UNIVERSIDADE DA BEIRA INTERIOR

SISTEMAS OPERATIVOS

Frequência, 4 de Junho de 2024, 18.15 Horas Escala 0:20 Sem Consulta Duração: 1h45m

- Por norma não se esclarecem dúvidas. Se tiver dúvidas, indique na folha de teste a sua interpretação.
- Utilize uma caligrafia legível.

Grupo A: História, Estrutura e Arquitetura dum Sistema Operativo (2 Valores)

1. Explique a diferença entre uma chamada a uma função que é uma chamada ao sistema e uma função normal. A sua resposta deve definir os termos "User Mode" e "Kernel Mode" (Modo de Utilizador e Modo de Kernel). Neste contexto, como escreveria o funcionamento da função de biblioteca padrão printf("teste\n") ?

Grupo B: Programas, Processos e Escalonamento (2 Valores)

2. Qual é o output do seguinte programa?
   Deverá mostrar o *trace do* funcionamento do programa.
   O output é determinístico? Podemos de facto afirmar alguma coisa sobre o output? Justifique!

```
int pid, status, x = 1;
pid = fork();
if ( 0 == pid ) {
    pid=fork();
    if (0==pid) x=x+2;
 }
else {
    wait(&status);
    fork();
    x=x-2;
 }
printf("x=%d\n",x);
```

Grupo C: Gestão de Memória (4 Valores)

3. Qual é a diferença entre uma página (page) e uma moldura (frame)? Explique. Como é que estão relacionadas?

4. Um sistema de memória virtual tem um tamanho de página de 16 palavras, 4 páginas virtuais e 3 páginas físicas. A tabela de páginas está no estado apresentado em baixo. Um endereço virtual neste sistema tem 7 bits, sendo que os primeiros 3 bits indicam o número de página. Para os seguintes endereços diga apenas se (a) o resultado é uma page hit/miss (falha/sucesso) ou é uma interrupção/trap para o sistema operativo e (b) calcule o endereço físico quando apropriado - em valor decimal.

(a) 011 0101
(b) 100 1001
(c) 001 0110
(d) 110 0011
(e) 000 1001
(f) 010 0001

Página virtual	Página física	*Válido/Inválido*
0	2	*1*
1		*0*
2	1	*1*
3	0	*1*

Grupo D: Concorrência, Sincronização e Bloqueio (7 valores)

5. Responde as perguntas seguintes considerando o seguinte programa em baixo.

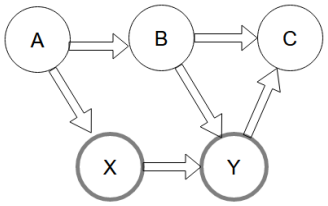
```
int x=0;
void *maisx(void *args)   {
    int id = *(int*)args ;
    x=x+id;
}
int main() {
   pthread_t  th[3];int ids[3]={1,3,5};
   for (int i=0; i<3; i++) pthread_create( &th[i],NULL, maisx, &ids[i]);
   for (int i=0; i<3; i++) pthread_join( th[i], NULL );
   printf("x=%d\n",x);
```

a) Quais são os outputs possíveis do programa?
b) Explique detalhadamente (faça uso dum *program trace*, pseudo-código, fluxograma etc) como é que o output poderá ser "x=6"
a) Usando a sintaxe de Posix *Threads* explique como é que pode usar um trinco logico de exclusão mútua para garantir que o resultado deste programa seja "x=9".

6. Considere um sistema de duas threads concorrentes (as linhas do código são numeradas). Este sistema tem de sincronizar uma sequência de eventos A,B e C feitos sequencialmente na thread ***um*** e eventos X e Y feitos sequencialmente na thread ***dois***. Os eventos são representados por "printfs" e tem que seguir uma ordem especifica de execução mostrado no grafo acíclico mostrado em baixo onda as arestas representam uma regra de precedência. Visto que este sistema executa numa plataforma com recursos reduzidos a empresa que desenhou este sistema tentou minimizar o número de semáforos e usou apenas dois. Infelizmente o sistema pode entrar em deadlock (ver segunda execução em baixo)

   - Explique como este sistema possa entrar em "Deadlock" fazendo um grafo de alocação de recursos. Deve também indicar os passos necessários para o Deadlock acontecer

```
sem_t t1, t2;
sem_init(&t1, 0, 0);
sem_init(&t2, 0, 0);
void *um(void *args){
1    printf("Evento A\n");
2    sem_post(&t1);
3    printf("Evento B \n");
4    sem_post(&t2);
5    sem_wait(&t1);
7    printf("Evento C\n");
}
void *dois(void *args){
7     sem_wait(&t1);
8     printf("Evento X\n");
9     sem_wait(&t2);
10    printf("Evento Y\n");
11    sem_post(&t1);
}
```




```
Linux> ./a.out
Evento A
Evento B
Evento X
Evento Y
Evento C
Linux> $ ./a.out
Evento A
Evento B
Evento C
```

Grupo C: Exercícios Práticos (5 valores)

7. (i) Neste exericio irá escrever uma nova versão da função, socp(), agora otimizada, para copiar um ficheiro (fonte) para um novo ficheiro (destino). Nesta nova versão o tamanho de bloco para escrita e leitura deverá ser definido através do atributo "st_blksize" obtido através a função "stat" ou "fstat" do ficheiro fonte (pode assumir que o tamanho otimal é igual para leitura e escrita)

   ```
   int socp ( char *fonte, char *destino );
   ```

   A função deverá abrir o ficheiro (o parâmetro fonte) e ler o ficheiro para memória em bloco usando I/O de baixo nível também deverá escrever para os ficheiros novos em blocos do mesmo tamanho.

   Caso qualquer dos ficheiros não puder ser aberto ou criado ou em caso de qualquer outro erro a função deverá terminar e devolver o valor -1. No caso de ter sucesso a função deverá devolver o valor 1.

   Não é necessário especificar as bibliotecas padrão (#include <fcntl.h> etc.) no seu programa.

   (ii) Explicar detalhadamente como pode usar esta função no "shell" desenvolvido durante as aulas praticas para implementar um novo comando embutido no seu shell chamado "optimalcopy". Pode assumir que o codigo da parte (a) está escrita no ficheiro shellio.c