Questão 11. Num sistema Unix, considere a seguinte sequência de operações, num processo P1, admitindo que, inicialmente, o ficheiro **/temp/f1** existe, mas está vazio, e o ficheiro **/temp/f2** não existe. Admita que o processo tem todos as permissões necessárias para aceder aos ficheiros e que os seus canais standard 0, 1, 2 mantêm os valores herdados do processo *shell* interativo que tenha criado P1.

```
processo P1:
char buf1[2], buf2[10];
int nw, fd1, nr, fd2;
...
buf1[0] = 'a';
buf1[1] = 'b';
/*sequência de instruções A */
link('/temp/f1', '/temp/f2');
fd1 = open('/temp/f1', O_WRONLY);
nw = write(fd1, buf1, 2);
unlink('/temp/f1');
close(fd1):
/*sequência de instruções B*/
fd2 = open('/temp/f2', O_RDONLY);
nr = read(fd2, buf2, 10);
close(fd2);
unlink('/temp/f2');
...
```

a) Acções se o programa acima, for executado por um único processo, de forma sequencial, seguindo a ordem indicada acima:

sequencia de instrucoes A

-- **link():** aparece um novo nome ´´f2´´, na directoria /temp que representa um novo nome (sinónimo ou *alias*) do mesmo ficheiro que já existia antes com o nome /temp/f1. Ou seja estas duas entradas naquela directoria, ainda que com nomes simbólicos diferentes, têm o mesmo i-node associado (recorde que o i-node é um descritor interno ao sistema de ficheiros no Unix que descreve os atributos de cada ficheiro).

-- **open()**: é aberto um canal para o ficheiro, com mode de escrita, sendo o número de canal devolvido na variável fd1. Recorde que o número de canal (chamado *file descriptor* no Unix) representa uma entrada numa tabela privada a cada processo – a Tabela de Canais do Processo – havendo uma destas entradas por cada canal aberto pelo processo para um ficheiro. No caso do enunciado, como é dito que as três primeiras entradas nesta tabela (0, 1, 2) estão já inicializadas (o que é vulgar pois os processos herdam os canais abertos pelo processo pai e em geral para processos criados através do terminal, o processo pai executa o programa do Shell (interpretador de comandos) cujos canais 0, 1, 2 estão respectivamente ligados ao teclado, écran e écran.), isto significa que a primeira entrada livre na tabela de canais do processo é a número 3, pelo que, segundo a especificação da chamada ao sistema open(), ficará então a variável fd1 igual a 3.

-- **write():** o conteúdo do buffer buf1 (´a´ seguido de ´b´) é escrito através do canal fd1, antes aberto. Como a chamada write() indica escrita de 2 bytes (no 3º argumento), ambos os caracteres a e b será escritos. A chamada devolve em nw o número de bytes escritos ou seja 2. Reveja a estrutura de tabelas do sistema de ficheiros do Unix, para ver onde ficam estes bytes guardados: além da Tabela de canais de cada processo (neste caso onde temos agora a entrada 3 com o canal aberto para o ficheiro), existe uma outra tabela – a Tabela Global de Ficheiros Abertos-- , esta no entanto é global (ou seja descreve informação para todos os processos, ao contrário da tabela de canais do processo, que lhe é privada) e tem uma entrada por cada canal aberto, na qual indica a seguinte informação:

-- o modo de abertura do canal, neste caso para escrita (write_only) como vimos

-- o valor do cursor de byte (byte offset) que indica a posição corrente onde se vai ler ou escrever no canal (recorde que no Unix cada ficheiro é apresentado como uma sequencia de bytes, de 0 até ao último, sendo as operações de read/write de acesso sequencial, ou seja usam a posição corrente do cursor de byte para ler/escrever a partir daí. Para ter acesso directo a uma dada posição algures no ficheiro, que não seja a posição corrente do cursor, é preciso primeiro chamar a chamada ao sistema lseek() que coloca o cursor na nova posição desejada, podendo depois ler-se ou escrever-se a partir daí).

-- um apontador para uma outra tabela global do sistema, chamada a Tabela de i-nodes em Memória: esta tabela contém uma entrada com o i-node de cada ficheiro que está correntemente em utilização (ou seja em relação ao canal algum processo pediu, usando open, para abrir um canal. No inicio estes i-nodes estão em disco, e é a operação open que, no seu primeiro passo, vai trazer para a tabela de i-nodes em memória, os i-nodes necessários.

O conteúdo de cada i-node tem toda a informação necessária para operar sobre um ficheiro:

- o tipo de ficheiro (normal em disco, directoria, ou especial)
- o nome/identificador do dono (geralmente o utilizador que o criou)

- as permissões de acesso read-write-execute para o dono, grupo e outros utilizadores
- as datas em que foi acedido
- a localização dos blocos onde estão os dados do ficheiro em disco, no caso de ficheiros em disco, ou a indicação das rotinas do Device Driver se for um ficheiro especial (que representa um periférico),
- outros atributos com informação sobre o ficheiro

Em particular, quando um ficheiro já tem um canal aberto, o seu inode já está em memória (na Tabela de inodes em memória), e é neste inode que o SO vai indicar os apontadores para os buffers que irão guardar os bytes do ficheiro, quando há operações read/write. Estes buffers estão em memória, para evitar que o SO tivesse de estar constantemente a aceder a disco, mesmo que o programa só leia ou escreva alguns bytes. No caso do processo em causa, num buffer associado ao inode do ficheiro /temp/f1, irão ficar os dois bytes escritos por write, ou seja ´a´ e ´b´. Quando é que estes bytes serão escritos em disco? Um dos casos é quando todos os canais para o ficheiro forem fechados (por close), nesse caso o SO actualiza os blocos do ficheiro em disco, a partir dos buffers em memória. Outro caso é quando esse buffer em memória fica cheio e nesse caso é escrito em disco para permitir que novos write possam escrever em memoria. Outro caso é periodicamente, ou seja o SO vai de tempos a tempos, actualizando em disco todos os blocos dos ficheiros abertos, a partir dos seus buffers em memoria (uma operacao invocada automaticamente pelos programas do SO que se chama *sync*, que actualiza os dados em memoria e em disco, pondo-os iguais).

Assim, ao fazer um write(), o SO vai primeiro à Tabela de Canais do processo, entrada número 3 neste exemplo, daí encontra um apontador para a TabelaGlobalFicheirosAbertos, onde está uma entrada descrevendo esse canal (fica a saber em que posicao vai o cursor de byte) e daí encontra um apontador para o inode do ficheiro na Tabela de inodes em memoria, onde estão os buffers associados, onde irá escrever. Assim:

TabelaCanaisProcesso -→ TabelaGlobalFicheirosAbertos → Tabela-inodes-em-memoria → buffers do ficheiro

Note que se depois disso, uma operação read for pedida sobre um canal aberto para esse ficheiro, os bytes poderão ser lidos a partir daqueles buffers.

-- unlink(): o nome /temp/f1 é removido da directoria /temp. Note que o ficheiro se mantém acessível a partir do seu outro nome /temp/f2.

-- **close():** o canal antes aberto (file descriptor ou numero de canal 3) é fechado. A entrada respectiva na Tabela de canais do processo é descartada, a entrada associada ao canal na TabelaGlobalFicheirosAbertos também é descartada (pois não há mais canais apontando para essa entrada neste exemplo) e a entrada onde está o inode do ficheiro em memoria pode ser libertada, por não haver mais canais abertos para o ficheiro. Note no entanto que, antes de descartar o inode em memoria, o SO escreve os buffers (onde estao os caracteres a e b) em disco, nos blocos correspondentes do ficheiro.

Sequencia de instrucoes B: tente resolver esta parte por si próprio (se tiver dúvidas contacte o docente: jcc@fct.unl.pt).

b) Acções se o programa acima, for executado por DOIS processos, de forma concorrente, em que um deles (chamemos-lhe P1) executa a sequência A e o outro (P2) executa a sequência B.

A execução de P1 e P2 de forma concorrente é conseguida através do regime de multiprogramação (reveja os apontamentos/acetatos das aulas). Neste caso, o SO vai gerir a execução dos processos, por forma a dar-lhes a oportunidade de executarem as suas instruções, mas o SO também vai tentar garantir que os recursos do computador são bem utilizados. A consequência disso é que nós não conseguimos prever a ordem pela qual os dois processos vão executar aquelas instruções das sequências A e B. Ou mais exactamente, tudo o que sabemos é que o processo P1 vai executar, por si, a sequencia A pela ordem indicada no programa e sabemos que o processo P2 vai executar, por si, a sequencia B pela ordem indicada no programa. Mas não sabemos se o processo P1 consegue executar a sua 1ª instrucao link(), por exemplo, antes ou depois de P2 executar a sua 1ª instrucao open(), pois a ordem pela qual P1 e P2 é decida pelo SO e é imprevisível. Isto pode originar erros, que é o que se pergunta nesta questão.

-- se P2 executar open(/temp/f2,...) antes de P1 ter executado link(/temp/f1, /temp/f2), entao o nome f2 ainda nao existe pelo que o open em P2 vai dar erro em fd2 que receberá o valor (-1), indicativo de erro. E nesse caso todas as instrucoes seguintes de P2 irão dar erro. Eventualmente, a última instrucao da sequencia B em P2: unlink(/temp/f2) pode não dar ou dar erro, dependendo de nessa altura P1 já tiver ou ainda não tiver executado link().

-- também, se P2 executar open() depois de P1 executar link() e nesse caso a chamada open em P2 já não dá erro, devolvendo bem um número canal fd2 para ler do ficheiro, o que pode acontecer a seguir? Como P2 vai agora ler desse canal, chamando read(), então, se P2 chamar read antes de P1 ter feito write, como o ficheiro nesse caso ainda está vazio, entao read vai devolver nr igual a 0 (leu 0 bytes) e não copiará nada para o buffer buf2.

Qual a conclusão a tirar? Para analisar o comportamento de programas (como o acima) executados por processos concorrentes, seria necessário estudar, caso a caso, todas as possíveis permutações na ordem de execução das instruções dos processos, pois esta ordem é imprevisível – e os programas só estarão correctos se não tiverem erros em nenhuma das permutações analisadas. Neste exemplo, as sequencias A e B têm poucas instrucoes o que nos permite fazer essa análise ``à mão´´, como antes se fez! Mas em geral, para programas com milhões... de instruções, vê-se que essa análise exaustiva não é praticável! Por isso teremos de estudar, no seguimento da matéria, formas de lidar com isto, na verdade estudando abordagens e técnicas de programação concorrente (porque envolvem mais do que um processo em execução).

A imprevisibilidade sucede, porque em regime de multiprogramação quando há um único CPU, como estudámos, o SO tenta alternar a execução de programas de forma gerir bem o tempo de utilização do CPU. (E se tivermos multiprogramação num computador com múltiplos CPU seria igualmente imprevísivel, pois depende do SO e do próprio hardware dos processadores e nesse caso P1 e P2 estariam a executar em simultâneo!). Para compreender melhor a que é devida esta imprevisibilidade, considere os três casos seguintes, de situações que ocorrem durante a execução de processos em regime de multiprogramação. Veja os três casos seguintes:

--(Caso1) Em particular, como estudámos nas aulas, se um processo fizer uma operação em que precisa de dados, quando os buffers donde deve ler, estiverem vazios, o SO, em vez de permitir que o processo fique em espera activa até que os dados viessem, vai suspender a execução do processo, passando ao estado Bloqueado, até que os dados venham. Nesse caso, o CPU fica disponível para executar outro processo, que será então seleccionado a partir de uma fila global de processos que estão prontos para execução (a fila de processos Prontos). – **Note que os momentos em que um processo se bloqueia nestes casos, são completamente imprevisíveis por um programador ou utilizador, pois são determinados segundo o ritmo a que um programa se executa, que também é imprevisível, e pelas estratégias/decisões internas dos programas do SO. Assim, os momentos em que ocorrem, podemos considerá-los aleatórios.**

----(Caso2) O SO também tenta evitar que um processo que tenha um perfil CPU-bound, venha a monopolizar o uso do CPU tirando a oportunidade de os outros processos executarem. Implementa por isso um mecanismo dito de Time-slice em que cada processo, ao ser activado do estado Pronto para o estado Executando, tem um temporizador activado com um intervalo de tempo máximo da ordem de 20 a 50 milisegundos – note 1 milisegundo = 1000 microsegundos = 1000000 nanosegundos e 1 nanosegundo é a ordem de grandeza do ciclo do relógio do processador, pois 1nanosegundo corresponde a uma frequencia de relógio de 1 GigaHertz ou seja 1000000000 ciclos por segundo – assim se vê que o Time-slice tem, do ponto de vista do CPU, uma duração muito grande. Quando o Time-slice de um processo executando expira, e o processo ainda não terminou ou não se bloqueou entretanto, o SO suspende a execução desse processo, voltando a colocá-lo na fila de processos Prontos, e dá a vez de execução a outro processo.
- **Note que os momentos em que um processo perde o controlo por expirar o Time-slice nestes casos, são completamente imprevisíveis por um programador ou utilizador, pois são determinados segundo o ritmo a que um programa se executa, que também é imprevisível, e pelas estratégias/decisões internas dos programas do SO. Assim, os momentos em que ocorrem, podemos considerá-los aleatórios.**

-- --(Caso3) A cada momento, um processo executando, pode ser interrompido por pedidos feitos pelo hardware das interfaces dos periféricos, para que o SO vá buscar/enviar bytes/comandos para essas interfaces, de forma a manter os periféricos em operação e também de forma a não perder dados. Para tratar esses pedidos, o hardware de controlo de interrupções garante que o estado de execução do programa que vai ser interrompido é salvaguardado (para mais tarde, quando o pedido de interrupção tiver sido tratado, poder retomar a execução do programa exactamente no ponto onde ia antes de ser interrompido -- o mínimo que o hardware salvaguarda é o valor do ProgramCounter e do registador de Flags de estado do CPU, que são em geral empilhados no stack). Depois de guardar esse estado mínimo, o hardware de interrupções muda o estado do CPU de modo Utililizador para Supervisor, desliga o atendimento de pedidos de interrupcao durante o tratamento deste pedido, e salta para uma rotina de servico para tratamento do pedido. Esta rotina, que é em geral muito simples e rápida, vai geralmente buscar ou enviar bytes às interfaces dos perifericos e depois retorna à execucao do programa interrompido (repondo o seu estado antes empilhado no stack e voltando o CPU a modo Utilizador e estar atento a novas interrupcoes). – **Note que os momentos em que um processo é interrompido nestes casos, são completamente imprevisíveis por um programador ou utilizador, pois são determinados segundo o ritmo a que um programa se executa, bem como pelo ritmo a que ocorrem os pedidos de interrupção dos periféricos, que também é imprevisível, e pelas estratégias/decisões internas dos programas do SO. Assim, os momentos em que ocorrem, podemos considerá-los aleatórios.**