**1. Qual arquitetura de sistemas operacionais que tem como principal distinção implementar diversas funcionalidades do SO como serviços em espaço de usuário em vez de mantê-las dentro do núcleo?**

Micro Kernel, implementa funcionalidades do sistema separadamente do núcleo.

Shell: um programa/interface que funciona como intermediário entre SO e usuário. Interpretador de comandos.

Kernel: núcleo do sistema operacional, gerenciamento das funcionalidades do SO.

O Kernel, utiliza navegador web, leitor de e-mail e toca música na interface do usuário.

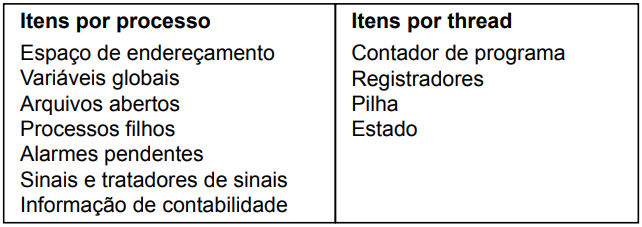
**2. Para cada recurso abaixo, indique se ele é compartilhado por todas as threads de um processo ou exclusivo de cada thread:**

**(a) registradores do processador;** exclusivo de cada thread

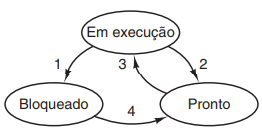
**(b) espaço do endereçamento;** compartilhado por todas as threads de um processo

**(c) variáveis locais;** exclusivo de cada thread

**(d) arquivos abertos.** compartilhado por todas as threads de um processo



**3. A figura abaixo mostra o diagrama de transição de estado de um processo:**



1. O processo bloqueia aguardando uma entrada

2. O escalonador seleciona outro processo

3. O escalonador seleciona esse processo

4. A entrada torna-se disponível

Em execução: Processo está usando a CPU

Pronto: Temporariamente parado enquanto outro processo executa a fila de prontos (aptos)

Bloqueado: Esperando por um evento externo.

- Processos entram no sistema na fila de prontos

- Transições dependem de interrupções para sinalizar condições término de operações de E/S, passagem do tempo, SINALIZAM CONDIÇÕES.

**Relacione as transições de estado 1-4 que podem ocorrer com um processo P1 em um sistema monoprocessado quando acontecem os seguintes eventos:**

**A. P1 executa up (S), onde S é um semáforo sobre o qual um processo P2 (mais prioritário que P1) havia executado down (S);**

2

**B. O processo P3 executa pthread\_mutex\_unlock(&m), onde m é um mutex;**

4

**C. P1 executa pthread\_barrier\_wait(&b), onde b é uma barreira;**

1

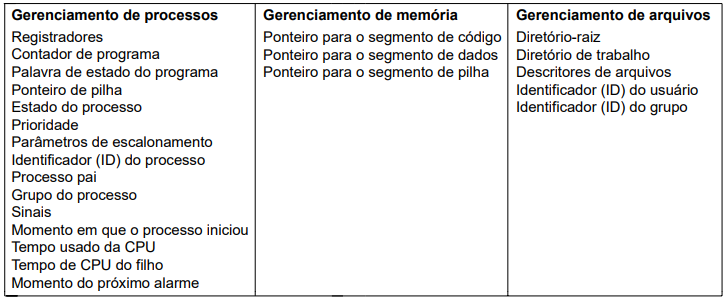
**D. O processo P4 solicita uma leitura de um bloco do disco.**

3

**ATENÇÃO:**

**- Considere os eventos isoladamente, ou seja, A, B, C e D não tem qualquer relação entre si.**

**- O fato de uma transição ser possível não significa que ela necessariamente irá acontecer.**



**4. Considere o código abaixo:**

**1 int main (void)**

**2 {**

**3 pid\_t f1, f2, f3;** *//identifica os processos filho. Variáveis definidas do tipo pid. Valor inteiro usado para representar um número de processo.*

**4 f1=fork(); //***Cria um processo. Sistema faz cópia do processo e aloca espaço de memória para execução do novo processo*

**5 f2=fork();** *//cria*

**6 if (f2==0) *//processo filho***

**7 f3=fork();** *//cria outro processo*

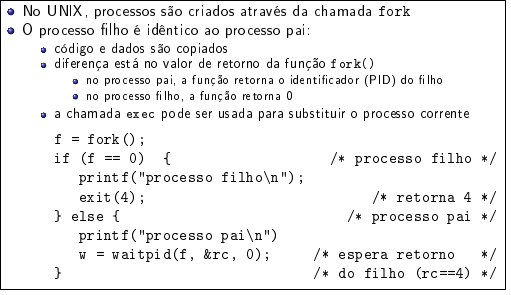
**8 if (f1==0 || f2==0) {**

**9 printf(“+”); //**

**10 }**

**11 return 0;**

**12 }**



**(a) Quantos processos (incluindo o pai de todos) são criados quando esse programa é executado?**

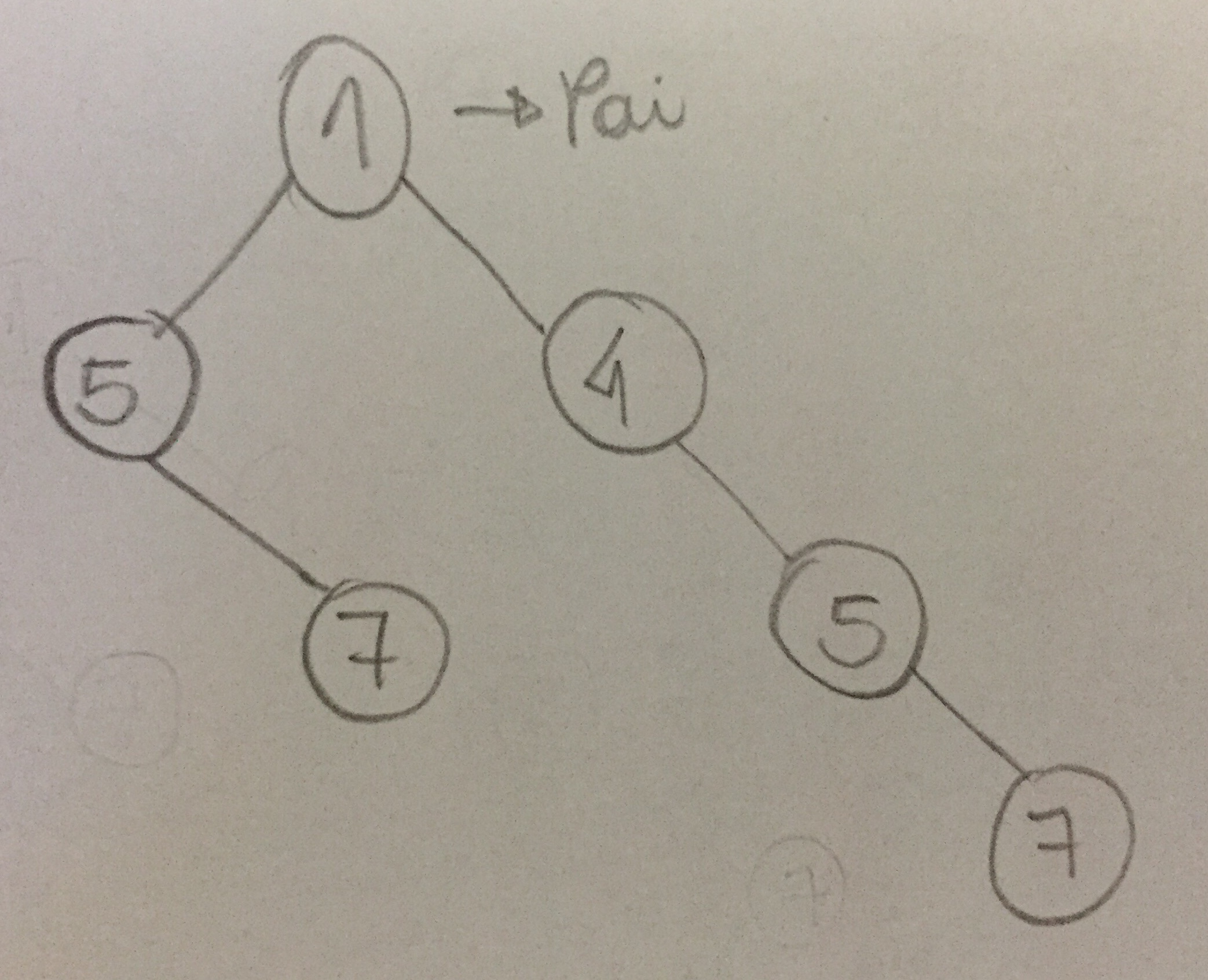
São criados 6 processos, incluindo o pai.

*Cada número são os números das linhas executadas.*

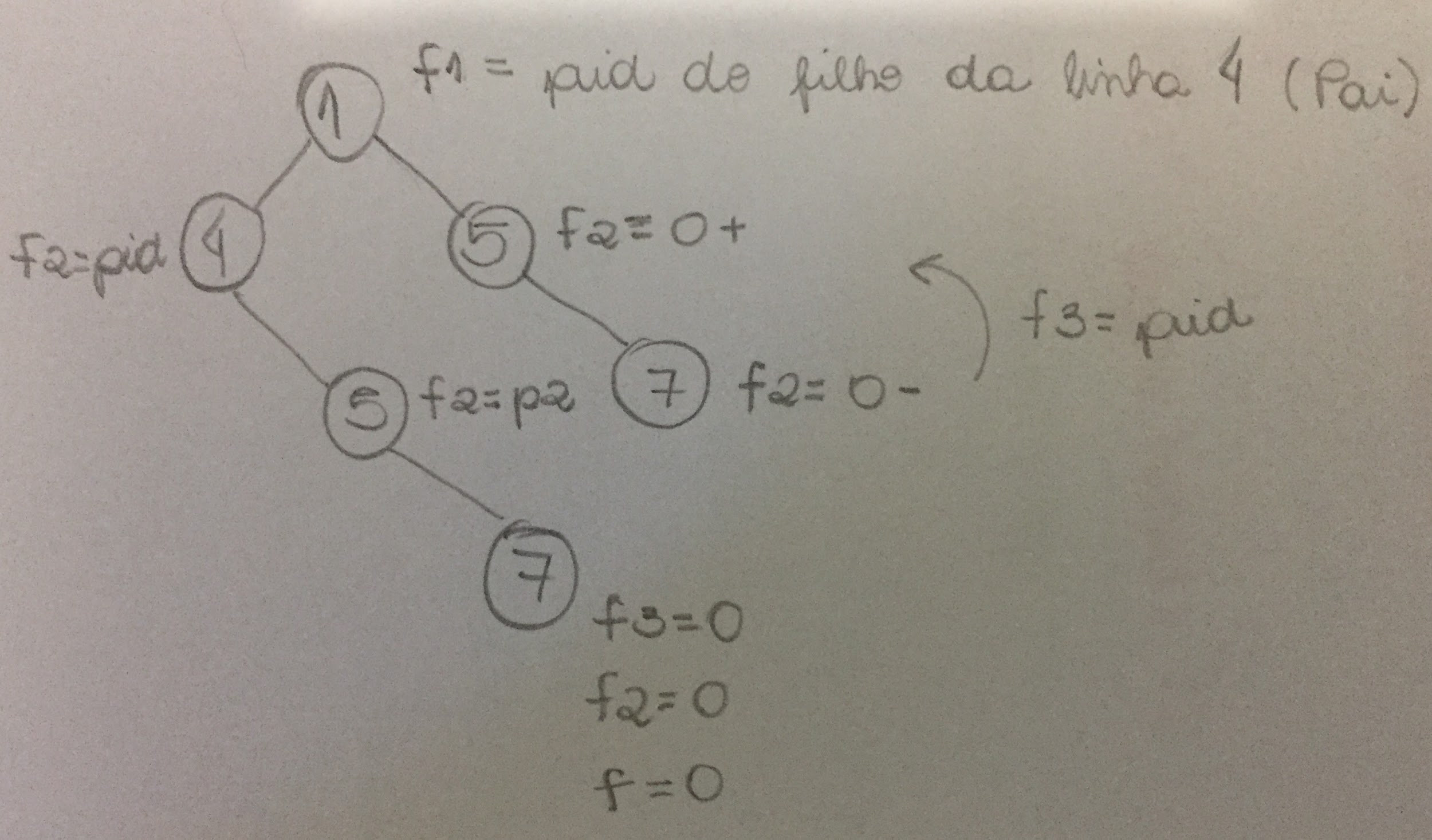
*Quando executar o próximo fork, terão dois processos e cada um deles vai gerar mais um processo.*

*Processo pai: função retorna o id do filho (PID)  
Processo Filho: função retorna 0*

*waitpid: espera retorno do filho.*



**(b) Desenhe a árvore de processos gerada na execução desse programa. Use o número da linha em que cada processo é criado para rotular os nós da árvore (use 1 para pai de todos)**



*Fork cria um novo processo no sistema e realiza uma tarefa que o usuário não pode. Duplica um processo.*

**(c) Quantas vezes o símbolo “+” é impresso quando o programa é executado?**

São impressos 5 vezes.

**5. Uma aplicação concorrente possui duas threads além da thread principal, que executa main(). A thread 1 executa as funções A() e B(), e a thread 2 executa as funções C() e D(). Essa aplicação possui os seguintes requisitos de concorrência:**

**R1. Nenhuma thread pode ficar bloqueada indefinidamente.**

**R2. As funções A() e C() modificam o mesmo estado compartilhado, sendo que a função C() só pode ser executada após o término da função A().**

**R3. Deve haver exclusão mútua entre as funções B() e D().**

*Exclusão mútua: 1. Dois processos nunca podem estar simultaneamente em uma região crítica. 2. Nenhum processo executando fora de sua região crítica pode bloquear outros processos. 3. Nenhum processo deve esperar eternamente para entrar em sua região crítica. SOLUÇÃO: Espera ocupada: O processo fica em loop até conseguir entrar na seção crítica.*

**O código relevante da aplicação está mostrado abaixo (os trechos indicados com … não são relevantes para o problema). Para cada um dos requisitos de concorrência acima, determine se o requisito é ou não satisfeito. Caso o requisito seja satisfeito, indique quais as construções do código que garantem isso. Do contrário, caso o requisito não seja satisfeito, diga por que isso ocorre, e mostre como o código poderia ser modificado para corrigir a situação.**

**/\*Definições e variáveis globais\*/**

**pthread\_mutex\_t m1= PTHREAD\_MUTEX\_INITIALIZER;** *// criação. É criado destravado*

**pthread\_mutex\_t m2= PTHREAD\_MUTEX\_INITIALIZER;**

**pthread\_mutex\_t m3= PTHREAD\_MUTEX\_INITIALIZER;**

**pthread\_cond\_t c1 = PTHREAD\_COND\_INITIALIZER;** *//Retorna 0 para sucesso e outro valor para erro*

**void Thread1 () {**

**…**

**pthread\_mutex\_lock(&m1);**

**A();**

**pthread\_cond\_signal(&c1, &m1);** *// acorda uma thread bloqueada por uma variável de condição. Invocada com mutex travado. Thread sinalizada só retoma execução depois que mutex for destravado por quem executou pthread\_cond\_signal();*

**pthread\_mutex\_unlock(&m1);** *// não é possível determinar qual das threads bloqueadas será escalonada*

**…**

**pthread\_mutex\_lock(&m2);**

**B();**

**pthread\_mutex\_unlock(&m2);**

**...**

**}**

**void Thread2 () {**

**…**

**pthread\_mutex\_lock(&m1);**

**pthread\_cond\_wait(&c1, &m1);** *//Bloqueia a thread até que cond seja sinalizada. Acesso a variável de condição cond deve estar protegido por mutex. Mutex é automaticamente destravado quando a thread bloqueia. Quando cond for sinalizado, a thread retoma a execução com um mutex travado para seu uso.*

**C();**

**pthread\_mutex\_unlock(&m1);** *// Destravar após uma região crítica. Verificar se cond foi satisfeita, pois o desbloqueio pode não ter sido causado por sinalização da variável.*

**…**

**pthread\_mutex\_lock(&m3);**

**D();**

**pthread\_mutex\_unlock(&m3);**

**...**

**}**

3 threads

thread 1 executa A() e B()

thread 2 executa C() e D()

R3 não é satisfeito, pois dois mutexes são utilizados, quando deveria ser apenas um bloqueando o acesso a região crítica após mutex lock.

Correção:

pthread\_mutex\_lock(&m2)

B();

pthread\_mutex\_unlock(&m2)

pthread\_mutex\_lock(&m2)

D();

pthread\_mutex\_unlock(&m2)

R2: Como na linha 4 da segunda thread existe uma cond\_wait, é garantido que a execução da C() na linha 5 só ocorre quando uma cond signal for enviada para o mutex, que ocorre na linha 5 da thread 1, justamente ao acabar A().

Porém caso a thread 1 executa o signal antes de a thread 2 executar cond\_wait, o signal será perdido e a thread 2 ficaria em deadlock. Para corrigir isso, basta usar 2 semáforos no lugar das condições.

**6. Considere o trecho abaixo em Assembly 8088, onde N é uma constante, i é um rótulo na seção de dados e f1 e f2 são subrotinas:**

*//Em vez de while é: desvio condicional, se determinada condição for verdadeira, desvia*

**MOV (i), 0** *//MOV Dest,Source (Move, copia)*

**MOV AX, 0**

**L1: CMP AX, N** *//CMP Op1,Op2 (Compara)*

**JAE L2** *//(Jump if Above or Equal) JAE Dest //Saltar se Acima ou Igual* **!CMP Op1, Op2 + JAE => desvia se Op1 maior ou igual ao Op2**

**PUSH AX** *//(Push onto stack: Empurre na pilha.) PUSH Source*

**CALL f1** *//(Call subroutine) CALL Proc*

**ADD SP, 2** *//ADD Dest,Source*

**ADD AX, 1**

**MOV (i), AX**

**JMP L1** *//Pula, JMP Dest*

**L2: CALL f2**

Identifique qual dos trechos em linguagem C abaixo melhor corresponde ao código Assembly acima e explique o que há de errado com os demais trechos. Você pode supor que os valores dos registradores de propósito geral (AX, BX, CX, DX) são preservados quando ocorre uma chamada de subrotina (isto é, a própria subrotina restaura os valores originais dos registradores antes de retornar para o chamador).

**Trecho 1:**

i=0;

do { *//diferença do só while é que os comandos executados pelo menos uma única vez.*

f1();

i++;

} while (i<=N);

f2();

**Trecho 2:**

i=0;

while (i<N) { //*Quando não sei quantas vezes um determinado bloco de instruções precisa ser repetido. Continua, até que a condição seja verdadeira: booleano*

f1(i);

i++;

};

f2();

**Trecho 3:**

i=0;

if (i<N) {

f1();

i++;

}

f2();

Resposta: O melhor trecho é o 2, pois no código Assembly, ao entrar em L1, a primeira instrução realizada é uma comparação, que é a checagem da execução do método while, a única diferença é que na linha 4, em assembly é executado JAE-L2, que irá para o segundo label (L2) se AX >= N.

O trecho 1 possui um do while(), ou seja, executaria as instruções do seu bloco pelo menos 1 vez, pois a condição é testada no final de cada iteração.

O trecho 2, como é um if, não poderia ter a linha 10, que retornaria para o início do bloco.