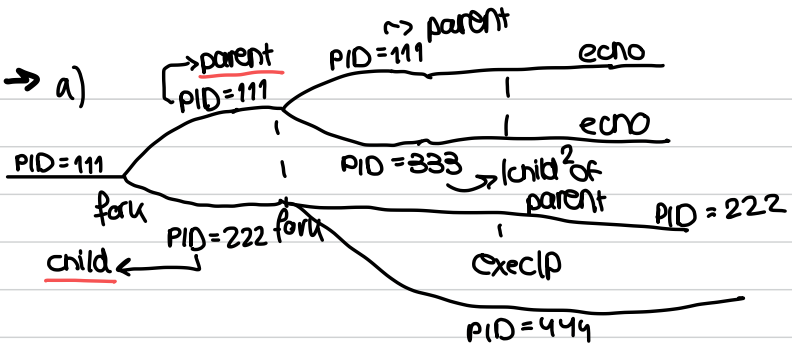
Exame Normal so

23

I → a)



4 processos

Dps de fork → o valor de fork será PID → pai

→ o valor do fork será 0 se filho

error → -1 → filho não criado

b)

```
int main (int argc, char * argv[])
{
    printf("1\n");          111
    fork();                 111, 222
    if (fork()!=0){         111, 222, 333, 4444
        printf("2\n")       111, 222
        execlp("echo","echo","3",NULL); 111; 222
        printf("4\n");      → nunca empnme quatb
    }
    printf("5\n")           333, 444
    return 0                333, 444
```

c)

output 1	output 2
1	1
2	2
2	3
3	5
3	2
5	3
5	5

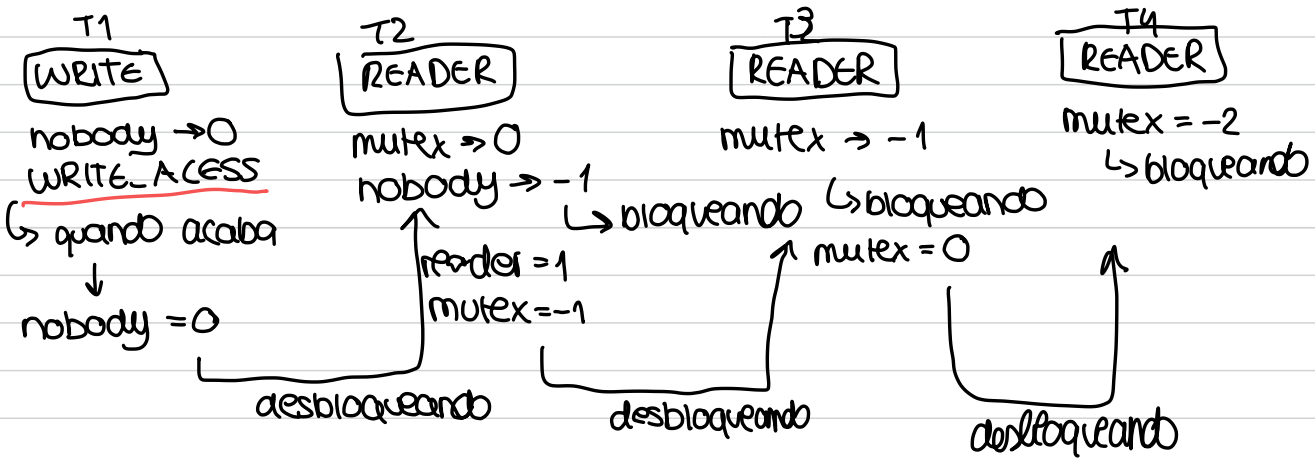
d)

um processo tem 5 estados. O estado new indica que o processo está a ser criado, running indica que as instruções do processo estão executando, waiting, que o processo está à espera de uma ocorrência de algum evento e terminated que termina execução.

II →

Inicio: mutex e nobody iniciados a 1

a)



b) O código anterior empelica o adiamento das escritoras, pois estas só conseguem escrever se forem as primeiras a executar ou se readers tiver a 0, o que pode não acontecer se a execução de threads leitoras for contínua.

```
readStart()
  turnshike.down()
  turnshile.up()
  :
  mutex.down
```

```
writeStart()
  turnshile.down()
  nobody.down()
(write)
writeEnd()
    turnshile.up()
    nobody.up()
```

c) Um deadlock acontece quando dois ou + processos estão à espera indefinidamente por um recurso que está na posse de outro dos processos em espera.
Quando acontecem 4 condições que se verificam:

↳ Exclusão mútua: cada recurso ou está livre ou foi atribuído a um e um só processo

↳ Espera com retenção: cada processo, ao requerer um novo recurso, mantém na sua posse os recursos anteriormente solicitados

↳ Não libertação: ninguém a não ser o próprio processo pode decidir da libertação de um recurso que lhe tenha sido atribuído

↳ Espera circular: cada processo requer recurso que está na posse do processo seguinte na cadeia.

Exemplo 1:

```
while (true)
  think()
  getForks()
    fork[left(f)].down()
    fork[right(f)].down()
  eat()
  putForks()
    fork[left(f)].up()
    fork[right(f)].up()
```

Exemplo 2:

```
import threading
import time

sem_verm = threading.Semaphore(1)
sem_azul = threading.Semaphore(1)


def alice():
  print("Alice quer brincar com brinquedo vermelho")
 sem_verm.acquire()
 print("Alice pegou brinquedo vermelho")
 time.sleep(1)  # Brincar com bring. vermelho
 sem_azul.acquire()
 print("Alice pegou brinquedo azul")
 sem_azul.release()
 sem_verm.release()

def bruno():
  print("Bruno quer brincar")
  sem_azul.acquire()
  print("Bruno pegou no azul")
 time.sleep(1)
 print("Bruno quer brincar com vermelho")
 sem_verm.acquire();
 print("Bruno pegou no vermelho")
 sem_verm.release()
 sem_azul.release()

 thread_al = threading.Thread(target=alice)
 thread_br = threading.thread(target=Bruno)

thread_al.start()
thread_br.start()
thread_al.join()
thread_br.join()
```

III →

a)

0	1	2	3	4	5	6	7
X	X	5	4	8	F	0	F
7	B	0	F	0	F	0	0
8	9	A	B	C	D	E	F

Diretoria Raiz: Cluster 2,5
Fich 1: Cluster 3, 4, 8, 7
Fich 2: Cluster 9, 11
Fich 3: Cluster 13

b)

1024 bytes → cada cluster

1200
−1024
176

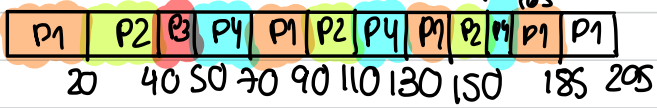
offset 176
Cluster 4

IV)

a)

P1	100	~~80~~	~~60~~	40
P2	50	~~30~~	~~10~~ ✓	
P3	10 ✓			
P4	45	~~25~~	~~5~~ ✓	

Round Robin q=20



b)

tempo médio = ((50 + 40 + 15) + (50 + 40) + 0 + 40 + 30) / 4

= 66,25

c) Round Robin

Vantagens
↳ Tempo limitado para uso do CPU
↳ Não deixa processos pequenos à espera (é cíclico)

Desvantagens
↳ Definição do quantum; pode aumentar sobrecarga ou resposta fraca a processos curtos)
↳ Poucos processos terminam até rodar

Cálculo intensivo → FCFS
Interativo → Round Robin

→ primeira tarefa a chegar é atendida. Por isso não há sobrecarga significativa de troca de contexto (tarefa) e permite que esta continue a executar até estar concluída.

↳ mais responsividade com o utilizador (resposta rápida a eventos de entrada) permite realizar várias tarefas, não se focando só numa

V →

a)

0	F	8	
1	C	9	E
2	A	10	
3	D	11	
4		12	G
5		13	
6		14	
7	B	15	H

b) Um page fault ocorre quando uma aplicação pede para aceder um endereço de memória e ele não está mapeado em RAM.
A resolução deste é função do SO. Este procura onde se encontra a página ausente (arquivo paginação ou memória secundária), carrega a página ausente da mem. secundária para a física atualiza as tabelas de página para refletir localização e reinicia instrução que gerou page fault

c) A memória virtual tem 4 objetivos:

↳ transparência: Processo tem acesso a muita memória (eventualmente mais do que em memória física), crendo como se toda a memória lhe pertencesse

↳ segurança: mecanismos que impeçam que processo altere zona de memória de outro processo.

↳ Eficiência da utilização da memória, devendo ser partilhada entre processos, e mantendo em memória apenas o necessário, cujos endereços não são endereços de memória física

↳ Partilha de Memória - vários processos acedem à mesma zona de memória (de forma controlada)