Tipo de Prova: sem consulta
Duração: 2 horas
Cotação máxima: 20 valores (10 da nota final!)
Estrutura da prova: Parte I (escolha múltipla, 25%); Parte II (mais convencional, 50%); Parte III (predominantemente de aplicação, 25%)

Soluções do **Exame Normal da Época Normal**
26.Janeiro.2009

PARTE I: Escolha múltipla [5 val.]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B	A	C	C	B	C	A	C	C	C	C	B	B	B	A

PARTE II: Mais convencional [10 val.]

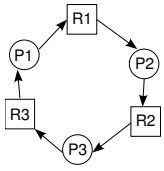
1. Indicar ao sistema uma função que será invocada quando o processo terminar normalmente (a função pode ser usada para "arrumar a casa").
Não é necessário estender as funcionalidades de `atexit()` a terminações súbitas devido à recepção de sinais, pois tal funcionalidade pode ser obtida pela instalação de rotinas de tratamento de sinais. (Uma excepção, seria, talvez, para o caso de recepção de um sinal "não tratável"...)

2. Um *thread* é uma actividade que se desenrola no contexto de um processo, sendo esta a entidade reconhecida pelo sistema operativo para a alocação de recursos. Assim: um *thread* usa o espaço de endereçamento do processo; tem, desde a sua criação, uma rotina associada; pode terminar sem com isso obrigar à terminação do processo.

3. Tal chamada permite desactivar a retenção automática de dados dirigidos à saída normalizada. Na prática, isto implica que as mensagens escritas com `printf()`, por exemplo, são imediatamente colocadas na saída; se o programa estourar a dado passo da sua execução, toda a informação que foi mandada imprimir, foi mesmo impressa, não tendo sido perdida pelo facto de estar retida num local de memória entretanto invalidado pelo sistema. A observação de tal informação constitui um importante auxiliar à depuração do programa.

4. Sendo uma construção associada a uma linguagem de programação (ver o caso de Java, em que métodos da uma classe podem ser classificados como *synchronized*) uma zona crítica pode ser facilmente conseguida pela sua colocação num monitor! Assim, o acesso exclusivo à zona é controlado pelo código introduzido automaticamente pelo compilador/interpretador da linguagem. Sem o concurso do programador!

5. Ver o exemplo da figura ao lado: o encravamento dá-se porque cada processo detém um recurso e quer outro que está na posse de um processo diferente.



a) Os círculos são processos e os quadrados recursos; de um recurso afecto a um processo vai uma seta até ao processo; de um processo que pretende tomar um recurso vai uma seta até ao recurso.
b) Não. Podem estar em actividade (estados de prontidão ou execução), tentando obter acesso ao recurso que pretendem. Tal situação designa-se, mais especificamente, por *livelock*.
c) Desafectando um dos recursos ao processo que o detém; se tal não for possível, terminando à força um dos processos (e recomeçando-o mais tarde).

6. Uma espera eterna do pai acontece no caso de o novo processo, filho, começar a executar primeiro após o `fork()` e enviar o sinal ao pai; quando, mais tarde, o pai invocar `pause()`, já o sinal "passou"!...

7. Um sistema com segmentação pode ser utilizado pelo programador para especificar zonas de memória logicamente distintas e que possam, por exemplo, ser protegidas pelo sistema operativo de maneira diferente (e.g. só leitura). Por outro lado, como os segmentos não têm, em geral, tamanhos iguais, vão complicar a gestão de memória física disponível, originando, por exemplo, problemas de fragmentação externa.
Um sistema misto, com segmentos paginados, resolve o problema apontado, mas complica o sistema operativo e o equipamento de suporte, uma vez que exige a gestão de ambas as técnicas. (Todavia, é bastante utilizado, tendo, por

exemplo, a Intel dado-lhe suporte nos seus processadores a partir da série *i386*.)

8. Note-se que nem toda a informação da tabela está, em geral, disponível para o sistema operativo que implementa um dos possíveis algoritmos de substituição de página de memória virtual!

a) A página mais antiga a ter sido referenciada, directa ou indirectamente, pelo programa em execução, foi a 87 que está colocada na moldura 3. Será substituída a seguir.

b) LRU é um algoritmo que se aproxima bastante do ideal (que substituiria a página demorasse mais tempo a vir a ser precisa outra vez) porque se verifica que na maioria dos programas em execução, uma página que não é referenciada há muito tempo, também só virá a ser precisa daqui a muito tempo!... (Note-se que isto é contra a teoria das probabilidades para acontecimentos aleatórios, mas o facto é que, segundo parece, os programas não são feitos "à sorte"!... :-)). O inconveniente de LRU é que é dispendioso de concretizar, pois exige um registo multi-bit de tempos para cada moldura e, normalmente, a manutenção de uma lista ligada das molduras, ordenada a cada referência, da mais antiga para a mais recente.

9. A camada "*User processes*" efectua o pedido de entrada/saída e recebe o resultado da resposta; a camada "*Device-independent*" efectua operações razoavelmente independentes do dispositivo, tal como tratamento de nomes (e.g. «em que disco está o ficheiro `/tmp/4967747d6f2a2`»?); a "*Device drivers*" conhece os registos específicos do dispositivo a aceder e coloca neles os valores apropriados ao pedido; irá também verificar o resultado da resposta; a "*Interrupt handlers*" é que recebe a indicação de que o dispositivo completou o pedido e disso informa o processador; finalmente a camada "*Hardware*" corresponde ao dispositivo a aceder propriamente dito e inclui o seu equipamento electrónico e o *software* que nele possa estar contido.

10. Porque a informação relativa a directorias em Unix (e na maioria dos sistemas operativos correntes) é estruturada de maneira que só o sistema operativo conhece, não podendo um programador aceder-lhe directamente. Repare-se que a chamada `read()`, para ficheiros normais é a "contrária" à chamada `write()` e, em ambos os casos, a informação passada é uma mera sequência de *bytes* cuja construção e interpretação é conhecida e feita pelo programador.

PARTE III: Predominantemente de aplicação [5 val.]

1.

a) `fork: Resource temporarily unavailable`
`exame-fifo` (e outro conteúdo do directório `/tmp`)
(note que «`ls terminou`» ***não*** é impresso!)

b) Sim, é possível, pois a FIFO foi criada com permissão de leitura e escrita para todos. Para que apenas os processos do criador da FIFO nela possa ler e escrever basta mudar a linha 4 para `mkfifo(FIFON, 0600);`.

c) O `read()` numa FIFO devolve 0 quando (`man 7 pipe`): «`If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read(2) from the pipe will see end-of-file (read(2) will return 0)`».
Portanto, no programa acima, a mensagem aparece quando o processo filho tenta ler da FIFO mas o pai já fez o `close()`. Ainda segundo o manual (`man 7 fifo`): «`The FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, opening the FIFO blocks until the other end is opened also.`». Portanto, é necessário que o processo pai e o processo filho passem pelo (sincronizem através do) `open()`; o processo filho é então suspenso pelo *kernel*, e o pai faz o `write()` e o `close()`; quando o *kernel* passa o processo filho novamente para o estado *run* e ele tenta fazer o `read()`, este retorna 0.

d) Esta pergunta foi mal colocada, pelo que foi cotada a 100% independentemente da resposta.
Aparentemente: O erro surge pois o `read()` ou `write()` falham, devolvendo um valor negativo. Neste caso o erro é «`EBADF fd is not a valid file descriptor`», o que significa que a variável `fd` devolvida pelo `open()` é um descritor inválido.
Um cenário seria: um dos processos faz o *open*, o *read* (ou *write*) e a seguir o *unlink*. O outro processo fica bloqueado no *open* que devolve erro (porque o FIFO já nao existe), e ao tentar fazer *write* (ou *read*) este falha. O modo de corrigir a situação é verificar ***sempre*** o valor devolvido pelo `open()` e tomar medidas adequadas em caso de erro.
Outro cenário: após o *open* ter sucesso em ambos os processos, um deles é suspenso antes do *read* ou *write*, enquanto o outro processo prossegue e faz o *unlink*; quando o processo suspenso for recomeçado, o descritor obtido refere-se a um ficheiro já não existente e o *read* ou *write* falham. Esta situação não seria passível de

detecção no *open*.
Na realidade: como visto na alínea anterior, qualquer um dos processos fica bloqueado no `open()` até ambos o executarem e terem sucesso (sincronizarem), nas condições do enunciado. Quando depois qualquer um dos processos avança e faz `unlink()` da FIFO, eliminando o ficheiro, o *kernel* **garante** que os ficheiros (ou FIFO) **já abertos** por processos mantêm a sua existência (só para esses processos): «`unlink() deletes a name from the filesystem... If any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed. If the name referred to is a fifo, then the name for it is removed but processes which have the object open may continue to use it.`»
Também o manual do `open()` diz: «`A call to open() creates a new open file description, an entry in the system-wide table of open files... A file descriptor is a reference to one of these entries; this reference is unaffected if pathname is subsequently removed or modified to refer to a different file.`»
Portanto, a mensagem de erro nunca ocorre, pois apesar de o FIFO poder não existir no sistema de ficheiros, existe como estrutura interna do *kernel*, e o *read*/*write* não falham com o erro EBADF.
O enunciado devia ter sido: «*Descomentando a linha 27, podem por vezes ocorrer as mensagens....?*»
Pode-se testar facilmente: abrir uma consola e escrever, "`mkfifo popo`" e depois "`cat > popo`". Mudar para outra consola e, no mesmo directório, fazer "`cat < popo`". Voltando à 1ª consola, tudo o que for escrito, ao ser terminado com `<enter>` aparece na 2ª consola. Abrir uma 3ª consola e, no mesmo directório, fazer "`rm popo`"; para ter a certeza, fazer "`ls popo`"; mudar então para a 1ª consola e escrever mais linhas, verificando que continuam a aparecer na 2ª consola, mesmo não existindo o FIFO no sistema de ficheiros.

e)
```
pid_t pid; /* make it global */

void killer(int sig) {
    printf("Bored... 30 secs elapsed and no response... shooting it!\n");
    kill(pid, SIGTERM);
}

int main(int argc, char *argv[]) {
    ...
    if (pid > 0) {
        int st;
        struct sigaction sa;
        sigset_t smask;
        sigemptyset(&smask);
        sa.sa_handler = killer;
        sa.sa_mask = smask;
        sa.sa_flags = 0;

        if(sigaction(SIGALRM, &sa, NULL) == -1) { /* install timeout handler */
            perror ("sigaction"); exit (-1);
        }

        ... /* open/write/close */

        alarm(30);  /* trigger timeout, will generate SIGALRM in 30 sec*/
        if (wait(&st) < 0) { /* block and wait for child return value */
            if (errno == EINTR)
                printf("Child was killed, return value is meaningless\n”);
            else
                perror("wait");
        } else
            printf("Child finished with code %d\n", WEXITSTATUS(st));

    } else if(pid == 0) {
        ...
        sleep(35); /* simulate a lengthly calculation */
        return n;
    } ...
```

2.

a) É O *array* `arg` é necessário. Tal como usado na linha 24, cada *thread* receberia um apontador para uma variável que está a ser constantemente modificada. Quando a *thread* finalmente acedesse à variável iria obter o seu valor no instante do acesso e não no instante da criação da *thread*. Podia acontecer que várias *threads* recebessem o mesmo argumento, e alguns valores do argumento não fossem usados por nenhuma *thread.* Não nos esqueçamos

que a sequência de execução das *threads* e o tempo durante o qual estão em execução não é previsível.

b) O manual diz que «`The pthread_detach() function shall indicate... that storage for the thread can be reclaimed when that thread terminates`». Assim, se uma *thread* não for *detached*, a memória que utilizar não será libertada quando ela terminar, a menos que se faça `pthread_join()`. Como o programa não o faz, isso levaria à exaustão da memória disponível ao processo e à impossibilidade de criar mais *threads.*
A parcela de memória mais importante, mesmo para *threads* simples, é o espaço de memória associado à sua *stack*, que em Linux é normalmente de 8 MB por *thread.* A memória virtual normalmente associada em linux a cada processo é de 2 a 3 GB, e de facto 8MB*382≈3GB. A partir de 382 *threads not-detached* deixa de haver memória no processo para mais *threads.* O seguinte segmento de programa, sem fazer *detach* nem *join*, permite criar 3200 *threads,* cada uma com 1MB de *stack*, antes de falhar:

```
size_t stack_size = 1000000;
pthread_attr_t attr; pthread_attr_init(&attr);
pthread_attr_setstacksize(&attr, stack_size);
for (i=0; i<10000; i++)
    if(pthread_create(&tid, &attr, start_routine, NULL) {
        perror("create"); printf("Launched %d threads.\n", i); exit(1);
    }
```

c) Sim, se uma *thread* está *detached* não se pode esperar por ela, pois não têm o atributo *joinable* activo. Se o segundo argumento de `pthread_create()` for `NULL`, por omissão , são criadas `threads` *joinable.* A alternativa a `pthread_detach()` é criar desde logo a *thread* no estado *detached*:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, start_routine, arg);
```

Além do anteriormente dito, seria impossível obter o valor devolvido por uma *thread detached*. Uma *thread detached* vê, ao terminar, toda a sua memória libertada automaticamente e, portanto, o apontador devolvido pelo *return* da *thread* ou por `pthread_exit()`, e que seria usado em `pthread_join()`, será necessariamente inválido. Mas esta não é a razão principal, pois poder-se-ia fazer sempre `pthread_join(tid, NULL)`, pretendendo apenas esperar pela terminação da *thread* sem querer o valor devolvido por ela.

d) Não, pois cada *thread* `filler` acede a uma posição pré-determinada e distinta do *array*, não havendo possibilidade de conflito; note que a *thread* `watcher` só ficará activa quando todas as *thread* `filler` terminarem.

e)
```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int cnt = 0;

void *filler_aux(void *arg) {
  filler(arg);
  pthread_mutex_lock(&mut);
  cnt++;
  if (cnt == NTHREADS)         /* only wakeup watcher when fillers are done */
      pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mut);
  return NULL;
}

void *watcher(void *arg) {
  int i, tmp;
  pthread_mutex_lock(&mut);
  while (cnt < NTHREADS) /* But “man” says: Spurious wakeups from the
                pthread_cond_wait() functions may occur (and I'm a Yes man:-) */
      pthread_cond_wait(&cond, &mut);
  pthread_mutex_unlock(&mut);
  ...
}.
```

JMMC, JFSC