U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA E COMPUTAÇÃO | 2º ANO
EIC0027| SISTEMAS OPERATIVOS | 2019/2020 – 2º SEMESTRE

EXAME - ÉPOCA NORMAL| 2020-06-19

Com consulta. Duração: 1h30m

Nome do estudante: _____ **Código UP:** _____ **Nº Prova:** _____

1. [4.5]

Indique quais das seguintes afirmações são verdadeiras e quais são falsas, assinalando respetivamente com **V** ou **F.**

NOTA - A pontuação nesta pergunta será dada pela fórmula: máximo (0, (nº_respostas_certas – nº_respostas_erradas) * 0.3)

	V / F
Em programação POSIX, uma *condition variable* deve ser sempre usada em associação com um *mutex*.	V
O código que implementa as primitivas **wait()** e **signal()** que operam sobre semáforos constitui uma secção crítica que pode ser implementada com recurso a instruções máquina especiais como, por exemplo, *test-and-set*.	V
No contexto do escalonamento de processos, a política de "envelhecimento" tem em vista evitar a inanição de alguns processos.	V
O código seguinte escreve na saída padrão o número **100**: **int x=100; write(STDOUT_FILENO,&x,3);**	F
Uma secção crítica é um pedaço de código que tem de ser executado sem interrupções.	F
É possível que dois processos acedam por endereços lógicos diferentes ao mesmo endereço físico.	V
Em Linux, na sequência de uma chamada **fork()**, o processo pai e o processo filho partilham as variáveis globais, podendo o processo pai manipular as variáveis do processo filho e vice-versa.	F
A multiprogramação só pode ser implementada em computadores com vários processadores (*CPUs*) ou com um processador multinúcleo (*multicore*).	F
Quando vários processos pretendem acrescentar informação no fim de um ficheiro partilhado, convém que o ficheiro seja aberto em modo **O_APPEND**, caso contrário, a operação de escrita deve ser feita numa secção crítica envolvendo as chamadas **lseek()** e **write()**.	V
Uma das condições necessárias para garantir a segurança do sistema operativo é o processador (*CPU*) suportar um modo de operação privilegiado.	V
No contexto da gestão de memória virtual a ocorrência de uma falta de página leva à terminação imediata do processo que a originou.	F
Um dos problemas da utilização de sinais como mecanismo de comunicação entre processos é que pode acontecer que nem todos os sinais enviados a um processo sejam recebidos.	V
Uma das vantagens do uso de *threads* sobre o uso de processos é que não é necessário usar memória partilhada ou mecanismos de intercomunicação, como *pipes*, para a comunicação entre *threads* de um mesmo processo.	V
Em Linux, as funções **signal()** e **pthread_cond_signal()** têm funcionalidades semelhantes, devendo a primeira ser usada com processos e a segunda usada com *threads*.	F
As chamadas **exec** fazem simultaneamente duas coisas: criam um novo processo e carregam o código de um programa a executar nesse processo.	F

Nome do estudante: _____ **Código UP:** _____ **Nº Prova:** _____

2. [4.5]

a) [2.0] Complete as frases seguintes com uma palavra ou um curto texto.

O mecanismo de *hardware* conhecido por DMA (*Direct Memory Access*) é fundamental para … a implementação da multiprogramação.
Ao longo da sua existência, os processos/*threads* vão comutando entre 3 estados principais que são: … pronto, em execução, bloqueado.
A forma mais simples de prevenir *deadlocks* nos programas dos utilizadores é … ordenar os recursos e requisitá-los sempre pela mesma ordem.
Uma elevada atividade do disco acompanhada de uma baixa utilização do processador pode ser um sintoma de … *thrashing.*
Em Linux, a estrutura de dados que descreve ficheiros e diretórios armazenados em disco é designada por … *i-node.*

b) [1.5] Três *threads*, **T1**, **T2** e **T3**, de um mesmo programa, têm três blocos de código, respetivamente, **f1()**, **f2()** e **f3()** que se pretende que sejam executados sequencial e repetidamente, começando sempre por **f1()**, seguindo-se **f2()** e **f3()**, por qualquer ordem, mas com a restrição que **f2()** e **f3()** nunca devem ser executados simultaneamente**; f1()** só será executado novamente após **f2()** e **f3()** terem concluído a sua execução**;** isto é, a sequência de execução deverá ser:
f1() → [f2()→f3() ou **f3()→f2()]** e novamente **f1()→ [f2()→f3()** ou **f3()→f2()] - …**.

Recorrendo a semáforos (apenas), indique como implementaria a parte do código de **T1**, **T2** e **T3** que garante a referida sequência de execução; indique também como deve ser feita a inicialização dos semáforos.

Considere que dispõe das funções **init(sem,value)**, **wait(sem)** e **signal(sem)** que operam sobre semáforos.

T1
```
...
while(1) {
 ...
 f1();
 ...
}
...
```

T2
```
...
while(1) {
 ...
 f2();
 ...
}
...
```

T3
```
...
while(1) {
 ...
 f3();
 ...
}
...
```

inicialização dos semáforos	**T1**	**T2**	**T3**
init (sem1, 0) init (sem23, 0) init (semX, 1)	f1 () signal (sem23) signal (sem23) wait (sem1) wait (sem1)	wait (sem23) wait (semX) f2 () signal (semX) signal (sem1)	wait (sem23) wait (semX) f3 () signal (semX) signal (sem1)
A solução anterior não está totalmente correta, pois permitiria que T2 ou T3 executassem 2 vezes consecutivamente. A solução correta é a seguinte:			
init (sem1, 0) init (sem2, 0) init (sem3, 0) init (semX, 1)	f1 () signal (sem2) signal (sem3) wait (sem1) wait (sem1)	wait (sem2) wait (semX) f2 () signal (semX) signal (sem1)	wait (sem3) wait (semX) f3 () signal (semX) signal (sem1)

c) [1.0] Num sistema que usa paginação a pedido (*demand paging*) qual a importância de cada página ter associado um bit de referência (*reference bit*) e um bit de modificação (*modified bit*)?

Ambos facilitam o bom funcionamento do mecanismo de troca de página, quando já não houver páginas físicas (quadros) disponíveis:
- o bit de referência permite saber quais as páginas em memória física que não foram acedidas recentemente; de acordo com o algoritmo de substituição de páginas LRU (Least Recently Used) as páginas acedidas menos recentemente devem ser as primeiras a ser substituídas;
- o bit de modificação permite saber quais as páginas em memória física que não foram alteradas desde que foram carregadas na memória física (ou desde a última ronda do mecanismo de troca de página); tipicamente, as que não foram alteradas são substituídas mais cedo, porque as que foram modificadas, se forem substituídas, terão de ser escritas no disco, o que implica um gasto de tempo adicional para serem substituídas.

Nome do estudante: _____ **Código UP:** _____ **Nº Prova:** _____

Nota: nesta prova apenas é necessário fazer tratamento de erros e indicar os ficheiros de inclusão quando tal for solicitado explicitamente

3. [4.5]

O programa **removebig** remove de um diretório todos os ficheiros regulares cujo tamanho seja superior a um dado valor. Os argumentos do programa são o nome do diretório e o referido tamanho (exemplo de invocação para remover os ficheiros do diretório **dir1** cujo tamanho seja superior a 1000000: **removebig dir1 1000000**). **Nota**: os subdiretórios devem ser ignorados.

a) [2.0] Escreva a parte do código de **removebig** que percorre o referido diretório e, por cada ficheiro regular encontrado cujo tamanho seja superior ao indicado, escreve o nome do ficheiro na saída padrão, incrementa um contador do número de ficheiros a remover (variável **count**) e invoca a função **void remove(char *filename)**, cujo parâmetro é o nome do ficheiro a remover. Não implemente por enquanto esta função.

```
while ((entry = readdir(dir)) != NULL)
{
  stat(entry->d_name, &statbuf);
  if (S_ISREG(statbuf.st_mode) && statbuf.st_size > atoi(argv[2]))
  {
    printf("%s\n", entry->d_name);
    rmcount++;
    if (fork() == 0)
    {
      //Visto que não eram dadas instruções especificas de
      //onde seria executado o programa, tanto podia ser passado
      //apenas o entry-d_name, como o path construido.
      //  char *const args[] = {"rm", entry->d_name, (char *)0};
      //  execv("/bin/rm", args);
      //ou
      char *fullpath = malloc(strlen(argv[1]) + strlen(entry->d_name) + 1);
      sprintf(fullpath, "%s/%s", argv[1], entry->d_name);
      remove(fullpath);
    }
  }
}
```

b) [1.5] Escreva o código da função **remove()** que remove o ficheiro cujo nome recebe como parâmetro, recorrendo para isso a um subprocesso (processo filho) que executa o utilitário **rm**. **Nota**: não use chamadas **system()**.

```
int remove(char* filename)
{
  char *const args[] = { "rm", filename, (char *)0 };
  execv("/bin/rm", args);
}
```

c) [1.0] Escreva a parte do código de **removebig** que espera que todos os processos filhos terminem e escreve na saída padrão o número total de ficheiros removidos.

```
pid_t kidpid;
int status;

while ((kidpid = wait(&status)) > 0) { }

printf("%d", rmcount);
```

Nome do estudante: _____ **Código UP:** _____ **Nº Prova:** _____

4. [4.0]

Considere dois programas (**server** e **client**). O processo **server** recebe, através de um *pipe* com nome (FIFO), "mensagens" que lhe são enviadas, direta ou indiretamente, por um ou mais processos **client**; as mensagens são *strings* de tamanho fixo; o processo **server** termina quando receber uma mensagem vazia (""). Um processo **client** recebe como argumento da linha de comando o nome de um ficheiro e recorre a um programa auxiliar (uma versão personalizada do programa **md5sum**) para calcular o "resumo criptográfico" do ficheiro (um número hexadecimal). O programa **md5sum** recebe como argumento o nome de um ficheiro e apresenta na saída padrão o referido número hexadecimal (ver abaixo um exemplo de execução de **md5sum**, na linha de comandos). Pretende-se enviar a saída do programa **md5sum** para o processo **server**. O programa **server** mostra essa saída (ou "mensagem") no ecrã e faz um processamento que não importa especificar, neste contexto.
Foram desenvolvidas duas implementações alternativas para o programa **client** que se pretende comparar e avaliar.

Exemplos de execução

Programa **client**:

```
$ ./client image.png
$ ./client document.pdf
$ ./client ""
//NOTA: envia mensagem vazia ao servidor
```

Programa **server**:

```
$ ./server
dcf92388dd338dbb74ad1b08e61091a7
7b7a53e239400a13bd6be6c91c4f6c4e
//NOTA: servidor foi encerrado (foi recebida uma mensagem vazia)
```

Programa **md5sum**:

```
$ md5sum image.png
dcf92388dd338dbb74ad1b08e61091a7
```

Programa **client**

```
// #include ...

#define MAX_SIZE 32
#define CHANNEL "/tmp/server"

int main(int argc, char *argv[]) {
  char cmd[256], msg[MAX_SIZE] = "";
  int fd;

  while ((fd = open(CHANNEL, O_WRONLY)) < 0) ;

  if (argv[1][0]=='\0') {
    write(fd, msg, MAX_SIZE);
    close(fd);
    return 0;
  }
  ... // código da implementação (ver alternativas abaixo)
}
```

Implementação 1

```
snprintf(cmd, sizeof(cmd), "md5sum %s", argv[1]);
FILE *f = popen(cmd, "r");
fread(msg, MAX_SIZE, 1, f);
write(fd, msg, MAX_SIZE);
close(fd); fclose(f);
return 0;
```

Implementação 2

```
dup2(fd, STDOUT_FILENO);
close(fd);
execlp("md5sum", " md5sum", argv[1], NULL);
exit(1);
```

a) [1.0] Explique sucintamente a diferença entre as duas implementações; recorra a figura(s), se achar conveniente.

Implementação 1	Implementação 2
* Referir que é criado um pipe [função popen()] entre os programas client e md5sum; * Referir que o programa client procede à leitura dos dados através desse pipe e que os envia posteriormente ao programa server através do FIFO CHANNEL.	* Referir que o programa client será substituído pelo programa md5sum após a invocação da função execlp(); * Referir que o redireccionamento (feito previamente) permite que o programa md5sum envie os dados directamente para o programa server através do FIFO CHANNEL.

b) [1.5] Indique, se existir e justificando sempre com uma frase, qual a implementação mais indicada nos seguintes casos: **(1)** **md5sum** escreve na saída caráter a caráter, **(2)** **md5sum** escreve na saída de uma só vez, **(3)** MAX_SIZE passa a ser superior a PIPE_BUF.

(1) Implementação 1: nunca ocorre sobreposição/entrelaçamento de mensagens (uma única escrita atómica);
(2) Implementação 2: mais simples e mais eficiente em termos de recursos utilizados;
(3) Nenhuma: a partir do momento em que a extensão das mensagens excede PIPE_BUF, deixa de ser possível garantir a escrita atómica das mesmas.

c) [1.5] Complete o código do programa **server**.

```
int main() {
  char msg[MAX_SIZE + 1] = { '\0' }; // todas os elementos serão preenchidos com '\0'
  int fd, nr;
  mkfifo(CHANNEL, 0644);
  fd = open(CHANNEL, O_RDWR);
  while (((nr = read(fd, msg, MAX_SIZE)) > 0) && (msg[0] != '\0')) {
    printf("%s\n",msg);  // ... escrita da mensagem recebida
    // ... processamento da mensagem recebida
  }
  unlink(CHANNEL);
  return 0;
}
```

Nome do estudante: _____ **Código UP:** _____ **Nº Prova:** _____

5. [2.5]

Recorde o problema **2.b** desta prova. Considerando uma implementação baseada em semáforos e/ou mutexes POSIX, responda às questões seguintes.

a) [0.9] Escreva o cabeçalho da função **thread2()**, que implementa o *thread* designado por **T2** no problema **2.b**, e as partes do seu código referentes à sincronização da execução com os restantes *threads*. Considere que o *thread* não recebe parâmetros.

```
void * thread2 (void * arg);
...
sem_wait (&sem23);
pthread_mutex_lock (&semX);
f2 ();
pthread_mutex_unlock (&semX);
sem_post (&sem1);
```

b) [0.8] Tendo em conta que os *threads* pertencem ao mesmo programa, diga onde deve(m) ser definido(s) o(s) semáforo(s) e/ou mutex(es) e onde e quando deve ser feita a sua inicialização. Dê um exemplo de declaração e inicialização de cada tipo de mecanismo de sincronização que selecionou.

```
// definidos como variáveis globais:
sem_t sem1, sem23;
pthread_mutex_t semX;

---

// invocado no thread principal, antes da formação de T1, T2 e T3:
sem_init(&sem1, 0, 0);
sem_init(&sem23, 0, 0);
pthread_mutex_init(&semX, NULL);
```

c) [0.8] Se, em vez de pertencerem ao mesmo programa, os *threads* **T1**, **T2** e **T3** pertencessem a programas diferentes, que alterações seria necessário introduzir na definição e inicialização dos mecanismos de sincronização utilizados? Não escreva código.

Seria necessário:

1ª alternativa:

- utilizar unicamente semáforos com nome, criados com as permissões adequadas à partilha por outros processos e inicializados por um só processo;

2ª alternativa:

- definir os semáforos anónimos e mutexes numa zona de memória partilhada por todos os processos intervenientes;
- no caso dos semáforos (anónimos), a sua inicialização deveria ser feita com o 2º argumento de sem_init (sem_t *sem, int pshared, unsigned int value) com um valor diferente de 0;
- no caso de mutexes, a sua inicialização deveria ser feita com um atributo, 2º argumento de pthread_mutex_init (pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr), que foi especificado para ser usado por diferentes processos: pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared), com pshared = PTHREAD_PROCESS_SHARED.

FIM