

Diretoria raiz/root (/)

A diretoria raiz / é o ponto de partida para toda a árvore hierárquica do sistema de ficheiros Linux

Subdiretorias

`/bin`	executáveis binários
`/boot`	ficheiros associados ao arranque do sistema
`/dev`	dispositivos ligados (usb, cdrom, rato, teclado)
`/etc`	ficheiros de configuração
`/home`	diretorias pessoais de cada conta de utilizador
`/lib`	bibliotecas de sistema
`/media`	montagem de dispositivos amovíveis (disquetes, cdrom)
`/mnt`	montagem de sistemas de ficheiros (nfs, smb)

Subdiretorias

`/opt`	software complementar
`/proc`	informação de processos/recursos do sistema
`/root`	diretoria inicial do administrador
`/run`	sistema de arquivos temporário para dados de execução
`/sbin`	executáveis binários utilizados pelo administrador
`/srv`	dados para serviços de servidor
`/sys`	sistema de ficheiros virtual para informações de hardware/drivers
`/tmp`	ficheiros temporários (a eliminar na reinicialização)
`/usr`	utilitários e dados/programas de utilizador de leitura apenas
`/var`	dados variáveis do sistema (logs, cache, filas, estados)

Linha de Comandos

Iniciar a linha de comandos: **`Ctrl+Alt+T`**

Não é necessário quando não temos GUI :)

`~$`

nome da diretoria atual

`~` significa a diretoria pessoal do utilizador

Ctrl+C, exit

```
$ ^C
```

Ctrl+C (representado por **^C**) é usado para parar o processo que está a correr

```
$ exit
```

termina a sessão atual e fecha a consola se não está aberta mais nenhuma sessão

Formato Genérico dos Comandos

`comando opções argumentos`

`comando`: nome do comando a executar ou caminho para o comando, no sistema de ficheiros

`opções`: afetam o comportamento do comando. Opções mono-carácter são usualmente precedidas de **`-`** , com possibilidade de combinar as várias opções:

`comando -a -b -c` ou **`comando -abc`**

Formato Genérico dos Comandos

```
comando opções argumentos
```

opções multi-carácter são usualmente precedidas de `--`

```
comando --opção
```

`argumentos`: itens (dados) necessários ao comando

Para correr vários comandos, usar `;` para os separar:

```
ls -la /tmp ; ls -l --all
```

man

```
$ ls --help
```

muitos comandos suportam a opção `--help`, que mostra uma descrição resumida da sintaxe

```
$ man ls
```

mostra o manual completo de comando para o comando "`ls`" (o manual é o local padrão para a documentação!)

Caminhos

```
$ cd /tmp
$ cd tmp
```

Caminho absoluto: começa com "/"

Caminho relativo: não

pwd

```
$ pwd
```

mostra o caminho absoluto para a diretoria corrente/de trabalho

cd

```
$ cd dir
```

define `dir` como nova diretoria de trabalho, sendo `dir` um caminho para a nova diretoria de trabalho

```
$ cd
```

define a nossa *home directory* como diretoria de trabalho

```
$ cd ..
```

"`..`" significa "a diretoria acima desta"

```
$ cd .; pwd
```

"`.`" significa "esta diretoria"

ls

```
$ ls
```

mostra/lista informação sobre objetos da diretoria atual

```
$ ls -la
```

l -> listagem dos objetos em formato longo

a -> lista todos os nomes, incluindo os objetos ocultos (iniciados por um ponto " . ")

mkdir

```
$ mkdir mydir
```

cria diretorias (vazias)

```
$ mkdir -p mydir/mysubdir
```

cria uma cadeia de diretorias, criando as intermédias se necessário

touch, nano, cat

```
$ touch myfile
```

podemos usar o **touch** para criar um ficheiro vazio

```
$ nano myfile
```

podemos editar **myfile** com o editor **nano** (**myfile** é criado, caso não exista)

```
$ cat myfile
```

mostra o conteúdo de **myfile**

Edição de código, compilação e execução

```
$ nano hello.c
```

editar

```
$ gcc hello.c -o hello
```

compilar

```
$ ./hello
```

executar

cp

```
$ cp myfile mycopy
```

copiar ficheiros e diretorias

```
$ cp -l myfile myhl
```

criar um *hard link* ao invés de uma cópia

```
$ cp -s myfile mysl
```

criar um link simbólico

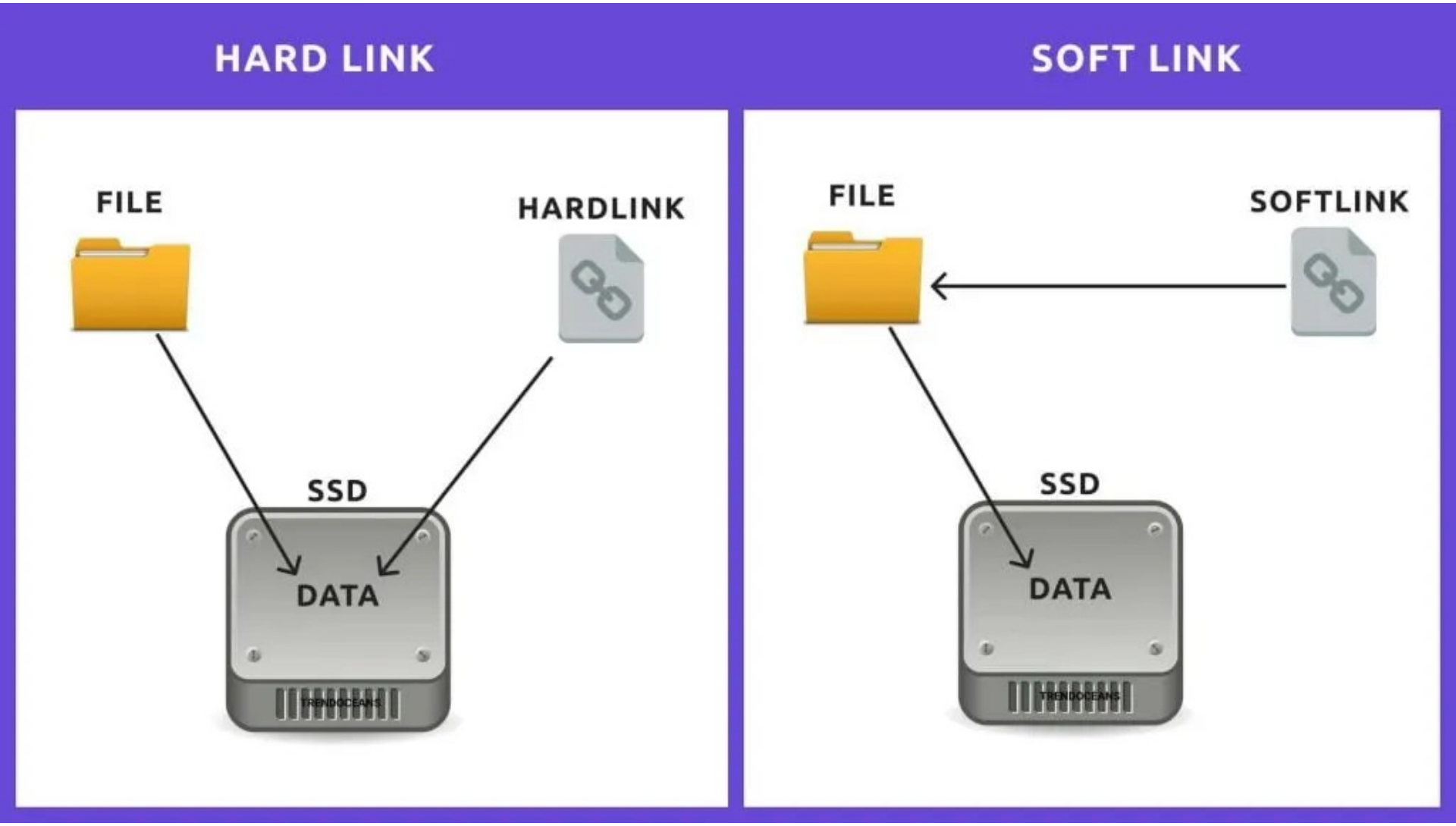
cp

Uma cópia é um novo ficheiro independente, com os seus próprios dados

Um hard link aponta para os mesmos dados do ficheiro original

Um soft link aponta para o caminho do ficheiro e é quebrado se o ficheiro original for eliminado

cp


HARD LINK
FILE
HARDLINK
SSD
DATA
SOFT LINK
FILE
SOFTLINK
SSD
DATA

mv

```
$ mv origin destination
```

mover ficheiros

rm, rmdir, rm -r

```
$ rm myfile
```

remover ficheiros

```
$ rmdir mydir
```

remover diretorias vazias

```
$ rm -r mydir
```

remover diretorias e os seus conteúdos recursivamente

who am i

```
$ whoami
```

mostra o nome de utilizador

```
$ id
```

imprime IDs de utilizadores e grupos

```
$ groups
```

imprimir os grupos a que o utilizador pertence

who

```
$ who
```

mostrar quem está logado

w

```
$ w
```

mostrar quem está logado e o que está a fazer

sudo

```
$ sudo command
```

executa o comando como utilizador root/super-utilizador

```
$ sudo bash
```

abre um novo shell com privilégios de root - mais poder, maior responsabilidade

Nota: Se o sudo não está instalado, será necessário fazê-lo! Ver:
https://www.cyberciti.biz/faq/how-to-install-and-configure-sudo-on-debian-linux/

apt install

```
$ su root
```

mudar para utilizador root

```
# apt install sudo
```

instalar o sudo

```
# nano /etc/sudoers
```

adicionar

```
student ALL=(ALL:ALL) ALL
```

gravar e sair

ls -l

```
$ ls -l
-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

- Tipo de ficheiro e permissões
- Contagem de links/atalhos
- Utilizador
- Grupo
- Tamanho
- Data da última alteração
- Nome do ficheiro

Tipo de Ficheiro

```
$ ls -l

-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

`-`	Ficheiro normal
`d`	Diretoria
`l`	Atalho

Permissões

```
$ ls -l

-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

r	read/leitura
w	write/escrita
x	execute/execução
-	a permissão está desabilitada para este campo

Nove definições de permissão

Permissões

```
$ ls -l
-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

Três conjuntos de três

Permissões

```
$ ls -l
-rwxr-xr-x 1 user1 group 573 Sep 24 22:07 myfile
```

Primeiro conjunto: proprietário/owner

```
read     yes
write    yes
execute  yes
```

Permissões

```
$ ls -l
-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

Segundo conjunto: grupo/group

```
read      yes
write     no
execute   yes
```

Permissões

```
$ ls -l
-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

Terceiro conjunto: todos os outros. Neste exemplo, temos:

```
read     yes
write    no
execute  yes
```

chmod

```
$ chmod XXX file
```

definir permissões para um ficheiro. XXX são três dígitos octais para permissões de proprietário, grupo e outros

```
$ chmod -R XXX dir
```

altera recursivamente as permissões de uma diretoria. `-R` aplica as permissões recursivamente a todos os ficheiros e subdiretorias

Nota: pode ser preciso o comando sudo se não tivermos permissão para modificar o ficheiro/diretoria.

Links/Atalhos

```
$ ls -l

-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

Número de links/atalhos.

Utilizador

```
$ ls -l

-rwxr-xr-x 1 user1  group 573 Sep 24 22:07 myfile
```

O proprietário do ficheiro.

Grupo

```
$ ls -l

-rwxr-xr-x 1 user1  group   573 Sep 24 22:07 myfile
```

O grupo a que pertence o proprietário do ficheiro.

Tamanho

```
$ ls -l

-rwxr-xr-x 1 user1  group   573 Sep 24 22:07 myfile
```

O tamanho deste ficheiro (em bytes).

Data da Última Alteração

```
$ ls -l

-rwxr-xr-x 1 user1  group    573 Sep 24 22:07 myfile
```

A última vez que o ficheiro foi alterado.

Nome

```
$ ls -l

-rwxr-xr-x 1 user1  group   573 Sep 24 22:07 myfile
```

O nome do ficheiro.

date

```
$ date
Mon Sep 23 12:29:21 AM WEST 2024
```

echo

```
$ echo hi
hi
```

Redirecionamento de Saída

```
$ date > mydate
$ cat mydate
```

Acrescentar

```
$ date >> log.txt
$ date >> log.txt
$ echo "End of log!" >> log.txt
$ cat log.txt
```

Útil para criar ficheiros de log, por exemplo

Pipes

```
$ ls -l | wc -l
```

Os pipes ligam a saída de um comando à entrada do comando seguinte

`wc -l` : conta o número de linhas de um ficheiro

Monitorização de Processos com o htop

- Instalar: **`sudo apt install -y htop`**
- Correr: **`htop`**
- Recursos: interativo, utilização por CPU/núcleo, gestão de processos
- Alternar entre consolas virtuais (ttys):
  - **`Ctrl + Alt + F1`** → tty1
  - **`Ctrl + Alt + F2`** → tty2
  - … até tty6 na maioria dos sistemas

Recursos Linux - Livro

The Linux Command Line, William Shotts, Fifth Internet Edition, 2019

Informação Útil

Configuração básica do teclado (Kernel e X)

https://wiki.debian.org/Keyboard#Basic_keyboard_configuration_.28Kernel_and_X.29

Como alterar o tamanho da fonte da consola

https://www.baeldung.com/linux/font-change-virtual-console

Exercício

- Consultar o manual de **mkdir**
- Criar as diretorias **OS/Lesson2**
- Mudar-se para a diretoria **Lesson2**
- Criar o ficheiro **hello.c** (usando o nano)
- Inprimir **"Hello World!\n"** em C
- Compilar e executar
- Apagar a diretoria **OS** e o seu conteúdo

Solução

- Consultar o manual de mkdir

  man mkdir

- Criar as diretorias OS/Lesson2

  mkdir -p OS/Lesson2

- Mudar-se para a diretoria Lesson2

  cd OS/Lesson2

- Criar o ficheiro hello.c (use o nano)

  nano hello.c

Solução

- Imprimir "Hello World!\n" em C

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

- Compilar e executar

```
gcc hello.c -o hello
./hello
```

Solução

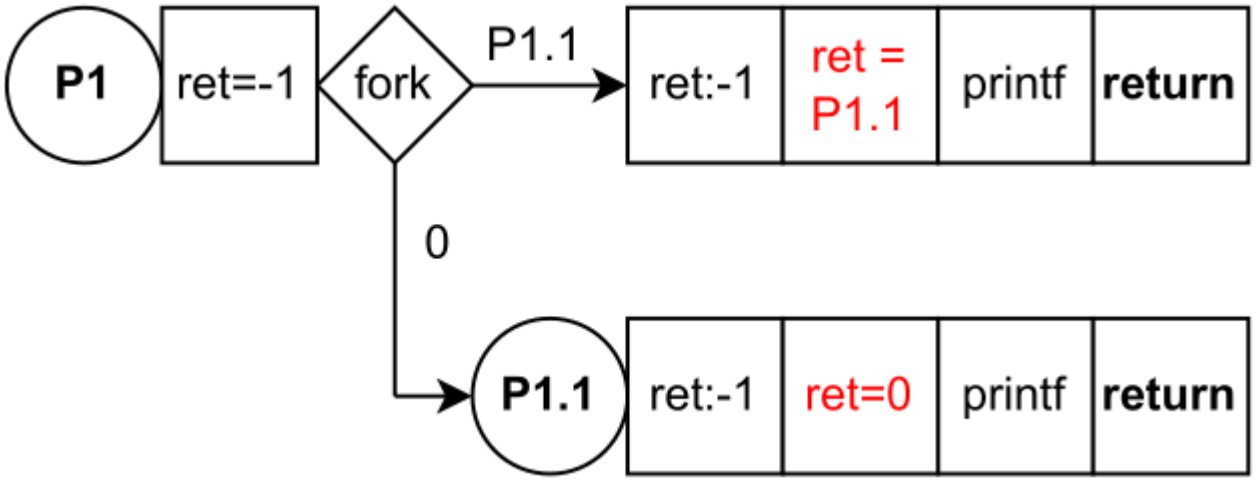
- Apagar a diretoria OS e o seu conteúdo

```
cd ../..
rm -r OS
```

Criação e Terminação de Processos (cont.)

Árvore de processos para o cenário (2) do Programa 1.3

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t ret=-1;

    // fork(); // (1)
    ret=fork(); // (2)
    printf("ret: %d\n", ret);
    return(0);
}
```

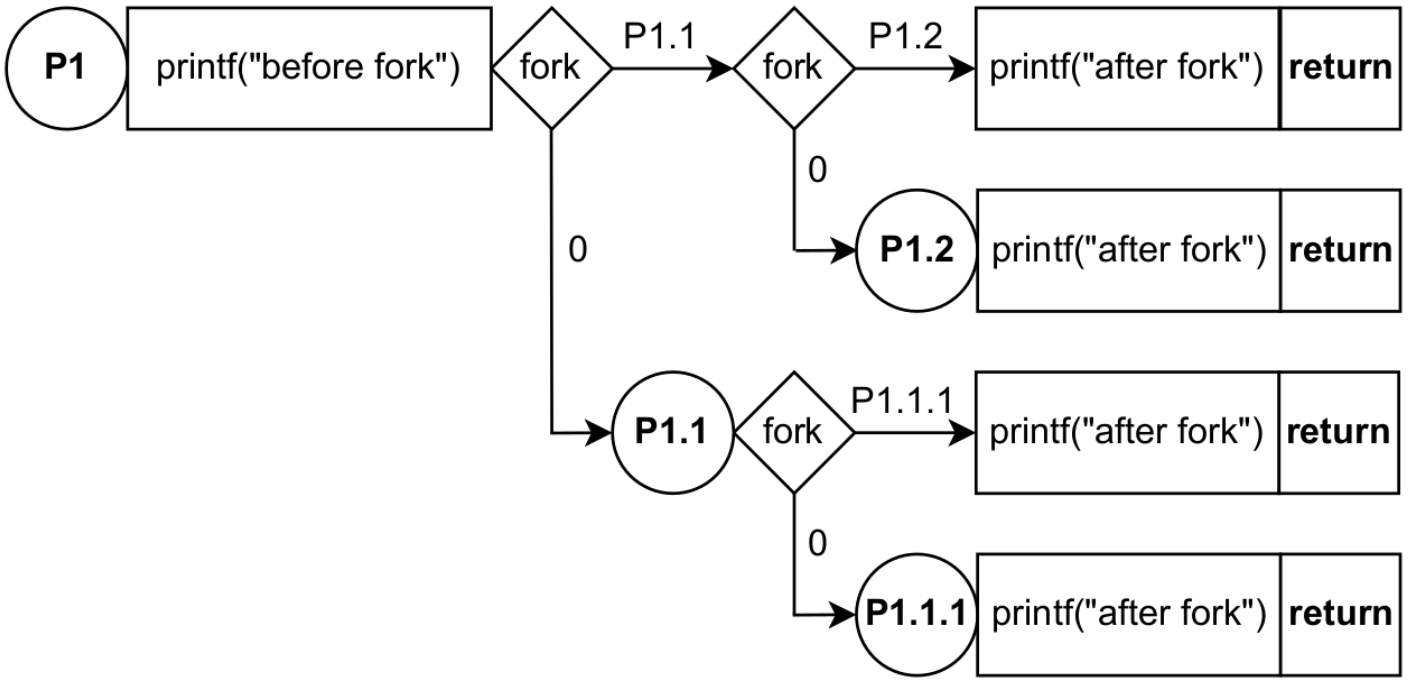


Árvore de processos para o cenário (1) do Programa 1.4

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

int main()
{
 printf("before fork\n"); // (1)
 //printf("pid %d: before fork\n", getpid()); // (2)
 //printf("pid %d (ppid %d): before fork\n", getpid(), getppid()); // (3)
 fork();
 fork();
 printf("after fork\n"); // (1)
 //printf("pid %d: after fork\n", getpid()); // (2)
 //printf("pid %d (ppid %d): after fork\n", getpid(), getppid()); // (3)
 return(0);
}
```

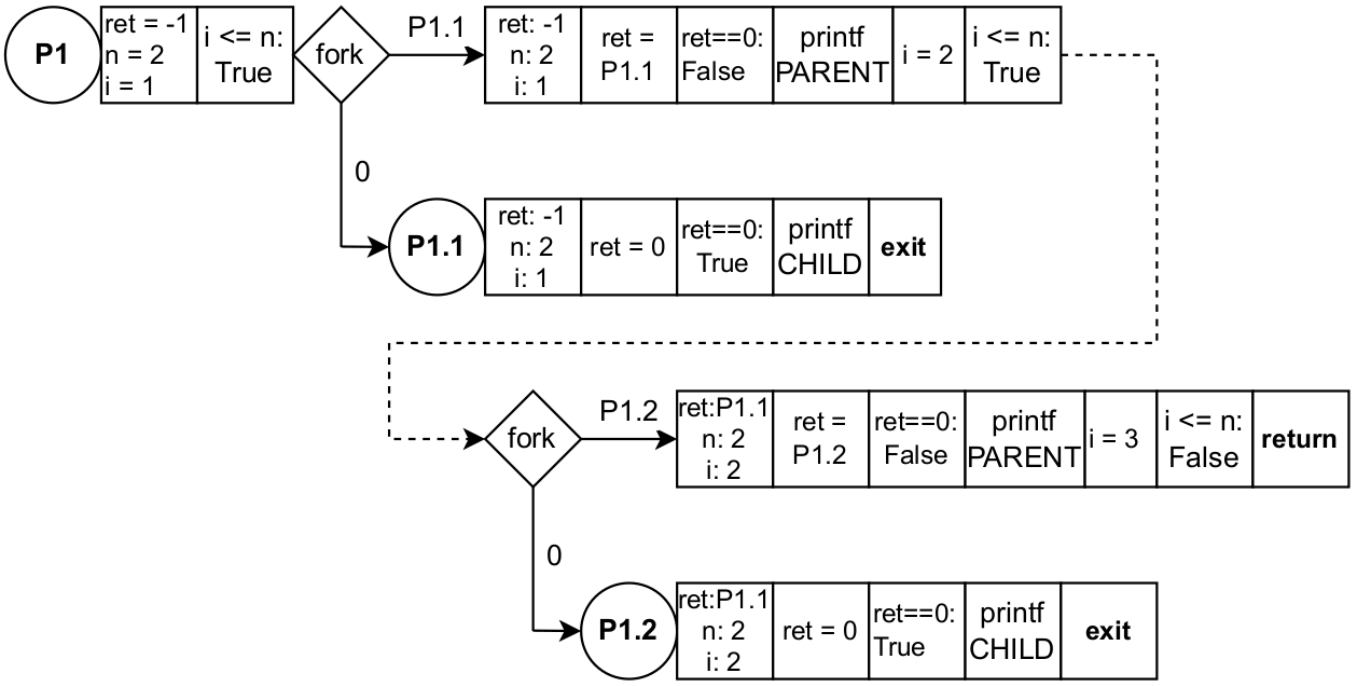
Árvore de processos para o cenário (1) do Programa 1.4

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

int main()
{
 printf("before fork\n"); // (1)
 //printf("pid %d: before fork\n"
 //printf("pid %d (ppid %d): befo
 fork();
 fork();
 printf("after fork\n"); // (1)
 //printf("pid %d: after fork\n",
 //printf("pid %d (ppid %d): afte
 return(0);
}
```



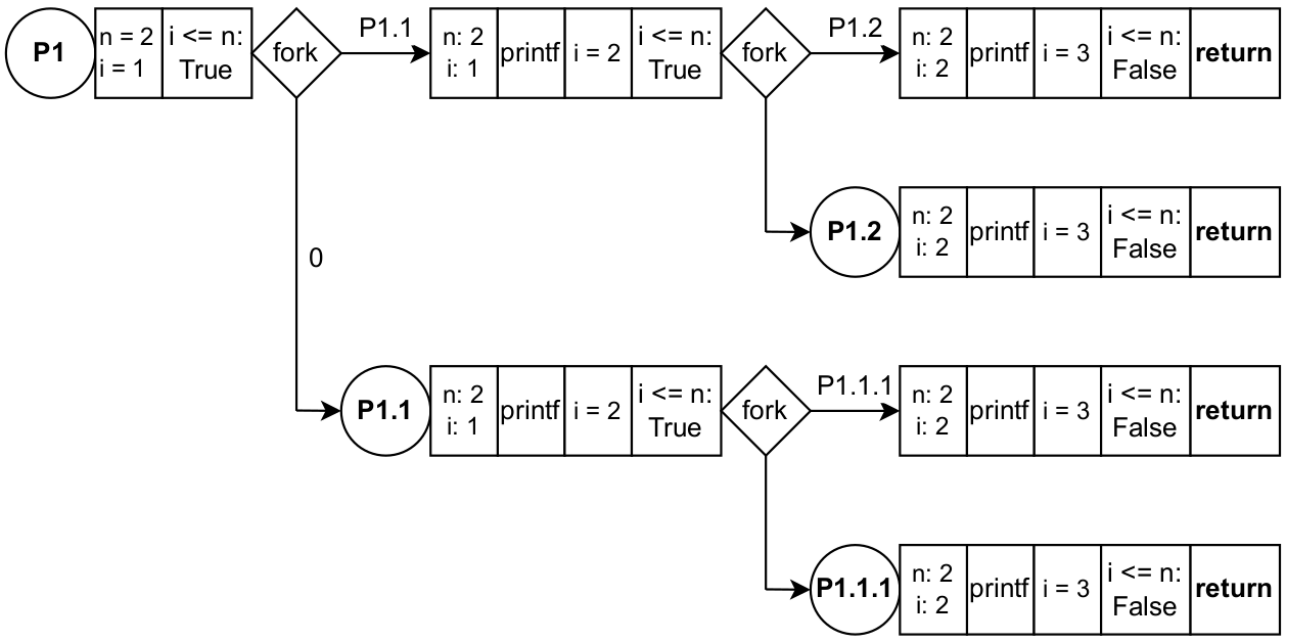
Árvore de processos para o Programa 1.7, com n=2

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

int main()
{
  pid_t ret=-1; int n=2, i=1;

  while (i<=n) {
      ret = fork();
      if (ret==0) {
          printf("CHILD %d ends\n", getpid());
          exit(0);
      }
      printf("PARENT %d continues\n", getpid());
      i++;
  }
  return(0);
}
```



Árvore de processos para o Programa 1.8, com n=2

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

int main()
{
  int n=2, i=1;

  while (i<=n) {
      fork();
      printf("i=%d \t pid=%d\n", i, getpid()); // (1)
      //printf("i=%d\tpid=%d\tppid=%d\n", i, getpid()
      i++;
  }
  return(0);
}
```

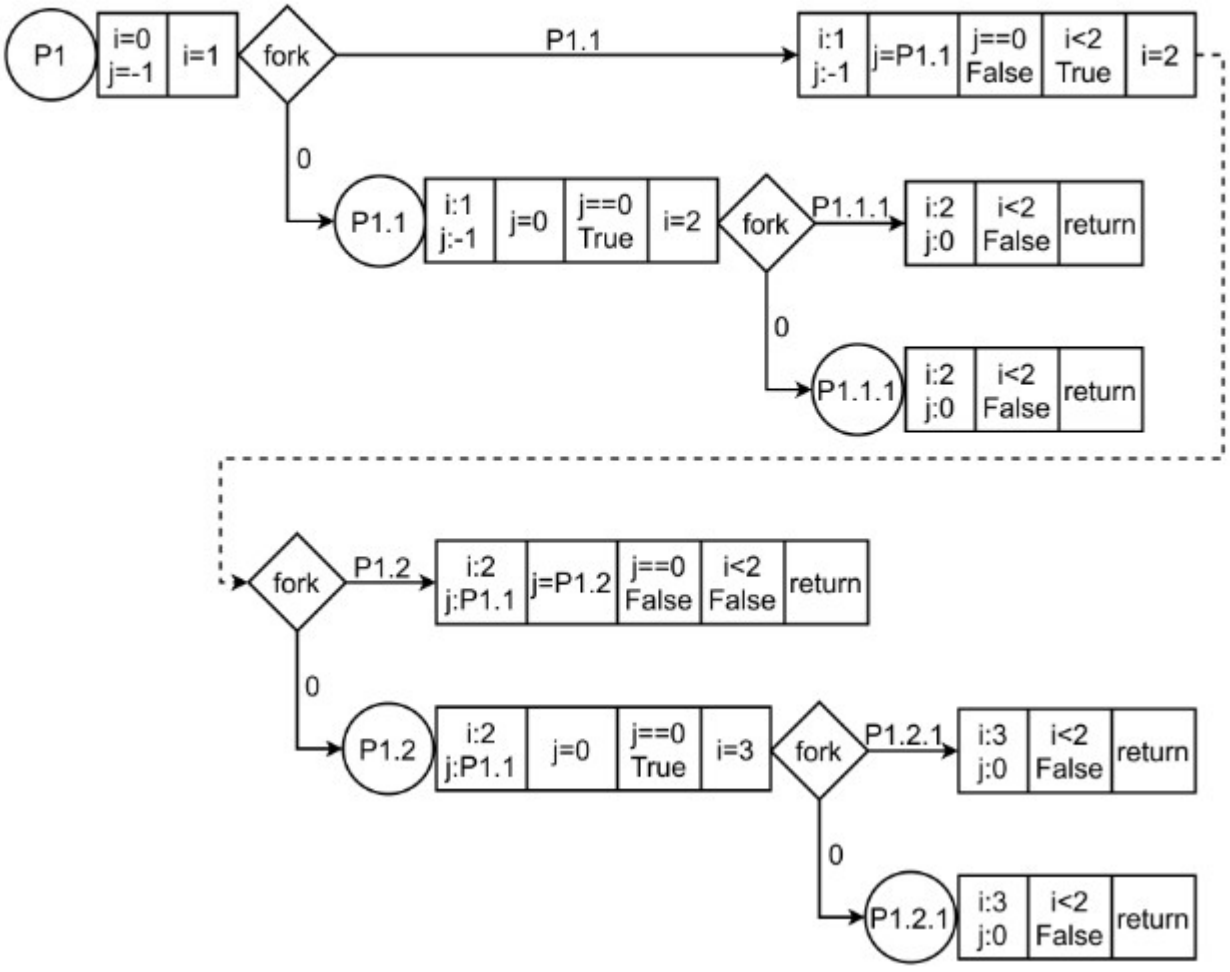


Exercício 1.2

2. (exame de 02/11/2015) Represente a árvore de processos gerada pelo seguinte código em C, incluindo na mesma a evolução do valor das variáveis `i` e `j`:

```
int i=0; pid_t j=-1;
do{
    i++;
    j=fork();
    if(j==0) {
        i++;
        fork();
    }
} while(i<2);
```

1.2


P1
i=0
j=-1
i=1
fork
P1.1
i:1
j:-1
j=P1.1
j==0
False
i<2
True
i=2
0
P1.1
i:1
j:-1
j=0
j==0
True
i=2
fork
P1.1.1
i:2
j:0
i<2
False
return
0
P1.1.1
i:2
j:0
i<2
False
return
fork
P1.2
i:2
j:P1.1
j=P1.2
j==0
False
i<2
False
return
0
P1.2
i:2
j:P1.1
j=0
j==0
True
i=3
fork
P1.2.1
i:3
j:0
i<2
False
return
0
P1.2.1
i:3
j:0
i<2
False
return

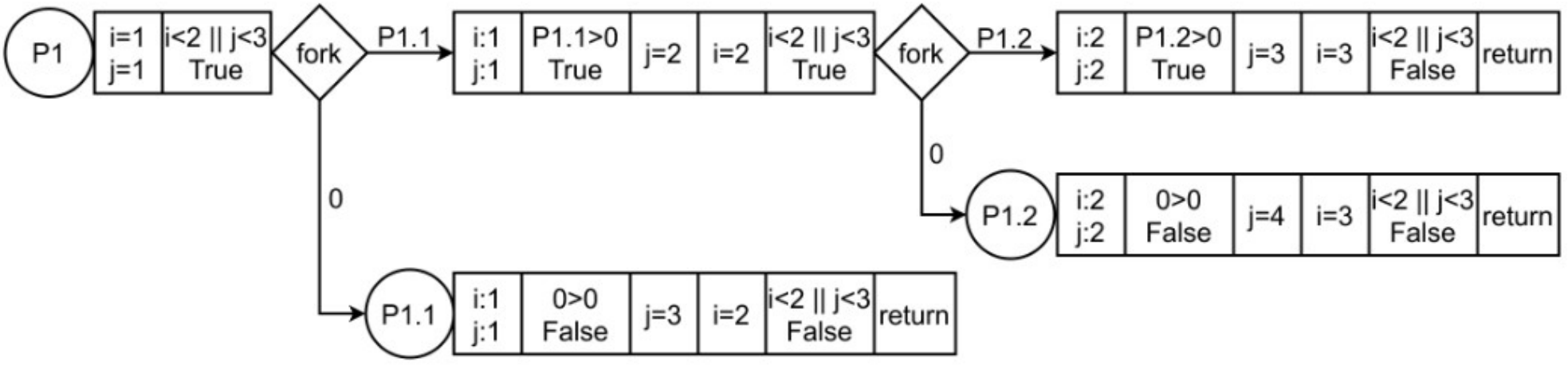
Exercício 1.3

3. (exame de 27/01/2011) Considere o seguinte fragmento de código em C:

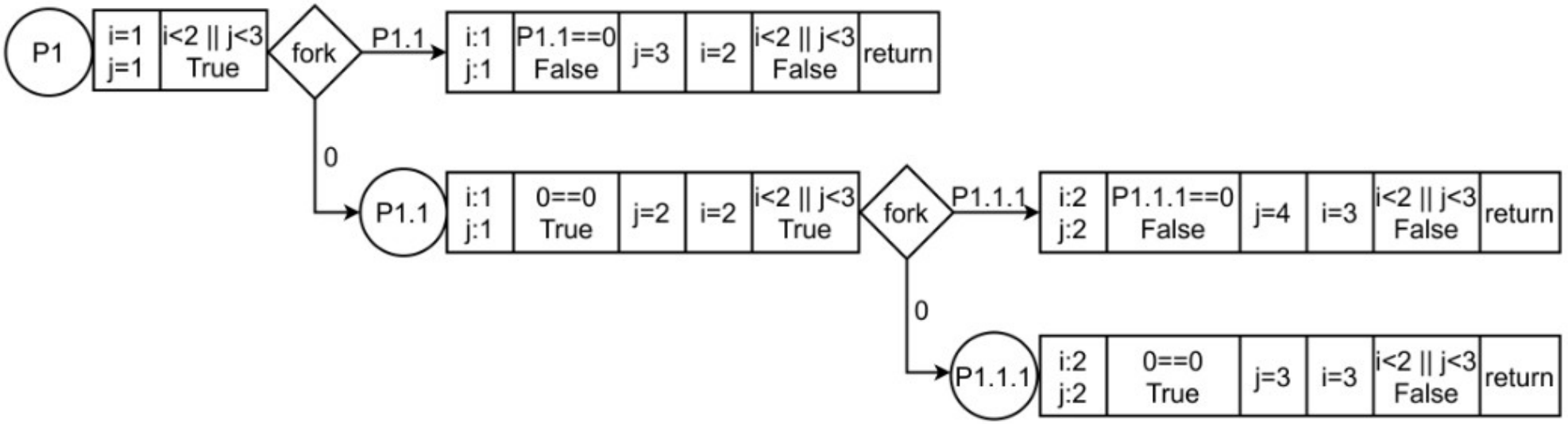
```
int i, j;
for (i=1, j=1; i<2 || j<3; i++)
    if( fork()>0 ) j+=1; else j+=2;
```

Represente a árvore de processos gerada pelo código anterior, incluindo na mesma a evolução do valor das variáveis `i` e `j`: a) Para o código tal como apresentado; b) Supondo que `if (fork()>0)` é substituído por `if (fork()==0)`.

1.3.a


P1
i=1
j=1
i<2 || j<3
True
fork
P1.1
i:1
j:1
P1.1>0
True
j=2
i=2
i<2 || j<3
True
fork
P1.2
i:2
j:2
P1.2>0
True
j=3
i=3
i<2 || j<3
False
return
0
P1.2
i:2
j:2
0>0
False
j=4
i=3
i<2 || j<3
False
return
0
P1.1
i:1
j:1
0>0
False
j=3
i=2
i<2 || j<3
False
return

1.3.b

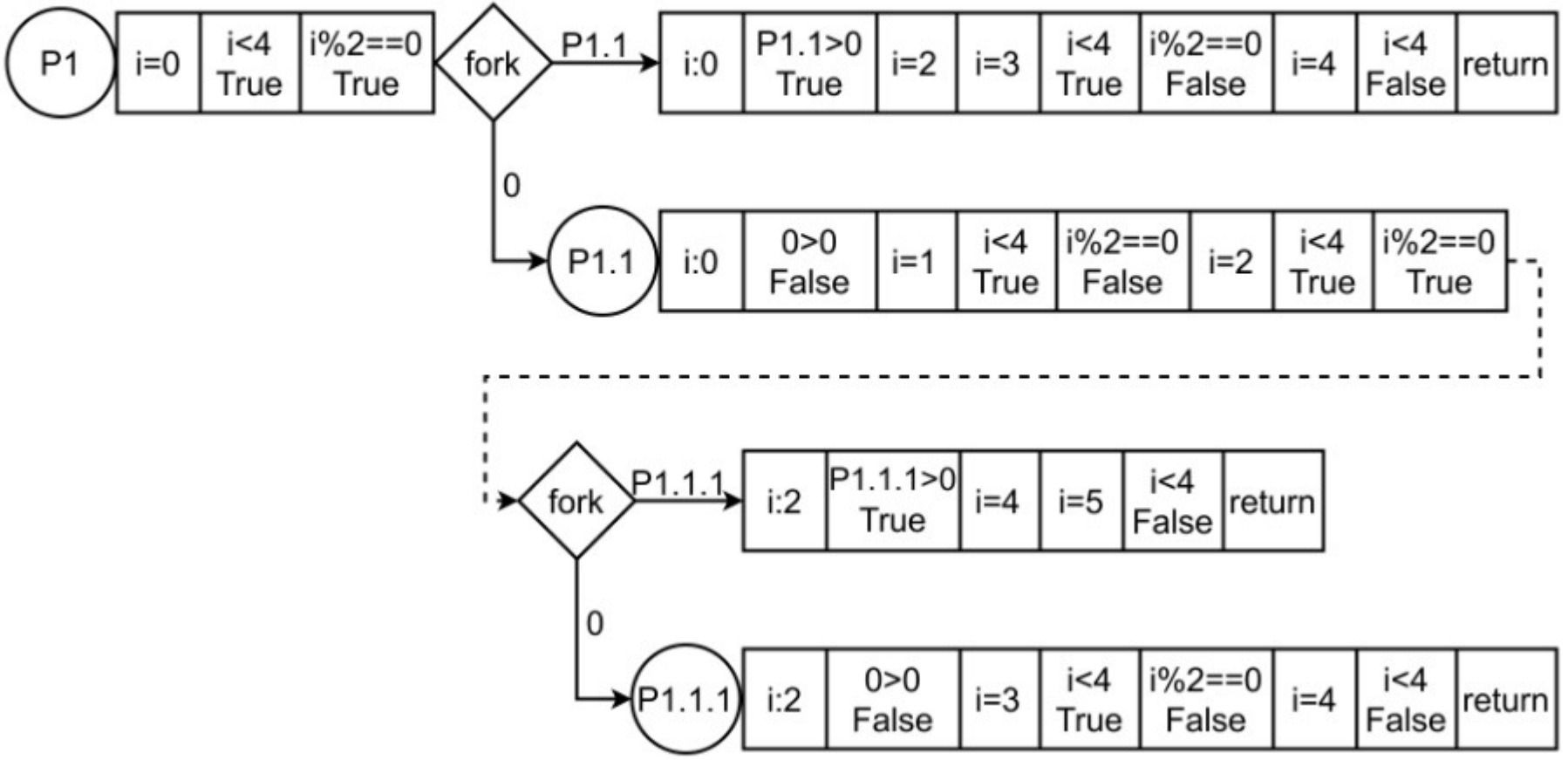

P1
i=1
j=1
i<2 || j<3
True
fork
P1.1
i:1
j:1
P1.1==0
False
j=3
i=2
i<2 || j<3
False
return
0
P1.1
i:1
j:1
0==0
True
j=2
i=2
i<2 || j<3
True
fork
P1.1.1
i:2
j:2
P1.1.1==0
False
j=4
i=3
i<2 || j<3
False
return
0
P1.1.1
i:2
j:2
0==0
True
j=3
i=3
i<2 || j<3
False
return

Exercício 1.4

4. (exame de 31/10/2013) Represente a árvore de processos gerada pelo seguinte código em C, incluindo na mesma a evolução do valor da variável `i`:

```
int i;
for (i=0; i < 4; i++) {
    if ((i % 2) == 0) {
        if (fork() > 0) {
            i += 2;
        }
    }
}
```

1.4


P1
i=0
i<4 True
i%2==0 True
fork
P1.1
i:0
P1.1>0 True
i=2
i=3
i<4 True
i%2==0 False
i=4
i<4 False
return
0
P1.1
i:0
0>0 False
i=1
i<4 True
i%2==0 False
i=2
i<4 True
i%2==0 True
fork
P1.1.1
i:2
P1.1.1>0 True
i=4
i=5
i<4 False
return
0
P1.1.1
i:2
0>0 False
i=3
i<4 True
i%2==0 False
i=4
i<4 False
return

forkSim

- **http://forksim.estig.ipb.pt:8080**
- username: so
- password: so20232024

Sincronização na Terminação de Processos

Sincronização na Terminação de Processos

- **Um processo pai pode bloquear** (espera passiva, no estado bloqueado), **aguardando pela terminação de um filho** (um qualquer, ou um filho específico), antes de prosseguir a sua execução
- **Um processo filho também pode sincronizar-se na terminação do seu pai original**, neste caso recorrendo a uma **espera ativa** (polling constante ou periódico, de uma flag ou valor de retorno)

wait e waitpid

- #include <sys/types.h>

  #include <**sys/wait.h**>

  pid_t **wait**(int ***status**);

  pid_t **waitpid**(pid_t **pid**, int ***status**, int **options**);
- A primitiva wait bloqueia um processo pai até que este: a) receba um sinal, ou b) um dos seus filhos termine

wait e waitpid

- A primitiva **wait** recebe como parâmetro o endereço de um inteiro status, onde será depositado o estado de saída do filho que terminou
- Para extrair o valor de retorno a partir do estado de saída é necessário recorrer à macro **WEXITSTATUS**
- Se o pai não está interessado no estado de saída do filho, invoca simplesmente **wait(NULL)**
- O valor retornado por wait é o PID do filho que terminou. Se um processo pai não tem filhos, wait retorna imediatamente -1. Assim, para esperar explicitamente pela terminação de todos os filhos, basta que o pai execute **while(wait(NULL) != -1)**

wait e waitpid

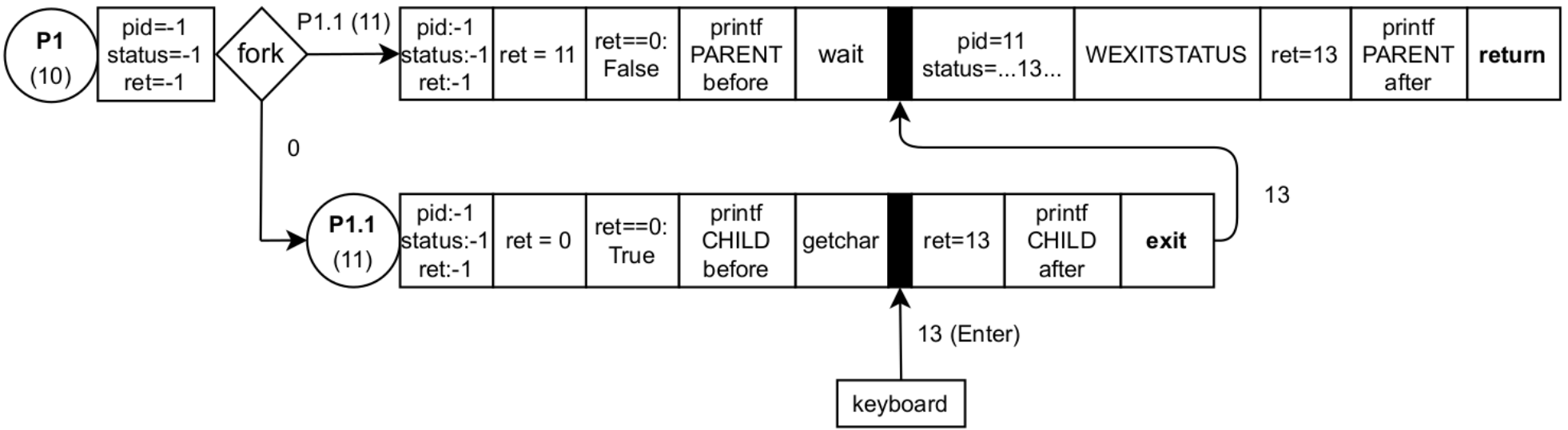
- A primitiva **waitpid** é semelhante à primitiva wait, mas permite especificar, como primeiro parâmetro, **o pid do processo filho cuja terminação se aguarda**
- O segundo parâmetro é o endereço de um inteiro **status** para receber o estado de saída
- Um terceiro parâmetro, **options**, permite especificar opções adicionais (por exemplo, se options=WUNTRACED, waitpid também retorna se o filho foi parado, mas não necessáriamente terminado)
- Em geral, o segundo e terceiro parâmetros assumem o valor **NULL** e **0**, respetivamente

wait

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

int main() {
 pid_t pid=-1; int status=-1, ret=-1;

 ret=fork();
 if (ret == 0) {
   printf("CHILD %d before getchar\n", getpid());
   ret=getchar(); // what would happen without getchar ?
   printf("CHILD %d after getchar\n", getpid());
   exit(ret);
}

 printf("PARENT before wait\n");
 pid=wait(&status); // PARENT blocks until the CHILD ends
 ret=WEXITSTATUS(status);
 printf("PARENT after wait: CHILD %d ended with return code %d\n", pid, ret);
 return(0);
}
```

wait

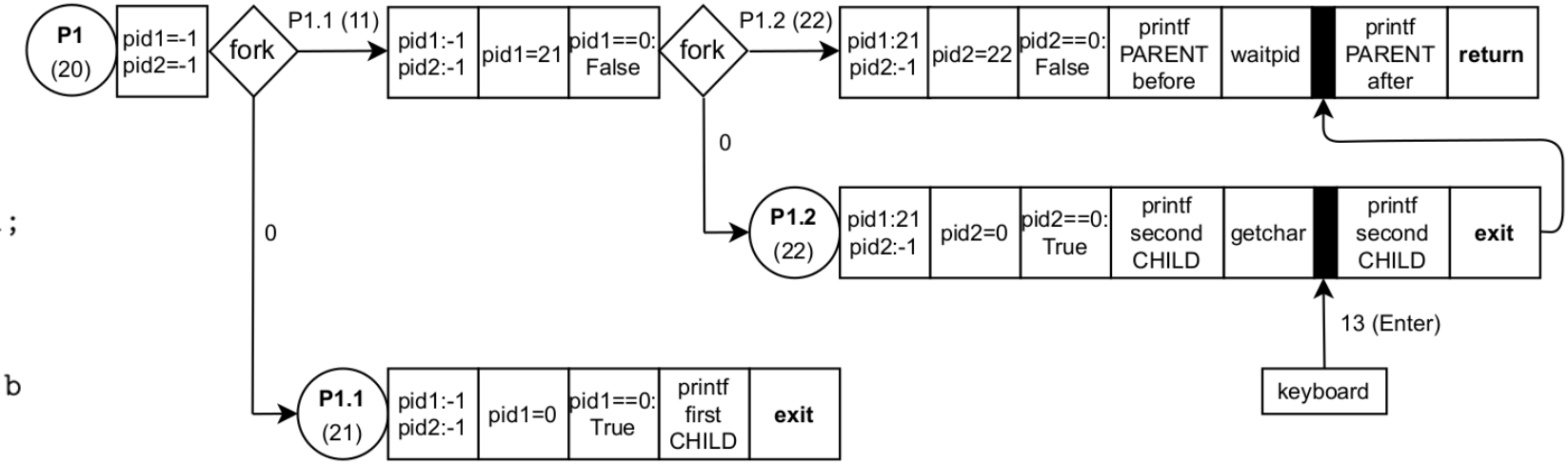
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
 pid_t pid=-1; int status=-1, ret=-1;

 ret=fork();
 if (ret == 0) {
   printf("CHILD %d before getchar\n", getpid());
   ret=getchar(); // what would happen without getchar ?
   printf("CHILD %d after getchar\n", getpid());
   exit(ret);
}

 printf("PARENT before wait\n");
 pid=wait(&status); // PARENT blocks until the CHILD ends
 ret=WEXITSTATUS(status);
 printf("PARENT after wait: CHILD %d ended with return code %d\n", pid, ret);
 return(0);
}
```



waitpid

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

int main() {
 pid_t pid1=-1, pid2=-1;

 pid1=fork();
 if (pid1 == 0) {
   printf("first CHILD before exit\n");
   exit(0);
 }

 pid2=fork();
 if (pid2 == 0) {
   printf("second CHILD before getchar\n");
   getchar(); // what would happen without getchar ?
   printf("second CHILD after getchar\n");
   exit(0);
 }

 printf("PARENT before waitpid\n");
 waitpid(pid2, NULL, 0); // PARENT blocks until second CHILD ends
 printf("PARENT after waitpid\n");
 return(0);
}
```

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

int main() {
 pid_t pid1=-1, pid2=-1;

 pid1=fork();
 if (pid1 == 0) {
   printf("first CHILD b
   exit(0);
 }

 pid2=fork();
 if (pid2 == 0) {
   printf("second CHILD before getchar\n");
   getchar(); // what would happen without getchar ?
   printf("second CHILD after getchar\n");
   exit(0);
 }

 printf("PARENT before waitpid\n");
 waitpid(pid2, NULL, 0); // PARENT blocks until second CHILD ends
 printf("PARENT after waitpid\n");
 return(0);
}
```



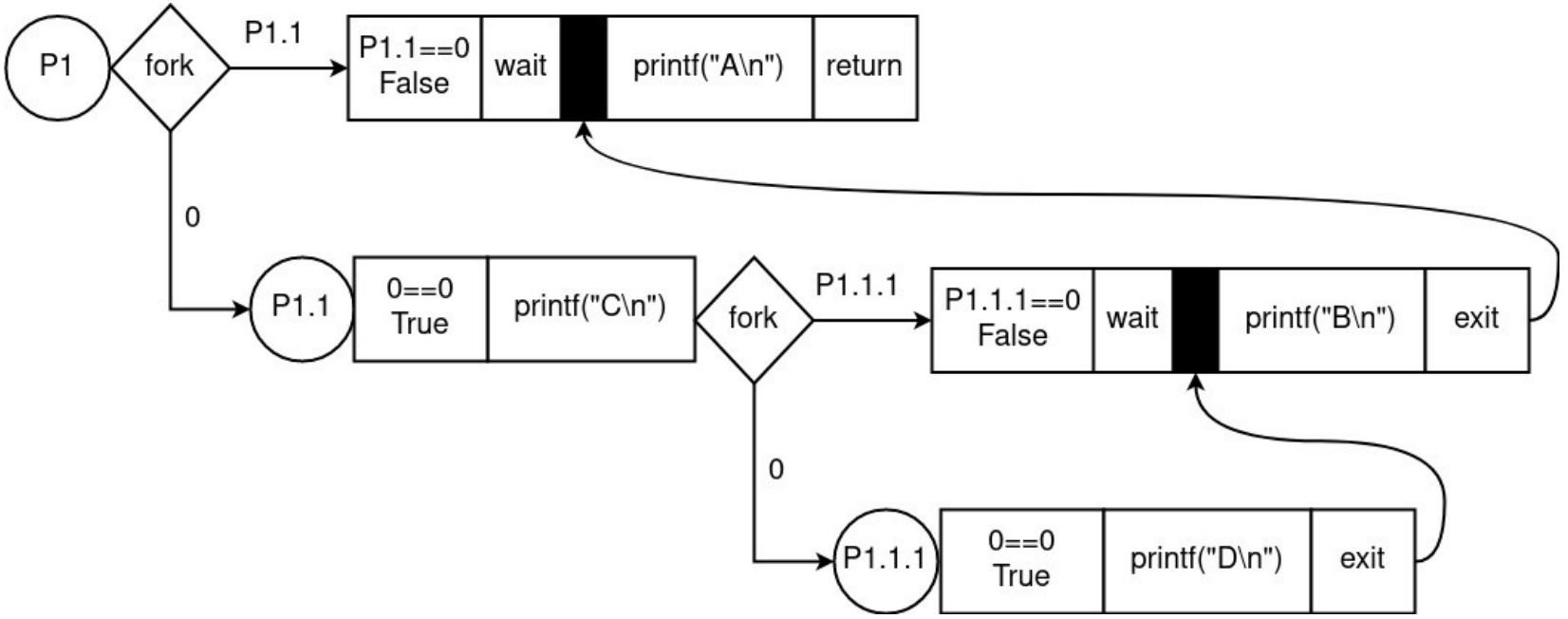
5. (exame de 16/09/2011) Considere o seguinte fragmento de código em C:

```
1:   if (fork() == 0) {
2:       printf("C");
3:       if (fork() == 0) {
4:           printf("D");
5:           exit(0);
6:       }
7:       wait(NULL);
8:       printf("B");
9:       exit(0);
10:  }
11:  wait(NULL);
12:  printf("A");
```

a) Represente a árvore de processos gerada pelo código anterior; b) Apresente todas as hipóteses possíveis de *output* (texto produzido por `printf`).

```
1:   if (fork() == 0) {
2:       printf("C\n");
3:       if (fork() == 0) {
4:           printf("D\n");
5:           exit(0);
6:       }
7:       wait(NULL);
8:       printf("B\n");
9:       exit(0);
10:  }
11:  wait(NULL);
12:  printf("A\n");
```

Solução



Terminação de Processos

- O processo pai pode esperar pela terminação de um processo filho usando a primitiva wait(). Obtém-se, assim, informação do status de terminação e o PID do processo que terminou
- **pid = wait(&status);**
- Se nenhum processo pai está à espera (porque não invocou a primitiva wait()), o processo diz-se **zombie**
- Se um processo pai termina sem ter invocado wait, o processo é um **orfão**
- Exemplos: programas 1.11 (processo zombie) e 1.12 (adoção de um processo orfão)

Exercícios

6. Modifique o programa 1.9 de forma a garantir que é o processo filho que aguarda pela terminação do processo pai (original). Sugestão: use `getppid`.

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

int main() {
 pid_t pid=-1; int status=-1, ret=-1;

 ret=fork();
 if (ret == 0) {
   printf("CHILD %d before getchar\n", getpid());
   ret=getchar(); // what would happen without getchar ?
   printf("CHILD %d after getchar\n", getpid());
   exit(ret);
}

 printf("PARENT before wait\n");
 pid=wait(&status); // PARENT blocks until the CHILD ends
 ret=WEXITSTATUS(status);
 printf("PARENT after wait: CHILD %d ended with return code %d\n", pid, ret);
 return(0);
}
```

Programa 1.9: Exemplo de utilização da primitiva `wait` e da macro `WEXITSTATUS`.

7. Escreva um programa no qual existe um array `A` de `N` inteiros, definido no processo pai, cabendo a `N` filhos analisar o array da seguinte forma: cada filho analisa uma célula específica do array e informa se o seu valor é par ou ímpar. Resolva cada uma das seguintes variantes: a) o valor de `N` é definido como uma constante, e os valores do array são definidos na sua declaração; b) o valor de `N` é solicitado ao utilizador, e os valores do array serão números aleatórios; c) variante de b) em que o pai deve apresentar o total de ímpares detectados.

8. (*) (exame de 2016-11-09) Considere o seguinte esqueleto de código:

```
#include <stdio.h>
// block 1 (additional headers files)

void generate_fibonacci(int N, long int *F)
{
    int n;
    F[0]=0; F[1]=1;
    for (n=2; n<N; n++) F[n] = F[n-1] + F[n-2];
}

void show_fibonacci(int N, long int *F)
{
    int n;
    for (n=0; n<N; n++) printf("%ld ", F[n]);
    printf("\n");
}
```

```
int main()
{
    int N, i; long int *F;

    scanf("%d", &N);
    // block 2

    for (i=0; i<N; i++) {
        // block 3
    }

    // block 4

    return(0);
}
```

Defina o conteúdo dos blocos em falta de forma a apresentar um programa completo, compatível com a seguinte descrição:

- pretende-se descobrir quais os elementos ímpares que existem na sequência dos `N` primeiros números de Fibonacci; tal sequência tem função geradora F(n) = F(n-1) + F(n-2) com F(0) = 0 e F(1) = 1; no código fornecido existem funções capazes de gerar a sequência dos `N` primeiros números de Fibonacci (`gera_fibonacci`) e de a apresentar no ecrã (`mostra_fibonacci`)
- a descoberta dos ímpares deve ser feita concorrentemente, por um número de processos igual ao número de elementos da sequência de Fibonacci
- por exemplo, para `N=10` o programa deve descobrir quais os ímpares existem na sequência dos 10 primeiros números de Fibonacci, devendo essa descoberta ser feita por 10 processos filhos
- assim, cada filho verificará se um certo elemento da sequência de Fibonacci é ou não ímpar, devendo apresentá-lo no ecrã em caso afirmativo
- o processo pai deve terminar imediatamente após a criação dos filhos

9. (*) Resolva de novo o exercício anterior, de forma a que o processo pai, antes de terminar, apresente o total de ímpares descobertos pelos filhos.

10. (*) (exame de 2017-02-15) Considere o seguinte esqueleto de código:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
// block 1 (additional headers files)

int main() {
    int i, num_pids_even=0, num_pids_odd=0;
    pid_t pid, pids_even[10], pids_odd[10];

// block 2 (additional variables)

// block 3 (code)

for (i=0; i<10; i++) {
    pid=fork();

    // block 4 (code)

}

// block 5 (code)
```

Defina os blocos em falta de forma a apresentar um programa completo, compatível com a seguinte descrição: um processo pai cria 10 processos filhos, guardando os PIDs destes em dois arrays diferentes, um para PIDs pares e outro para PIDs ímpares; antes de terminar, um filho deve mostrar o seu PID; depois de criar todos os filhos, o pai deve aguardar que os filhos com PID par terminem; de seguida, o pai deve aguardar que os filhos com PID ímpar façam a mesma coisa; só depois o pai pode terminar.

11. (*) Resolva de novo o exercício anterior, recorrendo a um só array de PIDs.

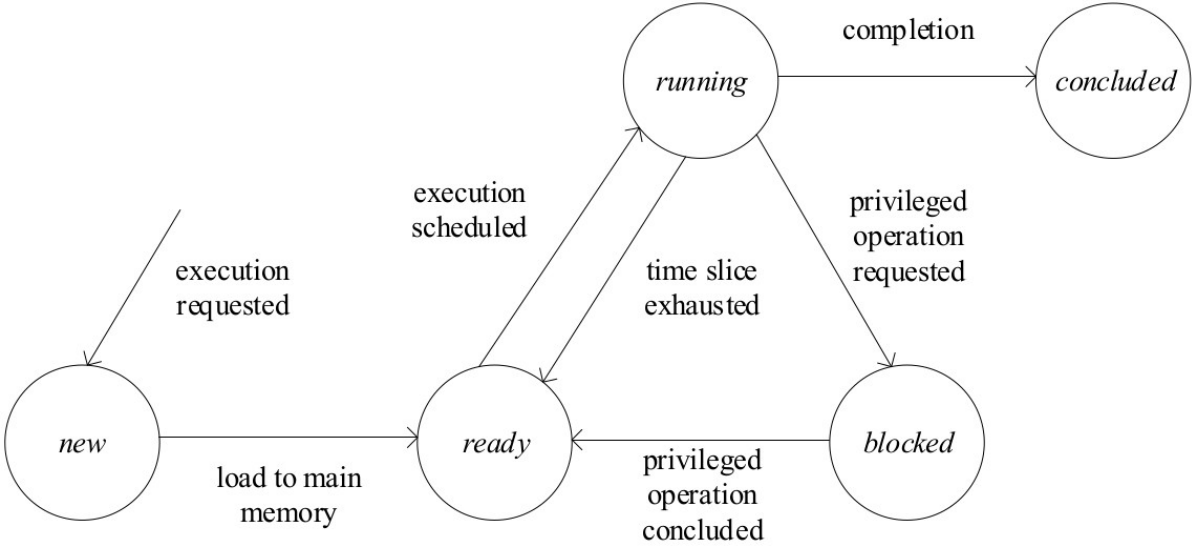
Sistemas Operativos

Processos

- Organização da Memória de um Processo
- Relações entre Processos
- Identificação de Processos (getpid e getppid)
- Criação e Terminação de Processos (fork e exit)

Programa vs. Processo

- Um programa é uma sequência de instruções
- **Um processo é um contexto de execução do programa (“ é um programa em execução)**
- Ao longo da sua vida, um processo percorre vários estados



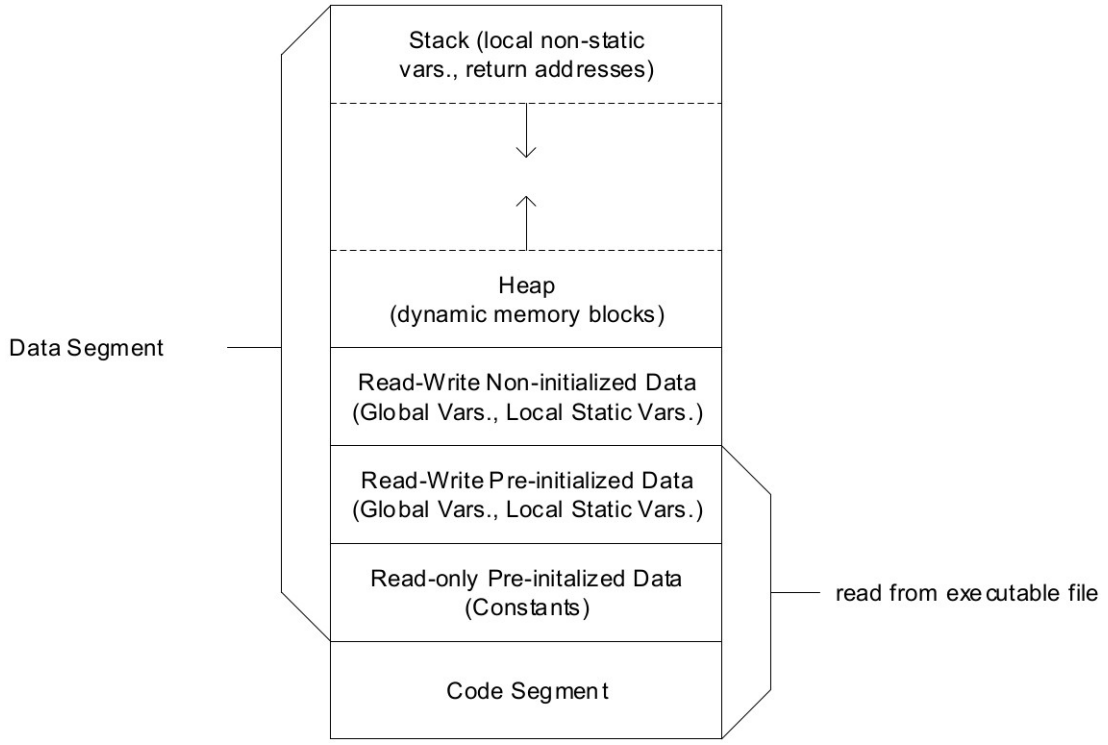
Organização da Memória de um Processo

- A memória (virtual) atribuída a um processo organiza-se em duas zonas
  - Zona de **Nível Utilizador**: a única zona acessível em Modo Utilizador
  - Zona de **Nível Supervisor**: apenas acessível em Modo Supervisor

Organização da Memória de um Processo

- A Zona de **Nível Supervisor** (ou **Bloco de Controlo** de um Processo) só é acessível em Modo de execução Supervisor/Privilegiado
- Nesta zona está a informação de gestão do processo, durante o seu tempo de vida:
  - o estado do processo
  - o conteúdo dos registos da CPU (usado na comutação de processos)
  - informação de escalonamento (prioridade, fatia de tempo de CPU, etc.)
  - informação de gestão de memória (localização e dimensão das suas zonas)
  - informação de dos recursos consumidos
  - tabelas de descritores de ficheiros, etc.

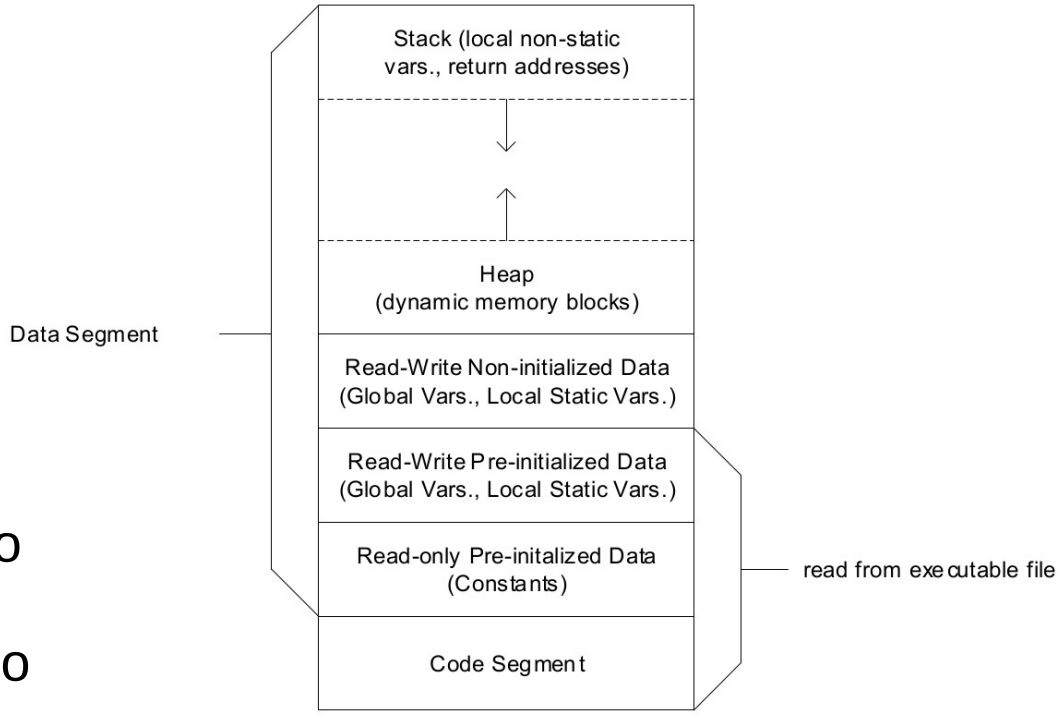
Organização da Memória de um Processo



A Zona de Nível Utilizador consiste no **Segmento de Código** e no **Segmento de Dados**

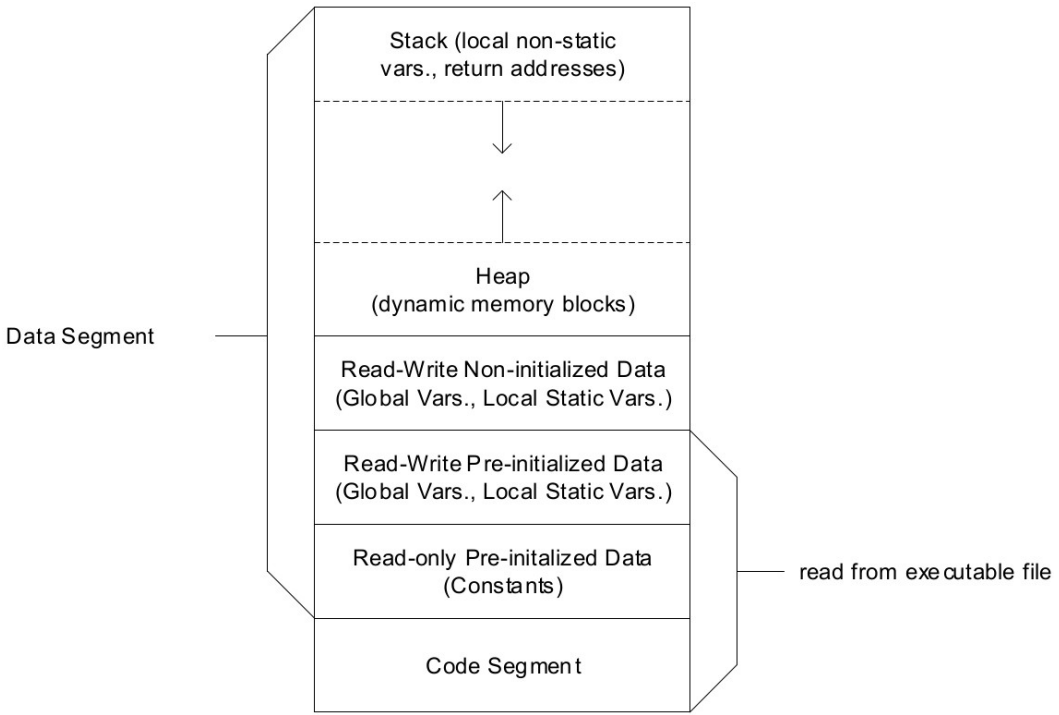
Organização da Memória de um Processo

- **Segmento de Código**
  - sequência de instruções do programa a executar
  - carregada a partir do ficheiro executável do programa
  - é apenas de leitura (o programa não se pode auto-modificar)
  - assim possibilitando a partilha do mesmo segmento de código por vários processos que executam o mesmo programa



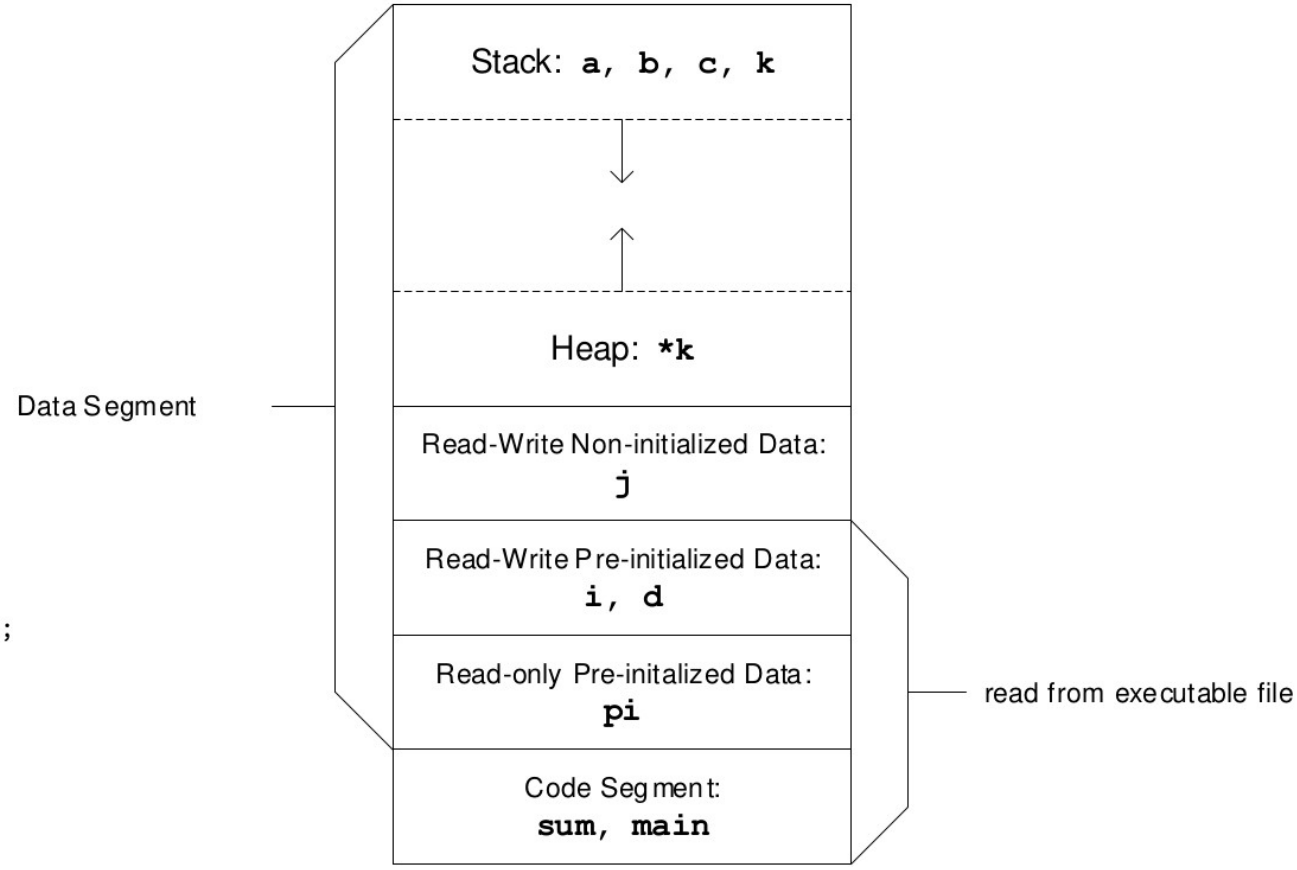
Organização da Memória de um Processo

- **Segmento de Dados**
  - **Stack/Pilha**: guarda o endereço de retorno (valor que o Program Counter deve assumir após a execução da função), o conteúdo dos registos da CPU imediatamente antes da invocação, os parâmetros, as variáveis locais e eventual valor de retorno
  - **Heap**: para os pedidos de memória dinâmica
  - **Dados de Leitura e Escrita, Não-Inicializados**: variáveis globais e variáveis estáticas, não-inicializadas explicitamente pelo programa
  - **Dados de Leitura e Escrita, Inicializados**: variáveis globais e variáveis estáticas, inicializadas
  - **Dados apenas de Leitura, Inicializados**: constantes



Distribuição na memória dos objetos de um programa

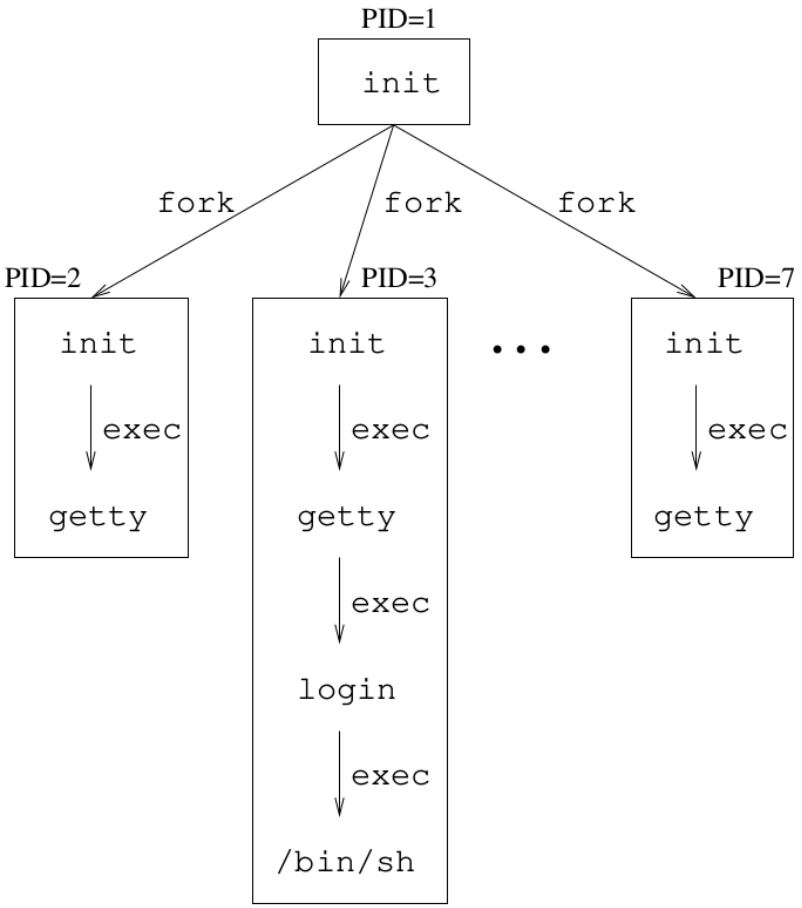
```
#define pi 3.1415927

float i=pi, j;

float sum(float a, float b)
{
  float c=0; static float d=0;
  d=a+b+c+d; return(d);
}

int main()
{
  float *k;

  j=sum(i,pi);
  k=(float*)malloc(sizeof(float)); *k=j; free(k);
  return(0);
}
```



Relações entre Processos

- Os processos são organizados numa hierarquia em árvore:
  - Pai
  - Filho
- O processo **init** é o primeiro processo em execução (geralmente criado pelo kernel, após o arranque do sistema operativo)
- A partir deste processo, são criados processos clonados que
  - herdam o mesmo programa do pai, ou
  - substituem o programa herdado

Árvore de Processos



O '**getty**' é responsável por exibir um prompt de login no terminal, onde o utilizador insere o seu nome de utilizador. Após o utilizador pressionar Enter, o 'getty' passa o controle para o processo de '**login**', que é responsável por autenticar o utilizador e solicitar a palavra-passe. Após um login bem-sucedido, o interpretador de comandos '**sh**' é lançado para permitir a interação com o sistema

Bibliotecas <unistd.h> e <sys/types.h>

- **<unistd.h>**: Biblioteca para interação com o SO
- Inclui funções como **fork()**, **getpid()**, **getppid()**, entre outras, para controle de processos e manipulação de descritores de ficheiros
- **<sys/types.h>**: Biblioteca relacionada aos tipos de dados usados em chamadas do sistema e manipulação de ficheiros
- Introduz tipos como o **pid_t** (tipo de dados correspondente a int) que especifica identificadores de processo

Identificação de Processos

- Cada processo é identificado por um número inteiro único (Process ID ou PID)
- Para identificação de processos usamos:

  #include <unistd.h>

  pid_t **getpid**(void); pid_t **getppid**(void);
- **getpid** retorna o PID do processo invocador (o PID próprio)
- **getppid** retorna PID do processo pai do processo invocador
- Exemplo: Programa 1.2 (p. 10)

Programa 1.2 (p. 10)

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

int main() {
 printf("my PID: %d\n", getpid());
 printf("my parent's PID: %d\n", getppid());
 return(0);
}
```

Criação de Processos

- A única forma de criar um novo processo (com excepção do init), designado de filho, é através da criação de uma cópia de um outro, designado de pai
- Os processos filho podem executar o programa herdado do pai ou, sendo conveniente, substituir esse programa por outro(s)
- O filho herda uma cópia do Segmento de Dados do pai (ou seja, **as variáveis não são partilhadas**)
- Como o Segmento de Código é apenas de leitura, o pai e o filho partilham-na, sem que seja feita uma cópia
- Um processo filho também herda do seu pai certos atributos de Nível Supervisor: diretoria de trabalho, cópia dos descritores dos ficheiros abertos pelo pai antes da criação do filho, etc.

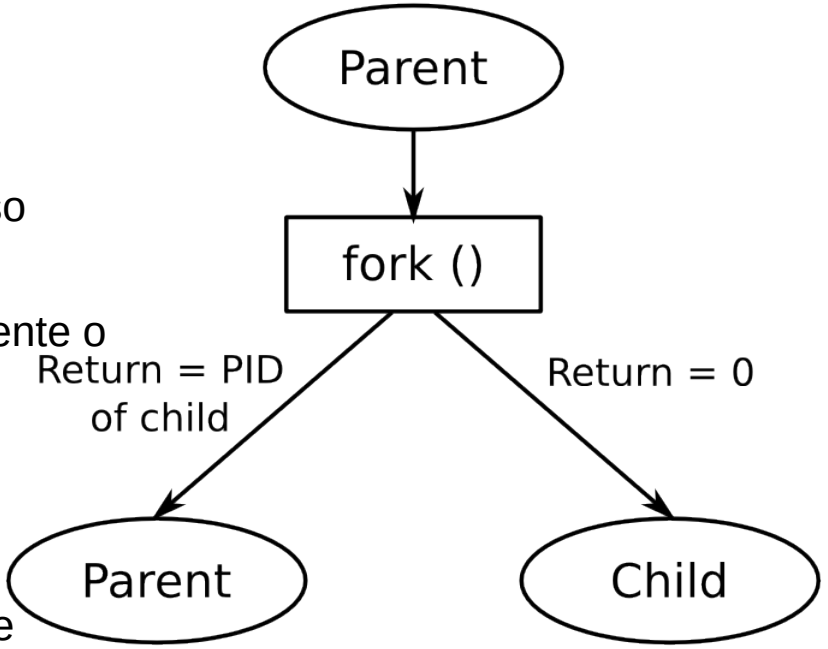
Primitiva fork

- #include <unistd.h>

  pid_t fork(void);
- man 2 fork
- É usada para criar uma cópia (o filho) do processo invocador (o pai)
- Após uma invocação bem sucedida a execução do programa prossegue na instrução a seguir a fork, mas em duplicado, de forma concorrente, pois passam a existir duas instâncias do mesmo programa, mas alojadas em processos diferentes (pai e filho)

Retorno na Chamada fork()

- Quando a chamada fork() é realizada, ambos o processo pai e o processo filho recebem um valor de retorno
- O processo pai:
  - Recebe o PID (Identificador de Processo) do processo filho recém-criado
  - O PID é um número inteiro que identifica exclusivamente o processo filho
- No Processo Filho:
  - Recebe o valor de retorno 0 (zero)
  - Isso permite ao processo filho identificar que ele é, de fato, o processo filho
- Exemplo: Programa 1.3 (p. 12)



Programa 1.3 (p. 12)

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

int main()
{
 pid_t ret=-1;

 fork(); // (1)
 // ret=fork();  // (2)
 printf("ret: %d\n", ret);
 return(0);
}
```

Primitiva exit

- #include <stdlib.h>

  void exit(int status);
- man 3 exit
- Quando um processo termina, deve invocar a primitiva exit com um **parâmetro inteiro**
- O código de saída reflete as condições em que o processo terminou
- O código de saída fará parte do estado de terminação podendo ser inspecionado pelo processo pai
- Convenção: status = 0 (sucesso), status = 1 … 255 (insucesso)
- O valor inteiro poderá ser o da variável global **errno** (#include <errno.h>)
- Exemplo: Programa 1.5 (p. 15)

Programa 1.5 (p. 15)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
 pid_t ret;

 ret=fork();
 if (ret<0) { perror("fork"); exit(errno); }
 if (ret==0) {
     printf("I am the CHILD %d of the PARENT %d\n", getpid(), getppid());
     exit(0);
 }
 printf("I am the PARENT %d of the CHILD %d\n", getpid(), ret);
 return(0);
}
```

Exercício 1 da Secção 1.12 (p. 27)

- Desenvolva e execute os seguintes programas, monitorizando a utilização da CPU no **htop** (instale com **sudo apt install -y htop** se ainda não estiver disponível):
  - a) programa onde um processo pai e um processo filho executem ambos um ciclo infinito, apresentando-se, por cada iteração do ciclo, o PID do processo em execução, e o valor de um contador inteiro que é incrementado de uma unidade por cada iteração
  - b) variante de a) em que, por cada iteração, também se invoca sleep(1), de forma a visualizar, pausadamente, o output dos processos
  - c) variante de a) em que o corpo do ciclo é vazio (ou seja, os processos são CPU-bound)
  - d) variante de c), com apenas um processo (neste cenário é mais fácil observar a migração do processo entre diferentes CPUs/núcleos; se necessário, gere carga adicional para favorecer a migração: **`sudo apt install -y stress-ng fio iperf3 sysstat; taskset -c $((RANDOM % 2)) stress-ng --cpu 1 --timeout 15s`**)