Resolução do exame de FSO de 16 de Janeiro de 2015

PARTE 1

1 a) Em modo sistema é possível executar instruções privilegiadas o que não é possível fazer em modo utilizador

1 b) - ligar/desligar interrupções
- instruções de leitura e escrita nos portos de entrada/saída
- programação da MMU e do temporizador do sistema.

Qualquer uma delas permite ao sistema fazer acesso exclusivo aos recursos do sistema

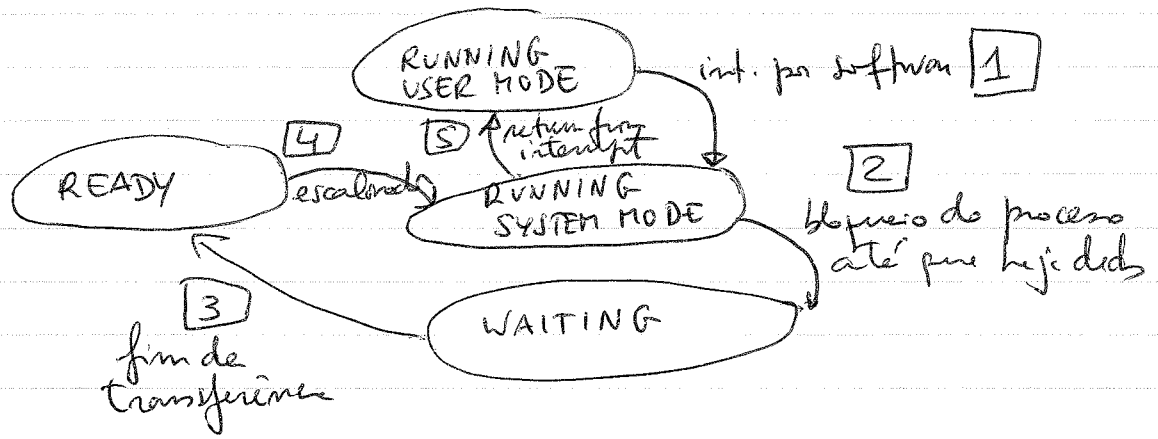
1 c) modo utilizador → modo sistema → interrupção hardware
interrupção por software

1 d) modo sistema → modo utilizador → execução da instrução máquina "return from interrupt"

2 a) A entrada no código do sistema implica várias acções como salvaguarda de parte do estado da computação, mudança do stack, verificação de parâmetros. No final da chamada ao sistema, também se muda de modo do CPU e se restaura parte do estado de computação. Tudo isto significa que uma chamada ao sistema demora (muito) mais do que a invocação de uma função

2 b) A função fread() usa um buffer associado ao canal. Assim sendo, se houver bytes disponíveis no buffer, não é feita uma chamada ao sistema. Logo o programa com fread() é mais rápido do que o programa com read() porque faz menos chamadas ao sistema.

3)



4) São criados 3 processos por fork() logo há 3 cópias de x

Valor de x quando o processo termina
- Processo inicial $x \leftarrow 20$
- 1º filho $x \leftarrow 10$
- 2º filho $x \leftarrow 15$

5) A máquina virtual tem
1 CPU - é completamente separado do CPU virtual do thread criador. O PC/IP recebe o endereço da função argumento do pthread_create e o SP é inicializado para o topo de uma forma privativa
2 Memória - é partilhada com a memória do criador
3 Tabela de canais abertos - é partilhada com o criador

6)
```
// início de uma crítica  do{ l=1; LOCK XCHG(l,b);
                          } while(l==1);
// fim de uma crítica
b = 0;
```

7)
```
func 1  while(1){ sem_wait(&s1);
                  printf("a"); fflush(stdout);
                  sem_post(&s2);
                }

func 2  while(1){ sem_wait(&s2);
                  printf("b"); fflush(stdout);
                  sem_post(&s1);
                }

declarações global sem_t s1; sem_t s2;
main
    sem_init(&s1,0,1); sem_init(&s2,0,0);
```

8) initSem `s->cnt = value;`

8) cont

waitSem

```
pthread_mutex_lock(&S->ex);
if(S->cont > 0) S->cont--;
else {
    while(S->cont == 0)
        pthread_cond_wait(&S->cond, &S->ex);
}
pthread_mutex_unlock(&S->ex);
```

protSem

```
pthread_mutex_lock(&S->ex);
if # fila de S->cond vazia #
    S->cont++;
else pthread_cond_signal(&S->cond);
pthread_mutex_unlock(&S->ex);
```

2ª PARTE

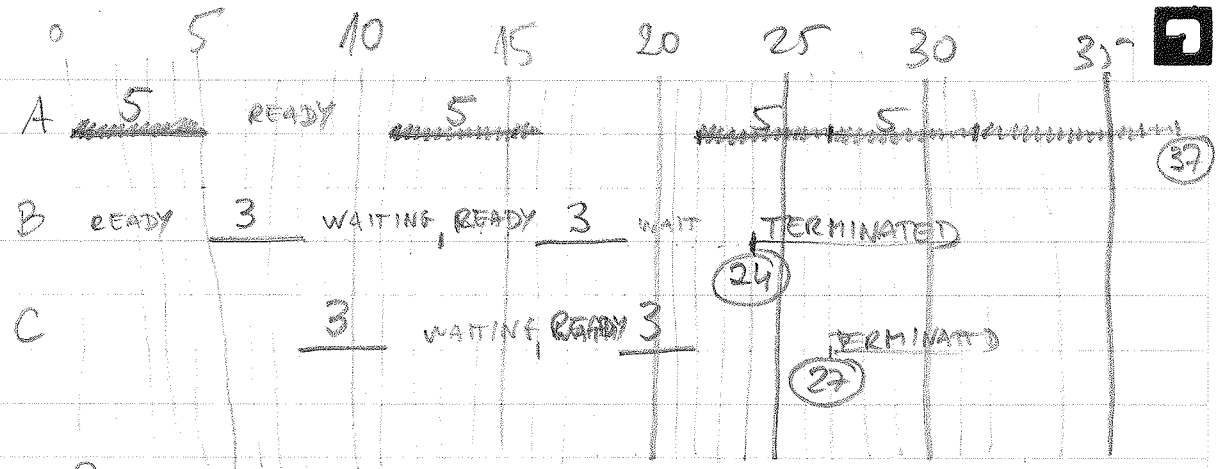
9) a) Os processos CPU Bound consomem apenas o recurso CPU que é escasso. Os processos I/O Bond consomem pouco CPU e usam os outros recursos do sistema. Assim, sendo, ao dar preferência aos I/O bond, mantêm-se todos os recursos em uso

b) Inserindo os processos que passam de WAITING para READY à cabeça da fila READY

c) Um processo CPU Bond praticamente só perde o CPU por passagem RUNNING → READY (1) ou fim de fatia de tempo
Um processo I/O Bond praticamente só perde o CPU por transição RUNNING → WAITING (2).
Assim basta manter para cada processo o nº de transições 1 e 2 efetuadas recentemente

I/O Bond	2	>> 1
CPU Bond	1	>> 2

10)



Processo A - 37 ms, Processo B - 24 ms, Processo C - 29 ms

11) a)

20 | 4096	4100 | 4096	8300 | 4096
20 | 0	4 | 1	108 | 2
d = pv	d = pv	d = pv

b) 8200 | 4096
8 | 1

$e_{fis} = 1 * 4096 + 8 = 4104$

p → pág. fís.

12)

1	PF		1
3	PF		1, 3
2	PF		~~1, 2, 3~~ 1, 3, 2
4	PF		~~1, 2, 3, 4~~ 1, 3, 2, 4
2	hit		
1	hit		
5	PF	vítim 1	3, 2, 4, 5
6	PF	vítim 3	2, 4, 5, 6
2	hit		
6	hit		
1	PF	vítima 2	4, 5, 6, 1
7	PF	vítim 4	5, 6, 1, 7
5	hit		
6	hit		
1	hit		

8 faltas de pág.

13) No caso do fork() o processo filho recebe uma cópia do espaço de endereçamento do pai. Pode-se poupar espaço em memória se as páginas read-only forem partilhadas. Isto acontece por exemplo para as páginas de código; esta parte da tabela de páginas pode ser igual nos dois processos.
Muitas vezes a um fork() segue-se um exec() em que todas as páginas do processo (filho) são deitadas fora. Neste caso estão a reservar e preencher páginas do processo filho é uma perda de tempo. Isto pode ser evitado usando a técnica COW (Copy On Writing)
Aqui a tabela de páginas do pai e filho são inicialmente iguais e todas as página são marcadas read-only. Quando 1 dos processos tenta escrever numa página, a MMU envia uma interrupção e o sistema operativo faz uma cópia da página e marca o original e a cópia como R/W

14 a) Com 16 bits endereçam 65536 blocos (2^{16}). Assim um ficheiro pode ter 2^{16} blocos. No caso o disco tem 2^{15} blocos, pelo que este é o máximo tamanho de blocos de um ficheiro.
b) Sendo endereços de blocos com 4 bytes, temos 512 endereços por bloco. Assim
Nº de blocos máximo de 1 ficheiro = 8 + 512 + 512 = 1032
Tamanho máximo de 1 ficheiro = 1032 × 2K bytes

15 a) Sendo que não há qualquer i-node em RAM
- leitura do i-node ∅
- leitura do bloco mencionado no inode ∅
- leitura do inode I1 do directório /x
- leitura do bloco mencionado no inode I1
- leitura do inode I2 do directório /x/y
- leitura do bloco mencionado no inode I2
- leitura do inode I3 do directório /x/y/z
- leitura do bloco mencionado no inode I3
- neste bloco fica o i-node do /x/y/z/w

8 acessos ao disco.

= abertura do fich /x/f1 para leitura =

15 b) - leitura do i-node I1 do directório /
- leitura do bloco mencionado no i-node I1
- leitura do i-node I2 do directório /x
- leitura do bloco mencionado no i-node I2
- obtenção do i-node B de /x/f1, leitura deste i-node para a tabela de canais abertos

= abertura do fich /y/f1 para escrita =
leitura do inode I4 do directório /y
leitura do bloco mencionado no inode I4
obtenção de um i-node novo I6
preenchimento e escrita do bloco de directório /y c/ a entrada "f2"/I6
colocação do conteúdo do inode I6 na tabela de canais abert

= leitura de todos os bytes do ficheiro /x/f1
leitura do 1º bloco do i-node I3

= escrita dos bytes lidos no fich. com i-node I6
- obtenção de 1 bloco livre B1, actualização do bitmap
- preenchimento do bloco B1, actualização de tab. geral de ficheiros abertos ~~e dos~~ (i-nodes)
- envio do bloco B1 para a cache de blocos e escrita

= close do canal para /x/f1
actualização da tabela FAT e data de acesso no disco

= close do canal para /y/f2
actualização do bloco em disco, directório /y/ e tabela de i-nodes

16) - verificação de validade da informação no bloco 0
e nº mágico, nº de blocos do disco, nº de blocos da FAT

- para cada entrada preenchida no directório, segundo a lista ligada verificar se o nº de blocos corresponde ao comp. do ficheiro e se todos os endereços fazem sentido (de 2 a nº de blocos - 1)

- construir 1 mapa de ocupação de blocos com todas as entradas a LIVRE. Para todas as entradas preenchidas no directório marcar os blocos ocupados. Verificar se há blocos reivindicados por mais de 1 ficheiro. Comparar o mapa construído com o conteúdo do bloco 0.