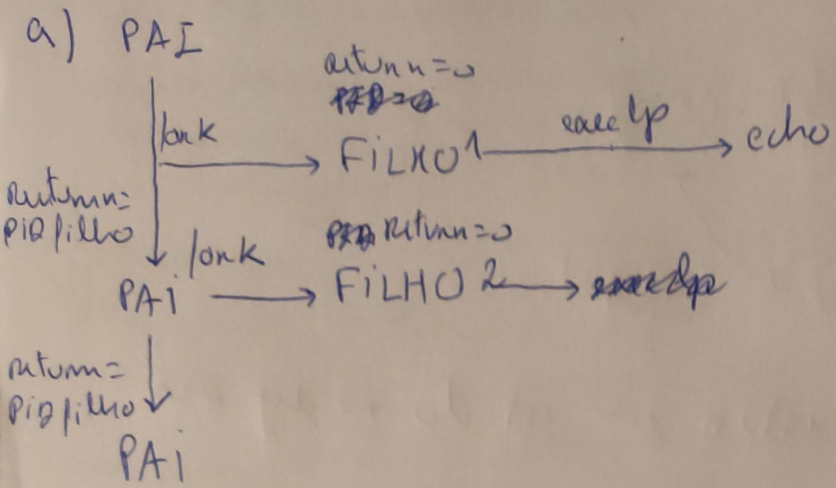
2019 — Normal

1 —

a)



3 processos: 1 pai, 2 filhos.

b)

```
int main (int arg, char *arg[]) → PAI
{
  printf("A\n");  → PAI
  if(fork()!=0){  → PAI + FILHO 1
    fork();  → PAI + FILHO2
    printf("B\n");  → PAI + FILHO2
    printf("C\n");  → PAI + FILHO2
  } else {  → FILHO 1
    printf("D\n");  → FILHO 1
    execlp("echo","echo","sop",NULL);  → FILHO1
```

c)

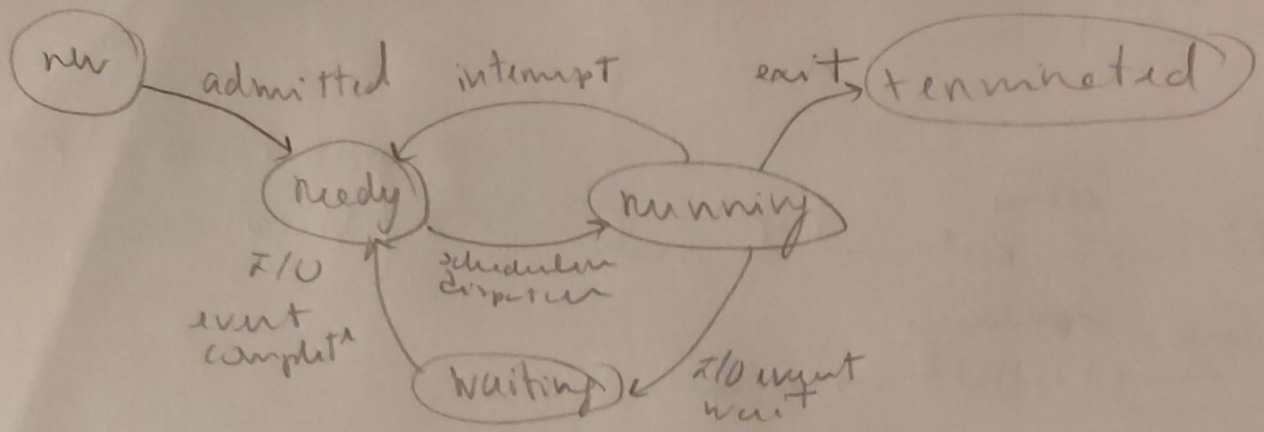
A	A
B	B
C	B
B	C
C	C
D	D
sop	sop

d) Os componentes do Process Control Block são process state, program counter, registos do CPU, CPU-scheduling information, memory-management information, accounting information + I/O status information, process number. Para os vários processos, os campos distintos são: process state, program counter, process number, registos do CPU. Os processos filho herdam do pai, bem como privilégios e atributos de scheduling do pai, certos recursos, como ficheiros.

e) fork — chamada ao sistema. Cria um novo processo que consiste na cópia do espaço de endereçamento do processo pai. Ambos os processos continuam a execução após o fork. O fork retorna 0 no filho e o PID do filho no pai. Dependendo do SO, o fork duplica todas as threads ou só aquela em que foi executado.

exec — chamada ao sistema. Substitui o espaço de memória do processo com um novo programa. (faz load de um ficheiro binário para a memória)

p1

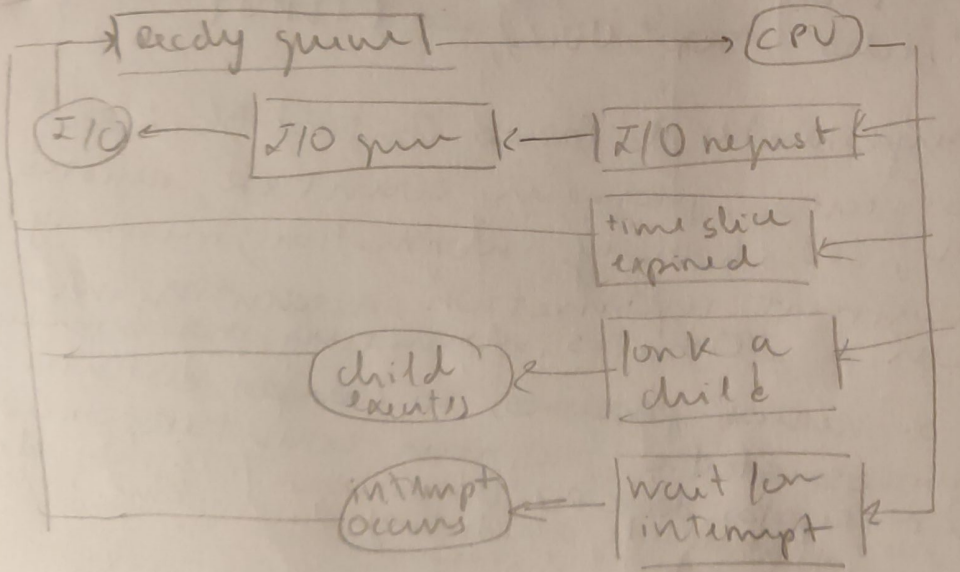


Todas as linhas podem estar em estado ready ou running.

Nas linhas de chamadas ao sistema — 7, 8, 13 — e nas linhas I/O — 9, 10, 12 — podem também estar no estado waiting.

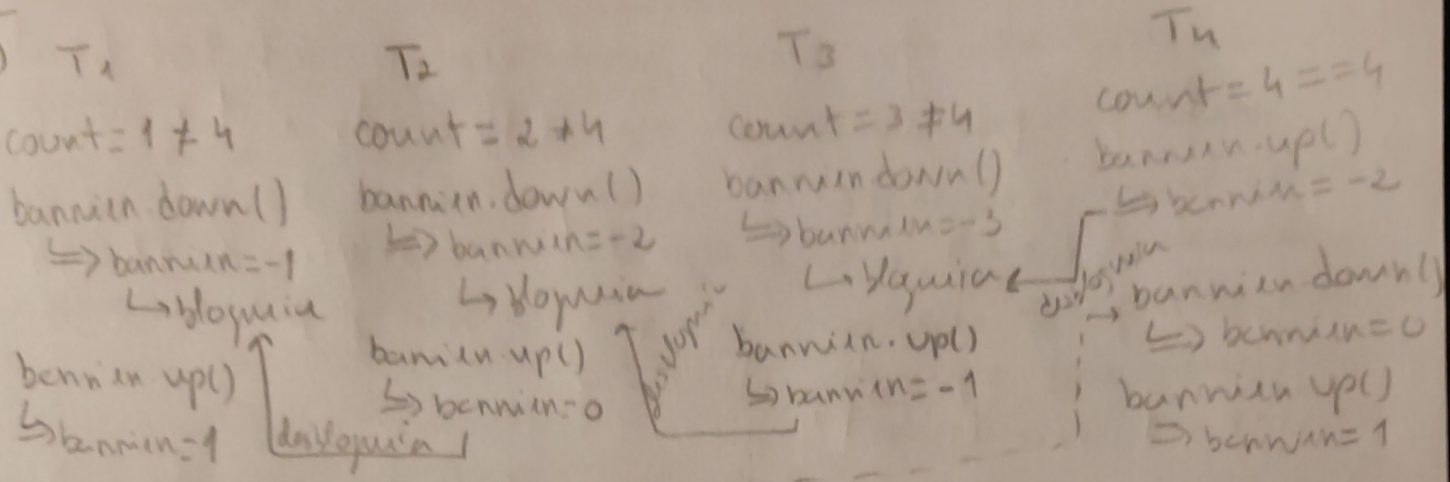
A linha 14 nunca está no estado running, só no ready. Quando os processos terminam, estão no estado terminated.

Funcionamento geral do escalonamento do CPU:



II

a)

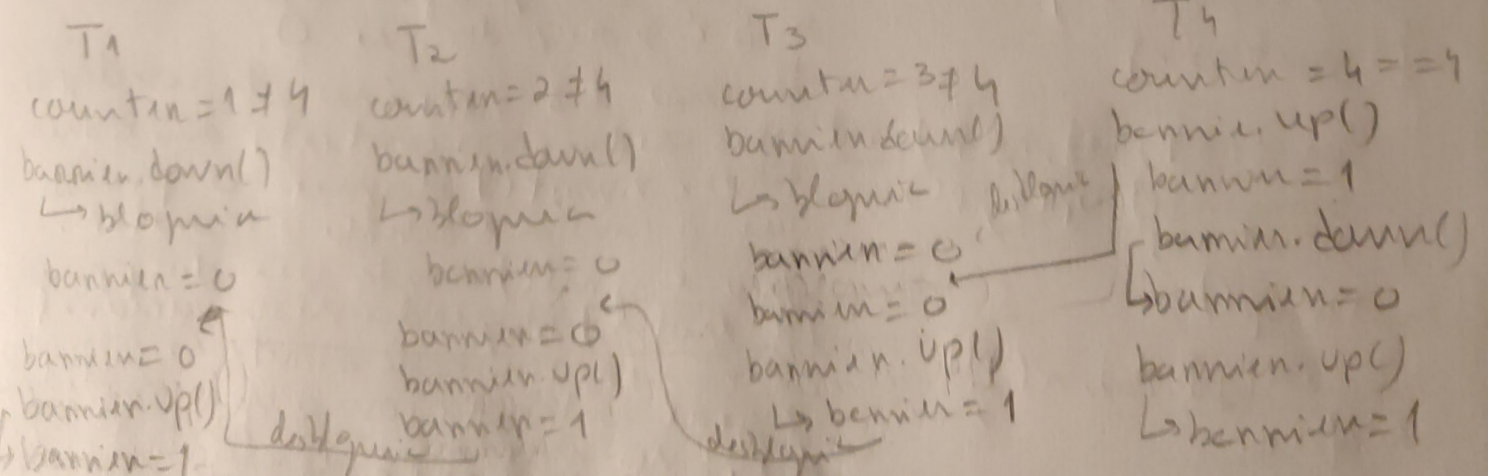


considerando a seguinte implementação:

```
Down()
{
  s--;
  if(s<0)
  {
    adicionar a lista
    de espera
    block();
  }
}
```

```
Up()
{
  s++;
  if(s<=0)
  {
    remove da lista
    de espera
    wake up(p);
  }
}
```

Ou então:



Considerando a implementação:

```
down(s)
{ while(s<=0)
  { }
  s--;
}
```

```
up(s)
{ s++;
}
```

b) A barreira não funciona, uma vez que o valor final do semáforo após correrem as 4 threads é 1 e não 0. Assim, numa primeira iteração, é executado o codeBefore das threads 1, 2, 3 e 4 e só de seguida o codeAfter das threads 1, 2, 3, 4, pelo que a barreira cumpre o seu papel. Mas se iniciarmos uma segunda iteração:

barrier = 1 inicialmente

T1	T2
codeBefore	codeBefore
count = 5 ≠ 4	
barrier.down()	
↳ barrier = 0	
barrier.up()	
⇒ barrier = 1	
codeAfter	

ou seja, pode dar-se o caso de o codeAfter da T1 executar primeiro do que o codeBefore da T2, pelo que o objetivo da barreira se perde

c)

Processo j, j ∈ 1, ... n

```
codeBefore
mutex.down()
count = count + 1;
if (count == n)
{  for (i = 1, i++, i ≤ n)
      barrier[i].up();
   count = 0;
}
mutex.up();
barrier[j].down();
codeAfter
```

d) Os semáforos implicam uma valor inteiro que, à parte da sua inicialização, é acedido apenas por duas operações atómicas. Estas duas operações — up e down — alteram sempre o valor do semáforo, podendo bloquear ou desbloquear um processo ou não ter qualquer ação — depende se S é > ou < do que 0.

No caso das variáveis de condição, estas existem no contexto dos monitores. x.wait() bloqueia o processo que o invoca e liberta o monitor. Já o x.signal() acorda um dos processos (se existir) bloqueado nesta variável. Assim, se não existir, nada acontece. Assim, o signal contrasta com o up na medida em que nos semáforos, o seu estado é sempre alterado, mas no signal pode não ser.

III

a) As condições necessárias à existência de deadlock são as seguintes:

- exclusão mútua: cada recurso, ou está livre, ou foi atribuído a um só processo. Ou seja, a sua posse não pode ser partilhada.
- 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 sobre a libertação de um recurso que lhe tenha sido atribuído.

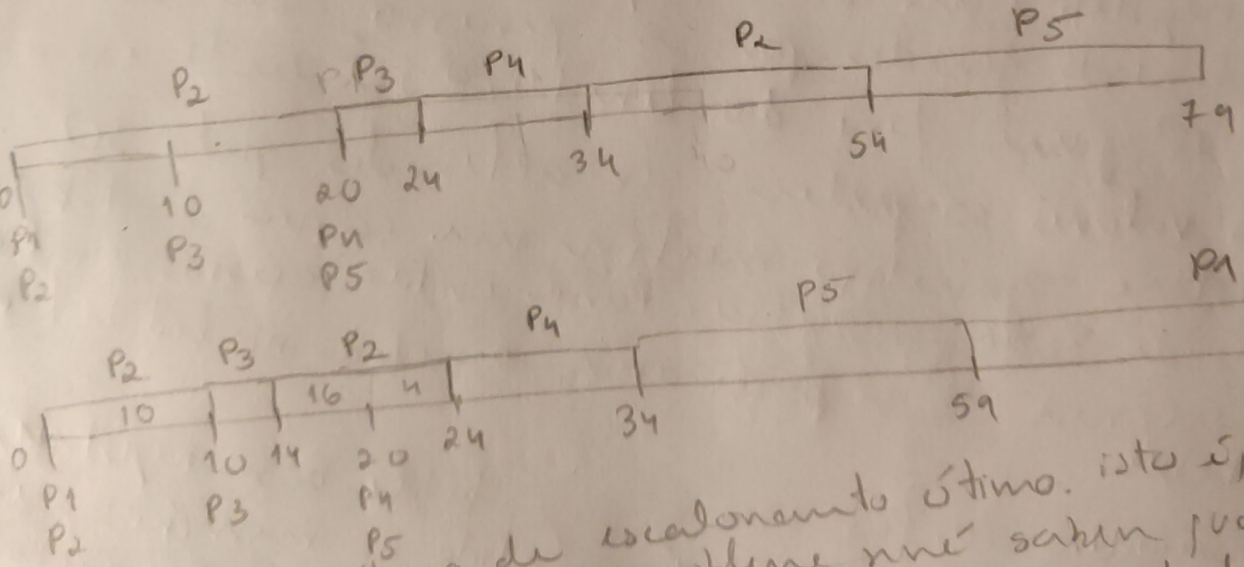
espera circular: formou-se uma cadeia circular de processos
recursos em que cada processo segura um recurso que está
na posse do processo seguinte na cadeia
Para não acontecer deadlock, basta negar 1 destas condições

Neste caso:
Se forem enviados 2 ficheiros que durante o tempo em
que são enviados são transferidos bytes dos 2 ficheiros,
chegando ao tamanho do disco (ou seja, enchendo-o), mas
sem que a transferência de 1 dos ficheiros esteja
completa, então a impressão não se inicia e há deadlock.

b) Podemos negar o princípio da não libertação. Ou seja,
quando o processo não tem os recursos todos para
continuar, tem de libertar tudo e tentar depois.
Ou então, negar o princípio de espera com retenção,
que se traduz em solicitar tudo de uma só vez.
Assim, neste caso, podemos impor o limite de, a partir
do momento em que um ficheiro começa a ser enviado
mais nenhum pode ser enviado. Além disso, antes de
enviar o ficheiro, deve ser verificado se o seu tamanho não
excede o tamanho do disco.

IV

a)
i)



ii)

b) O SJF é o algoritmo de escalonamento ótimo. Isto é, dá
o ST de espera mínimo. O problema é não sabermos qual o
ST do próximo processo. Este ST pode ser submetido pelo
user (usado em long-term scheduling) ou pode ser estimado
com $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$. O SJF é um caso particular de prioridade

V

a) Cada pág tem o tamanho de 1 byte

Tabelas de página:

P1

0	1
1	3
2	5
3	8

↑ nº de página virtual ↑ nº de página física

P2

0	11
1	4
2	0
3	9

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

endereço memória física = nº pág física x 1 byte + offset

Como temos sempre o nº de página virtual, o offset = 0

b) O sistema operativo garante que a tabela de página não é alterada por outros processos. Para isso, há um base register e um limit register. O CPU compara todos os endereços gerados por um processo com esses registos, caso estejam fora dos limites, o SO emite uma trap. Estes valores só podem ser alterados pelo SO. O sistema operativo mantém uma cópia da tabela de página de cada processo. Para evitar acessos inválidos à memória tb se pode usar o valid-invalid bit na page table.

c) A maioria dos PCs usa a page-table base register, uma vez que a tabela de página é mantida na memória principal. Assim, para acedermos a 1 byte, precisamos de efetuar 2 acessos à memória: 1 para aceder à TP e outro para aceder ao próprio byte. Isto provoca um delay. A solução é usar TLB. Este funciona como cache das TB. Existe uma [...] no CPU e se a página estiver lá, dá logo o frame senão, procura na TP

d) Eficiência ⇒ partilha, manter na memória apenas o necessário, e. virtual ≠ e. física.

Segurança ⇒ impedir alteração de conteúdo de memória de outros processos

Transparência ⇒ acesso a mk memória → como como se todo a mónica fosse pertence

Partilha de memória ⇒ vários processos acedem à mm zona de forma controlada