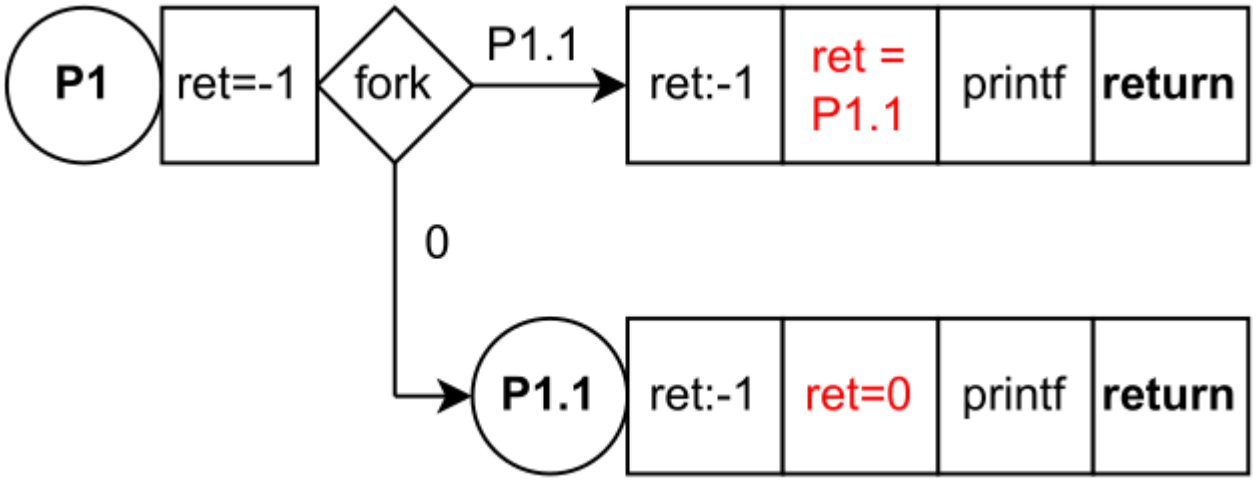
Criação e Terminação de Processos (cont.)

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

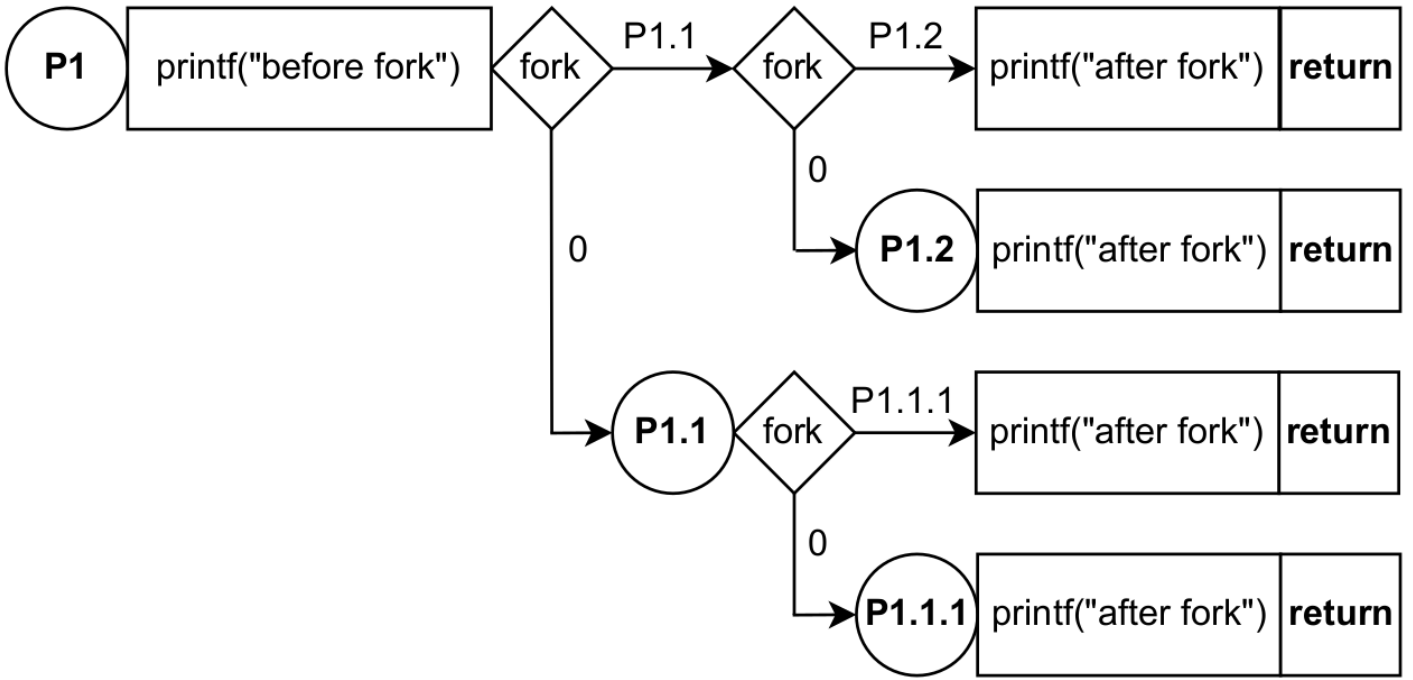
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

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);
}
```

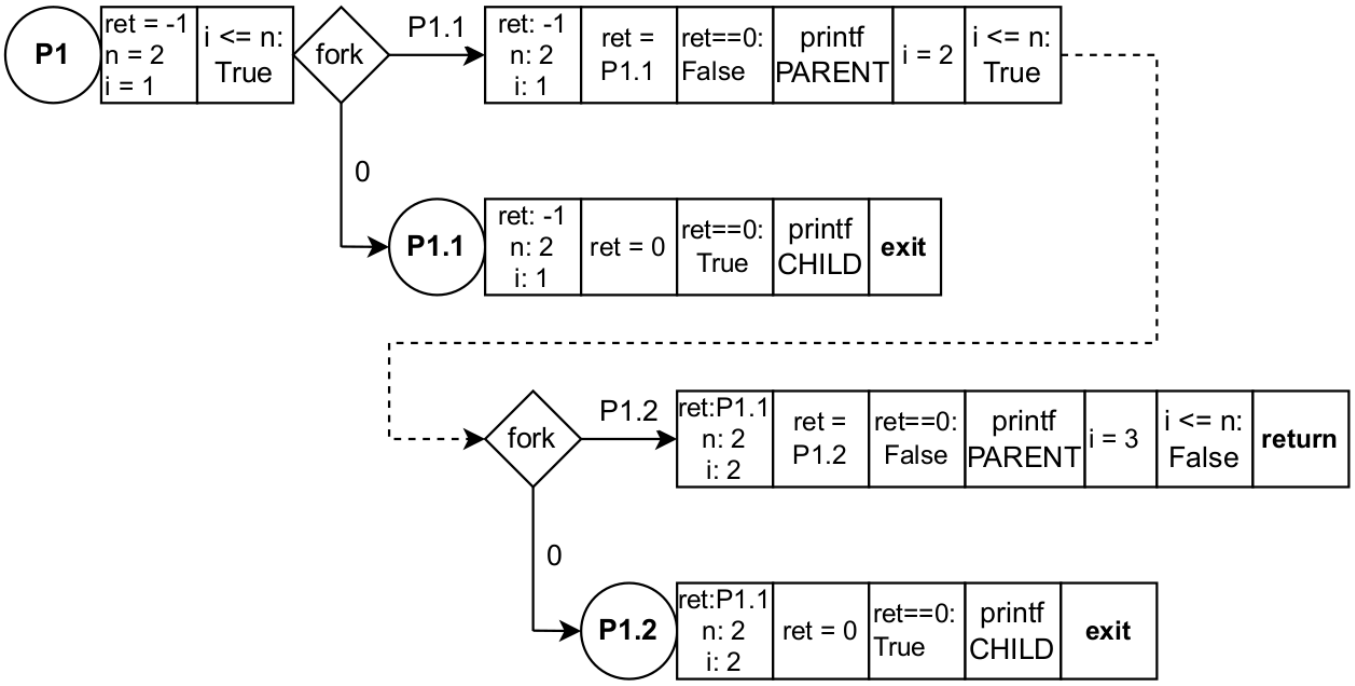


Árvore de processos para o Programa 1.7, com n=2

```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);
}
```

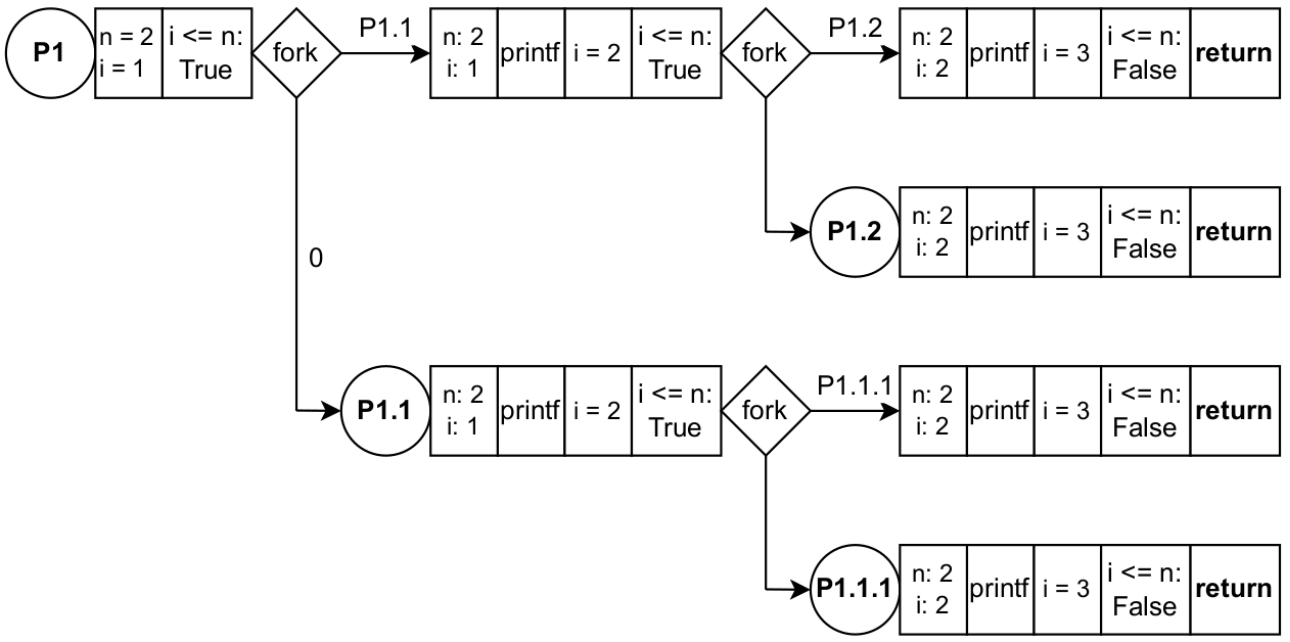


Árvore de processos para o Programa 1.8, com n=2

```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);
}
```

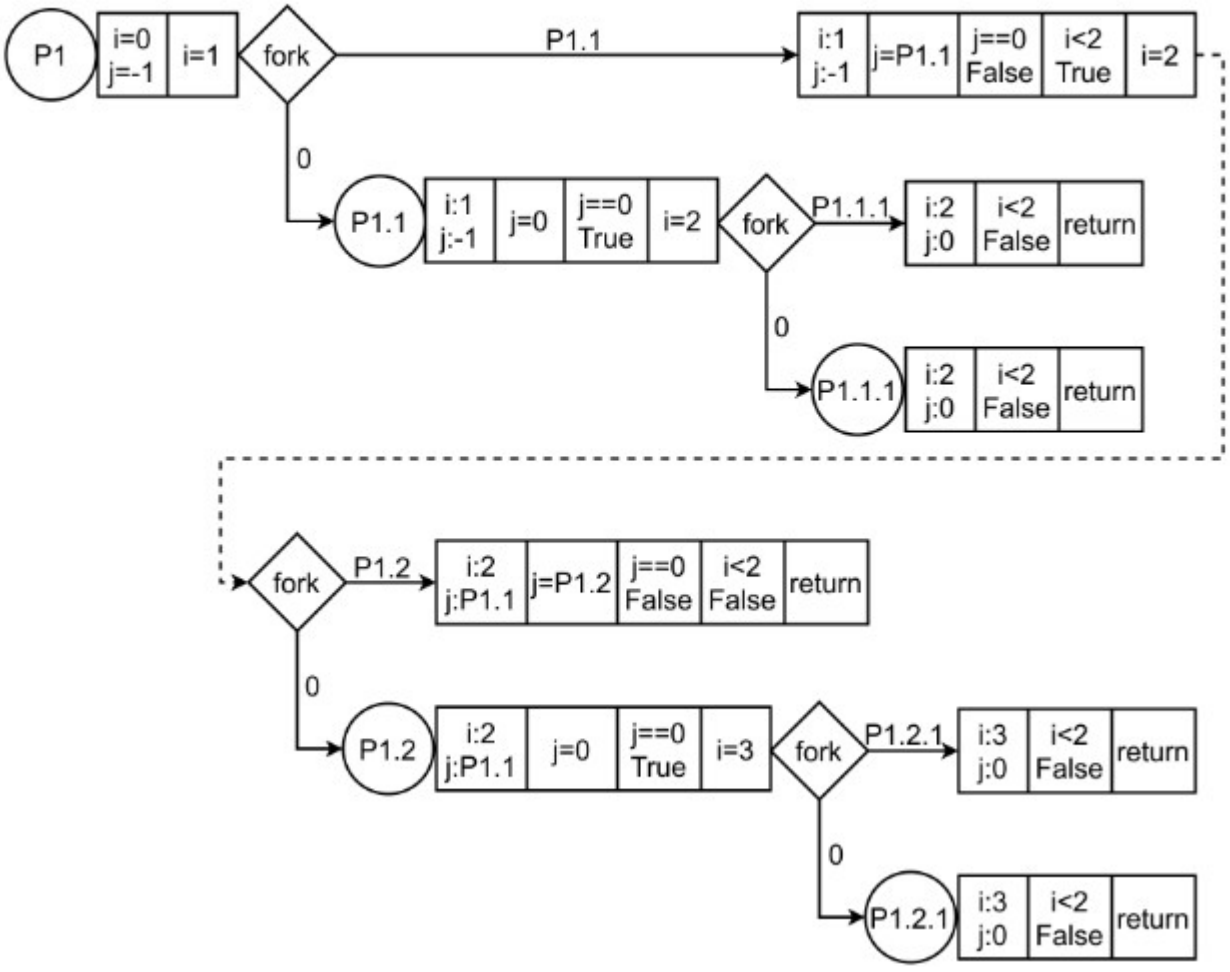


Exercício 1.2

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

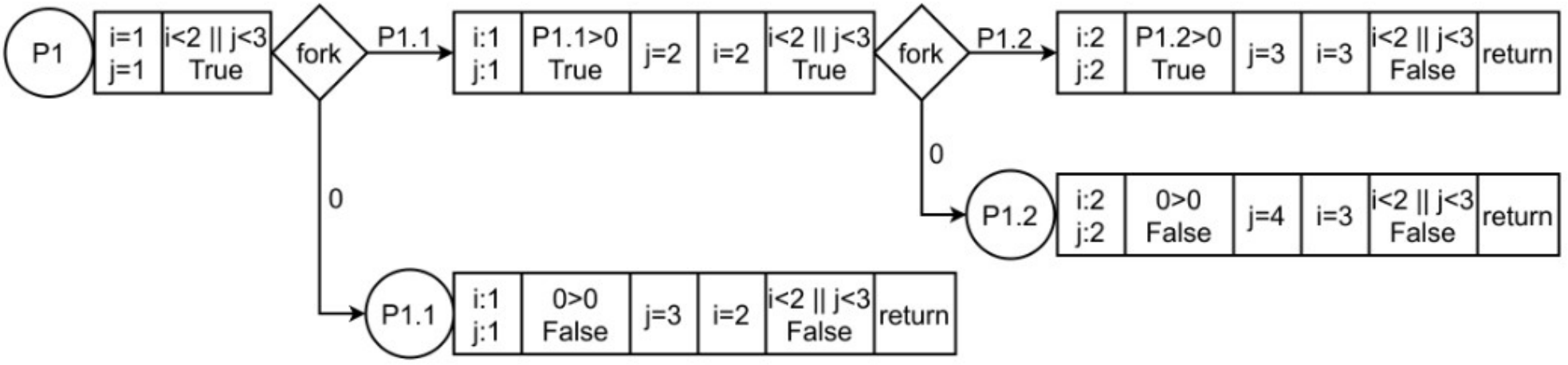
Exercício 1.3

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

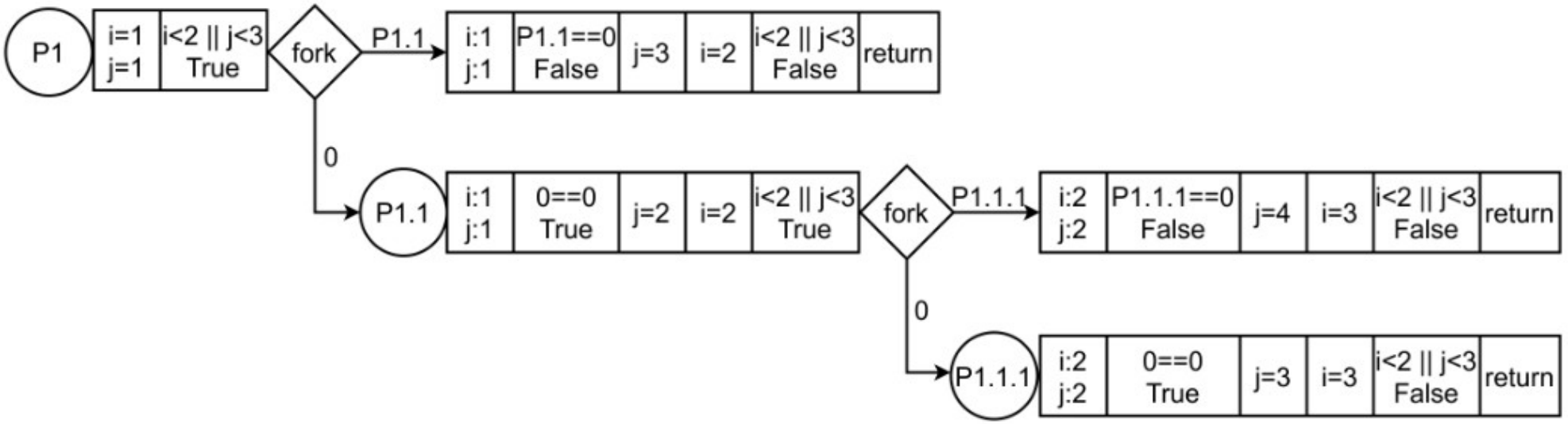
```
int i, j;
for (i=1, j=1; i<2 || j<3; i++)
    if( fork()>0 ) j+=1; else j+=2;
```

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

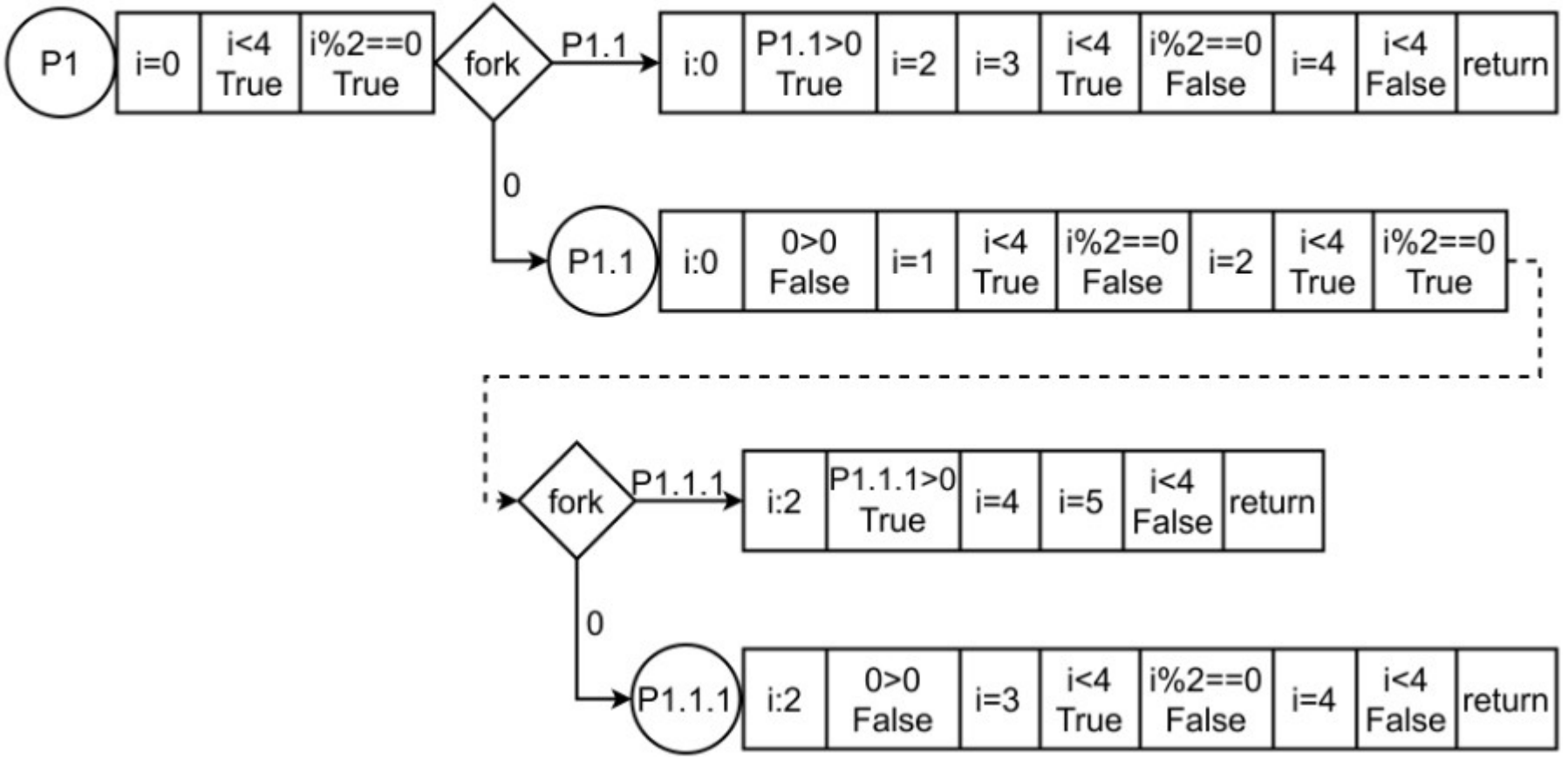

P1
i=1
j=1
i<2 || j<3
True
fork
P1.1
i:1
j:1
P1.1==0
False
j=3
i=2
i<2 || j<3
False
return
0
P1.1
i:1
j:1
0==0
True
j=2
i=2
i<2 || j<3
True
fork
P1.1.1
i:2
j:2
P1.1.1==0
False
j=4
i=3
i<2 || j<3
False
return
0
P1.1.1
i:2
j:2
0==0
True
j=3
i=3
i<2 || j<3
False
return

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);
}
```

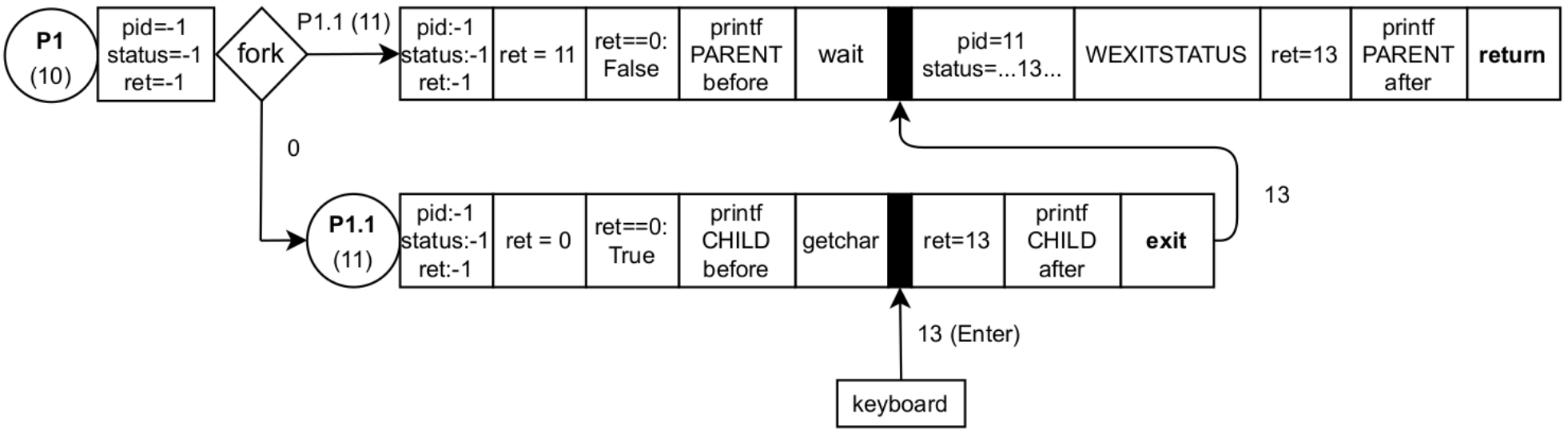
wait

```
#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);
}
```

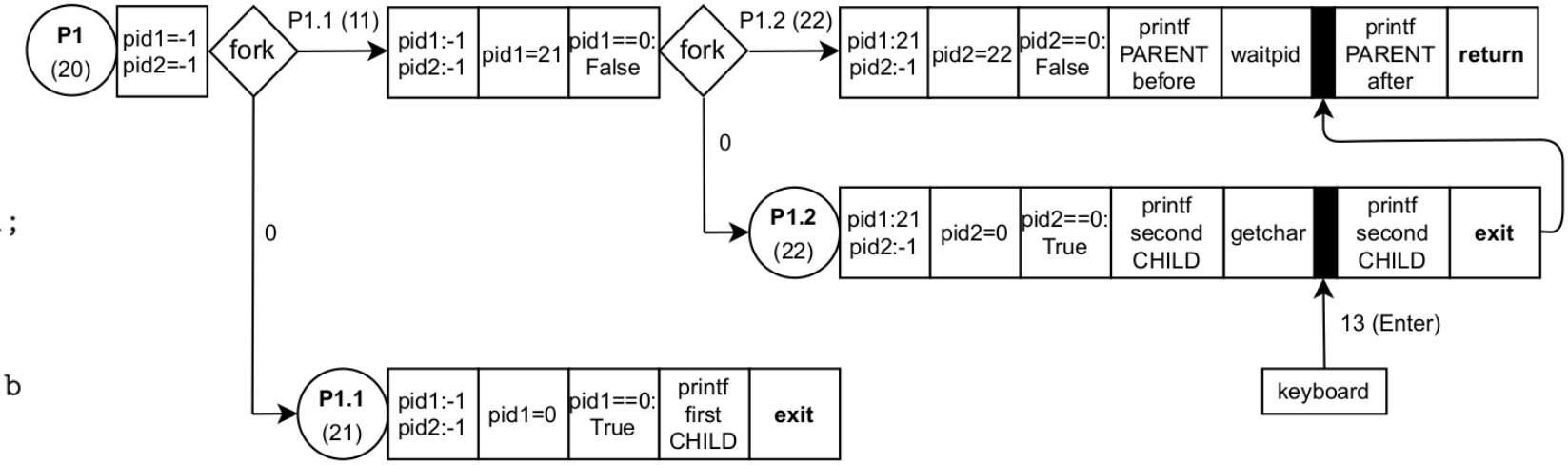
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

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);
}
```



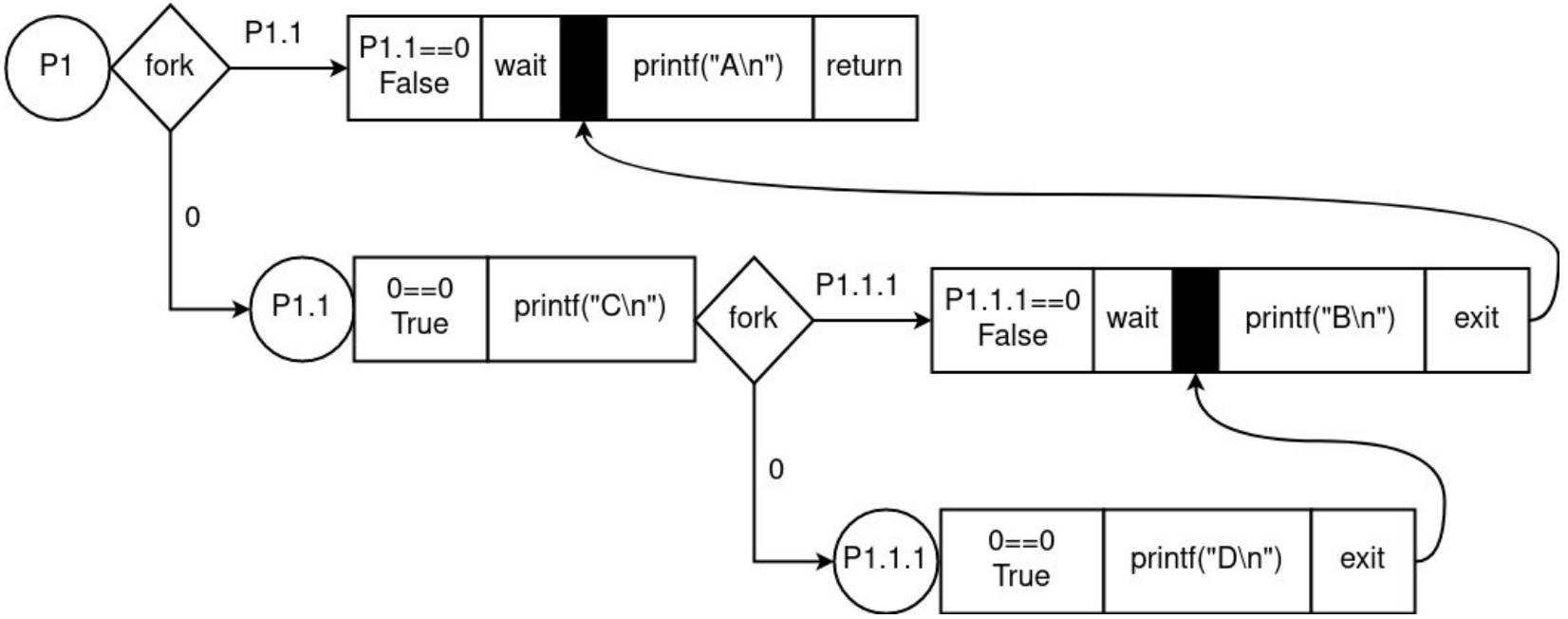
5. (exame de 16/09/2011) Considere o seguinte fragmento de código em C:

```
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.