3 Processos: Modelo Computacional

Ex. 1 - Considere o seguinte pseudo-código de um programa multi-tarefa com pthreads (por simplicidade, omitem-se verificações de erros):

```
1     int main(int argc, char **argv) {
2         pthread_t tid;
3         pthread_create(&tid, NULL, t2_start_routine, NULL);
4         printf("A\n");
5         pthread_join(tid, NULL);
6         printf("B\n");
7         return 0;
8     }
9
10    void *t2_start_routine(void *arg) {
11        printf("C\n");}
12        sleep(10); /* 10 segundos */
13        return NULL;
14    }
```

a) Que saídas são possíveis observar quando executamos este programa? Justifique. Assuma apenas execuções em que nenhuma das funções chamadas devolvem erro.

Tanto pode acontecer "A\nC\nB\n" como "C\nA\nB\n". A partir da linha em que a nova tarefa é criada, ambas as tarefas executam-se concorrentemente, logo não há garantia de ordem entre as execuções de cada uma, o que explica porque os printf das linhas 4 e 11 podem acontecer em qualquer ordem. Na linha 5, a tarefa original espera até a segunda tarefa terminar (linha 13), logo o printf da linha 6 acontecerá necessariamente depois dos outros printf (assumindo execução sem falhas).

b) Ao longo da execução do programa existirão diferentes conjuntos de tarefas em execução. Indique como evolui o conjunto de tarefas, relacionando com as linhas do programa acima.

Uma tarefa até à linha 3, duas tarefas até à linha 13, uma tarefa a partir desse momento (assumindo execução sem falhas).

c) Caso a tarefa t2 queira devolver uma cadeia de caracteres (char*) à tarefa original para esta a imprimir com printf, como deveria o programa ser alterado?

Na função t2_start_routine:

```
1     char *s = ...;
2     ...
3     return s;
```

Na função main:

```
1     char *s;
2     ...
3     pthread_join(tid, &s);
4     printf("%s", s);
```

Ex. 2 - Considere um dado programa, com vários fios de execução, que tem 3 versões que se executam num computador uni-processador com sistema operativo do tipo Unix e cuja funcionalidade não implica nenhuma operação de I/O:

- versão V1 em que cada fio de execução é suportado por um processo,
- versão V2 em que cada fio de execução é suportado por uma tarefa real, e
- versão V3 em que cada fio de execução é suportado por uma pseudo-tarefa.

Note que as 3 versões têm exatamente a mesma funcionalidade.

a) Como compara as versões V1 e V2 no que respeita à robustez do programa? Na sua resposta considere, por exemplo, que um dos fios de execução faz uma divisão por zero.

A versão V1 é mais robusta, uma vez que apenas o processo onde ocorre o erro é terminado, continuando as restantes tarefas a executar-se. Na versão V2 o processo (que inclui todas as tarefas) é terminado. No caso de uma divisão por zero, em V1, apenas o fio de execução associado ao processo onde ocorre a divisão por zero termina (e todos os outros fios de execução continuam a sua execução normal) enquanto que em V2, todos os fios de execução são terminados.

b) Como compara as versões V1 e V3 no que respeita à robustez do programa? Na sua resposta considere, por exemplo, que um dos fios de execução faz uma divisão por zero.

Resposta idêntica à anterior.

c) Como compara as versões V1 e V2 no que respeita à rapidez de execução do programa?

V1 será mais lento, pois a criação de processos e comutação entre processos é mais lenta que a criação/comutação entre tarefas, dado que os processos não partilham o mesmo contexto.

d) Como compara as versões V1 e V3 no que respeita à rapidez de execução do programa?

Resposta idêntica à anterior.

e) Considere agora que o programa é executado numa máquina com dois processadores. Compare o desempenho das versões V2 e V3.

A versão V2 será mais rápida pois diferentes tarefas poderão executar-se concorrentemente nos dois processadores. Na versão V3, todas as pseudo-tarefas irão partilhar um único processador.

Sugestão: Rever Secção 3.4.2 - Modelo Multi-tarefa

Ex. 3 - Considere um sistema operativo do tipo Unix no qual um processo executa o seguinte programa que se encontra num ficheiro "t.c"; "t.c" é compilado gerando um ficheiro executável "t.exe". Por simplicidade, omitem-se verificações de erros.

```
1     #include <stdio.h>
2     #include <stdlib.h>
3     main() {
4         int pid, pid_filho, status;
5         printf ("\nantes␣do␣fork␣pid=%d\n", getpid());
6         pid = fork();
7         if (pid == 0) {
```

```
8               printf ("pai=%d␣e␣filho=%d\n", getppid(), getpid());
9               execv ("t.exe",NULL);
10          }
11          else if (pid != -1){
12              pid_filho = wait(&status);
13              printf ("depois␣do␣wait␣feito␣pelo␣pid=%d␣obtendo␣pid_filho
                    =%d\n", getpid(), pid_filho);
14              exit (0);
15          }
16          else {
17              printf ("erro␣no␣fork\n");
18              exit (-1);
19          }
20      }
```

a) Quantos processos são criados?

Serão criados continuamente processos, de forma recursiva, até que se esgotem os recursos da máquina ou que os limites de utilização de recursos pelo utilizador sejam atingidos.

b) Assuma que o fork nunca retorna erro. A instrução depois do "wait" é executada?

Assumindo uma máquina e utilizador com recursos infinitos (logo o programa acima é capaz de criar um número infinito de processos filho), nenhum processo filho chega a terminar (assumindo também que a chamada "exec" nunca falha). Consequentemente, a função "wait" nunca retornará, logo alinha seguinte nunca será executada.

c) Assuma que o fork retorna erro na 3ª vez que é chamado. Considere que inicia a execução do programa "t.exe". Indique o seu output continuando o que se indica (antes do fork pid=2204).

antes do fork pid=2204
pai=2204 e filho=2205
antes do fork pid=2205
pai=2205 e filho=2206
antes do fork pid=2206 erro no fork
depois do wait feito pelo pid=2205 e obtendo pid_filho=2206
depois do wait feito pelo pid=2204 e obtendo pid_filho=2205
Os valores dos pids do processos apresentados podem ser outros pois isso depende do SO; no entanto, note que os valores em causa devem ser coerentes com a hierarquia dos processo em causa.

Sugestão: Rever Secção 3.5.2 - Operações sobre Processos

Ex. 4 - É possível implementar o conceito de pseudo-paralelismo sem recorrer ao núcleo do sistema operativo? Justifique.

Sim, é possível implementar o conceito de pseudo-paralelismo através da utilização de co-rotinas ou pseudo-tarefas. A gestão das pseudo-tarefas é feita explicitamente pelo programador usando bibliotecas que se executam no espaço do utilizador, i.e., sem recorrer ao núcleo do sistema operativo. Em particular, a comutação de tarefas poderá ser feita: i) explicitamente, o que implica que o programador invoque a função de comutação de tarefa "yield", ou ii) implicitamente através da utilização do mecanismo de exceções com base num intervalo de tempo que, quando expira, gera um signal que é tratado por uma rotina que efetuará a comutação de tarefas.

Sugestão: Rever Secção 3.4.2.2 - Conceito de tarefa

Ex. 5 - Considere o seguinte excerto de um programa Unix (por simplicidade, omitem-se verificações de erros):

```
1     int a; /* variavel global */
2     main() {
3         a = 0;
4         if (fork() == 0) {
5             a++;
6             printf("\%d", a);
7             exit(0);
8         }
9         else {
10            pid = wait(\&estado);
11        }
12        printf("\%d", a);
13    }
```

a) Qual a saída deste programa? Justifique.

Saída: '10'. O primeiro algarismo ('1') é impresso após a criação do processo filho e depois do incremento unitário à variável 'a', que depois é impressa (linha 6). O segundo algarismo ('0') é impresso pelo processo pai depois do processo filho terminar (linha 12). É de notar que os processos não partilham o mesmo espaço de endereçamento, razão pela qual a variável 'a' possui valores distintos nos dois processos.

b) Modifique-o de modo a obter outra sequência. Por simplicidade, omita verificações de erros. Justifique.

Várias respostas possíveis. Possível resposta:

```
1     int a; /* variavel global */
2     main() {
3         a = 0;
4         if (fork() == 0) {
5             a++;
6             printf("\%d", a);
7             /* exit(0); */
8         }
9         else {
10            pid = wait(\&estado);
11        }
12        printf("\%d", a);
13    }
```

Saída: '110'. Os primeiros dois '1's são impressos pelo filho (linhas 6 e 12). O '0' é impresso pelo pai (linha 12). O valor da variável 'a' no processo pais é zero pois apenas o filho incrementa a variável (linha 5). Uma vez que os processos não partilham o mesmo espaço de endereçamento, um incremento de uma variável no filho não tem qualquer efeito no processo do pai.

Sugestão: Rever Secção 3.5.2 - Operações sobre Processos

Ex. 6 - Para cada uma das seguintes afirmações, indique se esta pode ou não ser um motivo válido para

a utilização de múltiplas tarefas reais (tarefas-núcleo) num programa que corre num sistema operativo Unix, e explique porquê:

a) Ter múltiplos fios de execução com espaços de endereçamento separados no mesmo programa.

Não; ao utilizar tarefas reais, o espaço de endereçamento é comum a todas as tarefas do mesmo processo.

b) Poder sobrepor a execução de instruções de E/S com a utilização do CPU por outras partes do mesmo programa.

Sim, ao utilizar tarefas reais, enquanto uma ou mais tarefas podem estar a tratar eventos de E/S, outras poderão estar a utilizar CPU.

c) Executar dois programas diferentes (correspondentes a diferentes ficheiros executáveis) em paralelo.

Não; tarefas reais não são necessárias para atingir paralelismo entre dois processos diferentes. As tarefas reais podem apenas trazer paralelismo dentro do mesmo processo (i.e., vários fios de execução em paralelo).

Sugestão: Rever Secção 3.4 - Modelos de Programação de Processos

Ex. 7 - Considere o seguinte excerto de um programa que usa signals em Unix (por simplicidade, omitem-se verificações de erros):

```
1     void trata_signal(int num) {
2         signal(SIGINT, trata_signal);
3         printf("Operacao␣negada.");
4     }
5
6     int main(void) {
7         if (signal(SIGINT, trata_signal) == SIG_ERR)
8             {printf("Erro␣no␣signal");}
9         for (;;) {sleep(10);}
10        exit(0);
11    }
```

a) Explique o comportamento deste programa.

O programa começa por instalar uma rotina de tratamento para o sinal 'SIGINT' usando a chamada de sistema "signal" e fazendo o teste devido para verificar o retorno da chamada de sistema. De seguida, o programa entra num ciclo "for" sem fim onde, a cada iteração, o programa suspende a sua execução durante 10 segundos (utilizando a chamada de sistema "sleep"). A rotina de tratamento do sinal "SIGINT", se chamada, re-instala o sinal e imprime uma mensagem para a output. É de notar que o sinal "SIGINT" pode ser gerado através da combinação das teclas CTRL+C.

b) Identifique todas as funções do sistema operativo e da biblioteca stdio que são utilizadas no programa e explique o que faz cada uma.

"printf" - imprime carateres para o output do processo; "signal" - instala rotinas de tratamento de sinais; "sleep" - suspende a execução da tarefa atual durante um período de tempo; "exit" - termina a execução do processo.

c) De que forma um processo que corra este programa pode ser terminado? Justifique.

O processo pode ser terminado enviando um sinal cujo efeito sobre o processo é terminá-lo. Por exemplo, é possível utilizar a chamada de sistema "kill" com o sinal "SIGTERM" para terminar este processo.

Sugestão: Rever Secção 3.5.3 - Signals

Ex. 8 - Considere a chamada de sistema "kill" em Unix.

a) Explique o que faz esta função.

A chamada de sistema "kill" tem como propósito o envio de sinais a processos.

b) Esta função necessita de efetuar algumas validações de segurança. Explique quais e qual o motivo da sua existência.

Uma vez que a chamada de sistema "kill" pode interferir com outros processos, é importante garantir que, por exemplo, um utilizador sem privilégios de super-utilizador não consiga interferir com processos do sistema (que estão associados ao super-utilizador do sistema) ou com processos de outros utilizadores. Desta forma, usando a função de sistema "kill", apenas é possível enviar sinais a processos do mesmo UID. Apenas processos com privilégios de super-utilizador podem ultrapassar esta proteção.

c) O nome desta primitiva é enganador. Explique porquê.

É enganador pois a função não mata o processo, apenas envia sinais. A possível motivação para o nome resulta da implementação por omissão do tratamento da maioria dos sinais, que resulta no término do processo.

Sugestão: Rever Secção 3.5.3.4 - Operações Associadas aos Signals

Ex. 9 - Considere o seguinte código multi-tarefa (por simplicidade, omitem-se verificações de erros):

```
1     int i;
2     int main(int argc, char **argv) {
3         if(thread_create(t2_start_routine, (void*)0) == NULL) {exit(1);}
4         thread_yield(); /* comutacao de tarefa */
5         for (i = 0; i < MAX; i++) {printf("t1\n");}
6         return 0;
7     }
8     void *t2_start_routine(void *arg) {
9         for(i = 0; i< MAX; i++) {printf("t2\n");}
10        sleep(10); /* 10 segundos */
11        return 0;
12    }
```

Suponha que as rotinas "thread_create" e "thread_yield" são implementadas por uma biblioteca de co-rotinas (pseudo-tarefas).

a) Qual é a saída deste programa? Justifique.

Saída: 'MAX' linhas com 't2' seguidas de 'MAX' linhas com 't1'. Uma vez que este código usa pseudo-tarefas, cada tarefa vai executar-se (de acordo com o código apresentado) até que a outra chame a função "thread_yield" ou termine. Assim sendo, a thread que executa a rotina 't2_start_routine' vai executar-se até terminar (daí as linhas com 't2' serem impressas todas seguidas). A outra thread (que invocou a função "thread_yield") apenas imprime linhas com 't1' após a outra thread terminar.

b) A resposta à alínea anterior manter-se-ia se mudasse para tarefas reais? Justifique.

Não. Com tarefas reais, é possível que as linhas escritas por cada tarefa ficassem alternadas uma vez que é o núcleo do sistema operativo a decidir que tarefa corre em cada instante. Deste modo, como já não é o programador a decidir o escalonamento, ambas as tarefas vão executar-se concorrentemente.

c) Considerando ainda o código apresentado acima, consegue determinar o tempo mínimo que decorre entre a saída das diferentes tarefas? Qual é esse tempo mínimo? Justifique.

10 segundos é o tempo mínimo pois a tarefa que executa a função 't2_start_routine' executa um "sleep" antes de terminar. É de notar que a tarefa 't2' está a executar-se porque a tarefa 't1' invocou a função "thread_yield", que retira 't1' de execução até que tarefa 't2' termine.

d) A resposta à alínea anterior manter-se-ia se mudasse para tarefas reais? Justifique.

Não, com tarefas reais, a saída de ambas as tarefas pode sair alternada pois a chamada a "thread_yield" apenas faz com que a thread perca o tempo de CPU que lhe foi atribuído no último escalonamento. Desta forma, é possível que venha a ser escalonada novamente antes da thread 't2' terminar.

Sugestão: Rever Secção 3.5.4 - Tarefas-Interface POSIX

Ex. 10 - Considere o código do seguinte programa Unix (por simplicidade, omitem-se verificações de erros):

```
1   #include <stdio.h>
2   int a;
3   fn() {
4       a++;
5       printf("X␣%d\n", a);
6       exit(0);
7   }
8
9   main(int argc, char **argv) {
10      int val;
11      a = 0;
12      printf("Y␣%d\n", a);
13      if (fork() == 0) {fn();}
14      wait(&val);
15      printf("Z␣%d\n", a);
16  }
```

a) Qual a saída deste programa? Se houver mais do que uma possibilidade, indique todas as possíveis saídas.

Saída possível 1:
Y 0
X 1
Z 0

Saída possível 2 (se o "fork" falhar):
Y 0
Z 0

b) Escreva um novo programa que seja equivalente (no sentido de executar a mesma sequência de código, apesar de não ter necessariamente a mesma saída), exceto no seguinte aspeto: na linha 13 deve lançar uma nova tarefa em vez de um novo processo. Deve utilizar a interface POSIX para todas as funções relacionadas com as tarefas. Tenha o cuidado de atualizar as restantes funções do sistema operativo que são invocadas neste código em conformidade. Por simplicidade, omita verificações de erros.

```
1   #include <stdio.h>
2   int a;
3   void *fn(void *arg) {
4       a++;
5       printf("X␣%d\n", a);
6       return 0;
7   }
8
9   main(int argc, char **argv) {
10      pthread_t tid;
11      a = 0;
12      printf("Y␣%d\n", a);
13      pthread_create(&tid, NULL, fn, NULL);
14      pthread_join(tid, NULL);
15      printf("Z␣%d\n", a);
16  }
```

c) Qual a saída do novo programa?

Saída possível 1:
Y 0
X 1
Z 1

Saída possível 2 (se o "pthread_create" falhar):
Y 0
Z 0

d) Ordene em termos de velocidade de comutação, da mais rápida para a mais lenta, as seguintes possibilidades para a execução deste programa:

Usando tarefas núcleo; Usando pseudo-tarefas; Usando processos;

Pseudo-tarefas, tarefas núcleo, processos.

e) Para a resposta anterior, explique a razão principal para a escolha relativa entre as tarefas núcleo e as pseudo-tarefas.

A principal razão prende-se com o facto da comutação de tarefas reais implicar mais custo do que a comutação de pseudo-tarefas pois no primeiro caso é preciso passar de modo de utilizador para modo núcleo e vice versa enquanto que no segundo caso a comutação apenas em modo de utilizador.

Sugestão: Rever Secção 3.4 - Modelos de Programação de Processos

4 Gestor de Processos

5 Sincronização: Secções Críticas

Ex. 1 No âmbito das soluções que asseguram a exclusão mútua, diga em que consiste a noção de míngua (starvation). Como se devem comportar as soluções corretas no que diz respeito à míngua?

A noção de míngua (ou starvation) refere-se à situação onde uma ou mais tarefas, que pretendem aceder a um recurso, não o conseguem, sistematicamente sendo ultrapassadas por outras tarefas que lhe acedem (mesmo que estejam há menos tempo à espera de entrar). Uma solução correta relativamente à míngua deve garantir que todas as tarefas têm igual oportunidade de acesso ao recurso em causa, não havendo preferências ou prioridades na seleção de tarefas a entrar na secção crítica.

Sugestão: Rever Secção 5.2 - Requisitos de uma Secção Crítica.

Ex. 2 Diga o que entende por espera ativa numa solução para o problema da secção crítica.

Espera ativa numa solução para o problema da secção crítica significa que um ou mais fios de execução estão continuamente a testar uma condição de entrada na secção crítica e essa condição só se alterará se o fio de execução deixar de efetuar esse mesmo teste. A utilização de espera ativa é muito comum em soluções algorítmicas para exclusão mútua (e é uma desvantagem clara relativamente a soluções com base em objetos do sistema operativo que não ocupam o processador de forma inútil enquanto a condição de espera não varia).

Sugestão: Rever Secção 5.3 - Exclusão Mútua Algorítmica.

Ex. 3 Considere um sistema operativo em que, durante a sequência de teste e modificação de uma variável partilhada (trinco), a exclusão mútua é conseguida através da inibição das interrupções. Diga qual a desvantagem desta solução.

Existem várias desvantagens:

não funciona em multi-processadores; só funciona dentro do núcleo; só é aceitável com secções críticas muito pequenas; torna o sistema vulnerável a bugs dentro da secção crítica.

Sugestão: Rever Secção 5.4.1 - Inibição de Interrupções.

Ex. 4 Para resolver os problemas da secção crítica há três tipos de solução possíveis, que diferem consideravelmente nos recursos que implicam, no desempenho e na sua facilidade de utilização no modelo computacional. Diga quais são e apresente as vantagens/desvantagens comparativas.

Secções críticas podem der implementadas de três formas diferentes: 1. algoritmicamente. Esta solução apenas usa leituras e escritas de varáveis; no entanto, é difícil/complexa de usar na criação de programas e possui limitações relativamente ao número de tarefas que suporta e, mais grave ainda, implica a utilização de espera ativa; 2. baseada no hardware, que pode ser feita via inibição de interrupções ou usando instruções especiais. A primeira apenas pode ser utilizada em modo núcleo, em pequenos trechos de código e em sistemas com um único processador. A segunda pode ser usada com multi-processadores e pode ser usada em modo de utilizador. No entanto, requer instruções especiais (do tipo test-and-set) e requer que o processo fique em espera ativa (testando a variável do trinco); 3. utilizando objetos de sincronização (trincos e semáforos) suportados pelo núcleo. Esta solução combina as vantagens da solução anterior e permite evitar a espera ativa.

Sugestão: Rever Secções 5.3 (Exclusão Mútua Algorítmica), 5.4 (Exclusão Mútua Baseada no Hardware), e 5.5 (Exclusão Mútua com Objetos de Sincronização).

6 Programação Concorrente [com memória partilhada]

7 Mecanismos de Gestão de Memória

8 Algoritmos de Gestão de Memória

9 Sistemas de Ficheiros

10 Comunicação entre Processos

Ex. 1 Considere as duas implementações de mecanismos de comunicação entre processos: memória partilhada, objeto de comunicação do sistema.

a) Compare-as tendo em conta a complexidade da programação da sincronização. Justifique sucintamente.

A comunicação através de memória partilhada requer uma implementação mais complexa pois implica uma sincronização explícita por parte do programador. Isso não acontece com a utilização de objetos de comunicação, pois a sincronização é implicitamente tratada pelo núcleo.

b) Compare-as em termos de eficiência. Justifique sucintamente.

A comunicação via objeto de sistema é menos eficiente pois requer a cópia da mensagem pelo menos duas vezes; uma entre a memória do emissor e o buffer mantido no núcleo, e outra entre o buffer do núcleo e a memória do receptor. Utilizando memória partilhada é possível evitar cópia para o buffer no núcleo, resultando num envio mais eficiente.

Sugestão: Rever Secção 10.1.4 - Implementação do Canal de Comunicação

Ex. 2 No Linux existem mecanismos de comunicação entre processos que são identificados pelos programadores por nomes/endereços pertencentes a diferentes espaços de nomes. Dê exemplos.

1. Pipes com nome e sockets Unix: nome de ficheiro identifica o pipe. 2. Sockets Internet: endereço IP e porto. (entre outros)

Sugestão: Rever Secção 10.3 - Comunicação entre Processos em Linux

Ex. 3 Considere o seguinte comando shell do Unix resultante do encadeamento de comandos elementares:

```
ls -l | sort > saida
```

a) Descreva o resultado do comando;

O comando (ls seguido de sort) começa por listar os ficheiros da diretoria atual (primeiro processo, comando ls); o texto dessa listagem é, por sua vez, ordenado (segundo processo, comando sort); e finalmente escrito no ficheiro 'saida'.

b) Explique que mecanismos possibilitam este encadeamento de comandos;

Dois mecanismos principais são usados neste tipo de composições de comandos: pipes e redirecionamento de saídas. Neste caso concreto, existe um pipe sem nome utilizado para estabelecer a comunicação entre o processo que executa o comando de listagem 'ls' e o processo que executa o comando de ordenação "sort'. Usando um pipe, o output do primeiro é usado como input no segundo. O segundo mecanismo é usado em três momentos: i) para redirecionar o output da listagem para o pipe, ii) para redirecionar o input da ordenação para o pipe, e para iii) redirecionar o output da ordenação para um ficheiro.

c) Descreva sucintamente os principais passos do código que a shell executa para efetuar as operações deste comando encadeado.

Passos principais: 1. Processo shell abre o ficheiro 'saída'; 2. Processo shell abre um pipe sem nome; 3. Processo shell criar um novo processo filho Em paralelo: Processo filho filho.1 Redireciona o seu stdout para o ficheiro filho.2 Redireciona o seu stdin para ao extremidade de leitura do pipe filho.3 Executa o

programa 'sort' Processo original (pai): pai.1 Redireciona o seu stdout para a extremidade de escrita do pipe pai.2 Executa o programa 'ls'.

Sugestão: Rever Secção 10.3 - Comunicação entre Processos em Linux

Ex. 4 Considere as funções com os protótipos seguintes:

```
1    int select(int width, fd_set *readfds, fd_set *writefds, fd_set *
        exceptfds, struct timeval *timeout);
2    int read(int fd, void* buf, int count);
```

a) Qual o significado do valor de retorno para cada uma destas funções em condições normais?

O select retorna o número de ficheiros que, entre o conjunto de ficheiros indicado nos bitmaps passados como argumentos, estão prontos (segundo cada classe de operações – ler, escrever ou exceções). O read retorna o número de bytes lidos.

b) Pode ocorrer que estas funções retornem prematuramente sem que todos os acontecimentos esperados tenham sido assinalados, ou sem que todos os dados disponíveis tenham sido efetivamente recebidos.

i) Identifique as ocorrências em que este retorno prematuro tem lugar;

No caso do select, duas situações podem ocorrer, um signal ser recebido ou o timeout ter sido alcançado. No caso do read, pode ter sido recebido um signal.

ii) Identifique de que formas isso se reflete nos valores retornados ou modificados pelas funções e no comportamento assumido por estas.

Situação em que select ou read são interrompidos por signal, função retorna erro (-1). Situação em que select expira timeout: retorna 0.

Sugestão: Rever Secção 10.3.2.7- Sincronização e Espera Alternativa

Ex. 5 Explicite as diferenças entre pipes e fifos (também denominados pipes com nome) em Linux. Descreva, para cada um dos tipos de comunicação, qual o seu âmbito de utilização.

Pipes sem nome: comunicação entre processos pai e filho. Pipes com nome (fifo): comunicação entre quaisquer processos na mesma máquina.

Sugestão: Rever Secção 10.3 - Comunicação entre Processos em Linux

Ex. 6 Considere o ciclo principal de um servidor que faz apenas o eco das mensagens que lhe chegam de clientes. Os clientes podem interagir com o servidor quer com ligações stream TCP quer com datagramas UDP.

```
1    for(;;) {
2        mask = testmask;
3        select(MAXSOCKS,&mask,0,0,0);
4        if(FD_ISSET(strmfd,&mask)) {
5            clilen = sizeof (cliaddr);
6            newfd = accept(strmfd,(struct sockaddr*)&cliaddr, &clilen);
7            handleClient(newfd);  //Auxiliary function that creates a
                 new process to exchange messages with the client
8            close(newfd);
```

```
 9          }
10          if(FD_ISSET(dgrmfd,&mask)) echo(dgrmfd);
11      }
```

a) Qual a primeira linha em que o servidor se bloqueia?

Na linha 3 (espera no select).

b) Explique o funcionamento da variável mask indicando o valor deve ter na linha 3 antes da invocação do select e que valor ou valores terá quando for testada na linha 4.

A variável mask é um vetor de bits. Na linha 3, os bits no vetor correspondentes aos descritores strmfd e dgrmfd devem estar com valor 1; todos os restantes bits do vetor devem estar a 0. Quando for testada na linha 4, tanto o bits correspondentes a strmfd e a dgrmfd podem ter qualquer combinação de valores (consoante strmfd e dgrmfd estejam ou não prontos para ler).

c) Quantos sockets devem ter sido criados antes do servidor executar esta secção de código? Para que são usados?

Foram criados 2 sockets; um socket stream e um socket datagram. O socket stream é usado para aceitar ligações e receber mensa. O socket UDP é usado para receber mensagens.

d) É criado mais algum socket (além dos que indicou na resposta anterior) na sequência acima? Se sim, para que será usado?

É criado mais um socket na sequência da execução da função accept que, quando retorna, retorna o valor do descritor de um novo socket TCP. O novo socket é usado para comunicar com o cliente.

e) Se a chamada sistema da linha 8 não existisse o que poderia suceder? Justifique.

O socket newfd não é usado depois desta linha; se não for fechado fica a ocupar uma entrada na lista de descritores do processo em causa.

Sugestão: Rever Secção 10.3 - Comunicação entre Processos em Linux