Teste T1 – FSO 2012-13 – Sugestoes de resolucao

Consulte os slides e apontamentos de FSO----

Q1. SPOOL – simultaneous peripheral operation online

Consiste num regime em que se suportam multiplas actividades de processamento de entradas e saidas em concorrencia com a execucao de um programa. Tipicamente as actividades de entrada e de saidas controlam as transferencia de dados entre um periferico mais lento e um periferico mais rapido. Este ultimo é tipicamente o disco, no qual sao armazenadas temporariamente os dados que vao sendo transferidos.

Exemplo de SPOOL de entrada:

Um periferico lento Pi1 (antigamente era por exemplo um leitor de cartoes) desencadeia a transferencia de bytes do seguinte modo:

- chega um byte à interface hardware de Pi1
- a interface de Pi1 gera um pedido de interrupcao por hardware, ao CPU
- o programa corrente no CPU é interrompido
- uma RSI – rotina de servico de interrupcoes é activada e vai buscar o byte à interface de Pi1 e coloca-o numa zona de buffer em memoria. No caso de este buffer ficar cheio, aqui a actividade de SPOOL de entrada vai desencadear a escrita do buffer numa zona de disco. Se o buffer nao ficar ainda cheio com este byre, entao a RSI limita-se a retornar e o programa que fora interrompido retoma a execucao no ponto onde ia.

As accoes acima indicadas vao-se repetindo, à medida que byte a byte os dados do periferico lento Pi1 vao chegando. Assim, ao longo deste processo, os bytes vao-se acumulando em zonas de disco, que actuam como se fossem buffers onde estes dados de entrada vao ficando armazenados até que os programas invoquem operacoes de read e entao, em vez de irem ler directamente de Pi1 (esperando mais tempo) basta-lhes lerem das zona de disco, e assim esperam menos tempo.

Outra vantagem é que os perifericos como o Pi1, sendo lentos, sao mantidos em continua operacao, ou seja podem estar sempre activos desde que haja dados a chegar, e estes dados podem ir sendo guardados antecipadamente antes de o programa os pedir.

Repare que o SPOOL pode funcionar mesmo em regime de monoprogramacao: neste caso o unico programa que o SO mantem em execucao, quando precisar de fazer um read vai ler de disco, e embora espere menos tempo do que se tivesse de ler directamente de Pi1 (caso nao houvesse SPOOL), vai mesmo assim esperar da ordem de milisegundos.. e durante esse tempo de espera o CPU nao tem nada que fazer. Por isso o ideal é ter SPOOL e um regime de multiprogramacao.

Exemplo de SPOOL de saida:

Um periferico lento Po1 como uma impressora vai podendo receber bytes um de cada vez. Se o programa corrente de cada vez que quer imprimir um ficheiro tivesse de tratar de enviar directamente para a impressora, o programa ficaria muito tempo a esperar que os bytes pudessem ir sendo enviados para a interface hardware da impressora porque esta é muito lenta.

Assim, havendo SPOOL de saida:

quando o programa quer pedir impressao de bytes, o programa indica isso e faz copias para buffers em memoria (ou indica simplesmente o nome de um ficheiro) e a partir daqui vai ser o SPOOl que trata de enviar para a impressora, mas faz isto de forma concorrente com a execucao do programa
os dados que se quer imprimir sao primeiro copiados para zonas em disco (ou já estao em disco sob a forma de ficheiros e neste caso a unica coisa que se faz é pôr uma copia do ficheiro a imprimir numa fila onde ficarao a aguardar a sua vez de impressao)
a partir das zonas de disco, o SPOOL vai enviando os bytes para buffers em memoria: por exemplo inicialmente copia um bloco de 4Kbytes de disco para um buffer em memoria ,e agora comeca a enviar byte a byte, a partir deste buffer em memoria, para a interface da impressora. De cada vez que a impressora termina de imprimir um byte, gera um pedido de interrupcao ao CPU e o programa corrente é temporariamente interrompido:
uma RSI rotina de servico de interrupcoes vai buscar o proximo byte ao buffer em memoria, e envia-o para a impressora, e retorna ao programa interrompido.
Estas accoes vao-se repetindo até que o buffer de 4K bytes em memoria esteja todo impresso, e quando este buffer fica vazio, o SPOOL vai buscar o proximo buffer a disco e prossegue este processo até nao haver mais bytes para imprimir.
Portanto vê-se que em ambos os casos, de SPOOL de entrada ou de saida, existe um fluxo de bytes a ser transferido de Pi1 para disco e de disco para Po1, sempre atraves de buffers intermediarios em memoria. E as transferencias sao efectuadas à custa do mecanismo de interrupcoes, que permite que, exceptuando os pequenos momentos em que as rotinas de RSI têm de tratar os pedidos de interrupcao do Pi1 ou Po1, nos outros momentos o programa corrente pode estar em execucao, enquanto os perifericos lentos, tranferem os bytes. É esta concorrencia de actividades que caracteriza o regime de SPOOL.

Q2. Mono versus multiprogramacao

O regime de monoprogramacao é um regime de execucao estritamente sequencial em que o SO executa os programas cuja activacao foi pedida pelo utilizador, um de cada vez, ou seja, tendo iniciado a execucao de P1, nao inicia a execucao de qualquer outro programa enquanto P1 nao terminar a sua execucao por completo. Mesmo quando P1 tiver de se bloquear à espera de dados de um periferico ou `espera que um buffer onde escreveu deixe de estar cheio, mesmo nesses casos, o SO não inicia a execucao de outro programa P2 enquanto P1 espera. A consequencia é uma baixa taxa de utilizacao do CPU pois que longos momentos de tempo o CPU nao esta a fazer trabalho util (quando P1 esta a esperar). A taxa de utilizacao dos perifericos tambem é baixa pois que havendo um unico programa P1 activo, este nao dá trabalho suficiente para os perifericos, exemplo de entrada, disco, impressora, etc se manterem em operacao continua, o que seria vantajoso pois estes perifericos sao lentos e convinha irem transferindo dados concorrentemente – ja´ se viu que o SPOOL ajuda um pouco a compensar isto, mas o ideal seria poder ter multiplos programas activos pois assim tambem estes irao fazer mais pedidos aos diversos perifericos, dando assim mais oportunidades para os manter ocupados, donde aumentando as suas taxas de utilizacao. Tambem em relacao ao débito de trabalhos, que é o numero de trabalhos (ou seja pedidos de execucao de programas feitos pelo utilizador) por unidade de tempo, o

regime de monoprogramacao é muito limitado pois que, enquanto o unico programa P1 nao se terminar, todos os outros terao de esperar.... O tempo de resposta a cada programa, dos que estao em espera, tende assim a ser maior do que em multiprogramacao pois que neste caso o SO vai dando atencao a multipros programas de acordo com os pedidos de i/0 ou com o controlo do time-slice.
A multiprogramacao, ao suportar a execucao concorrente de multipros programas, aproveita os tempos em que P1 espera por exemplo por dados, pelo que aumenta a taxa de utilizacao do CPU – ou seja diminui a percentagem de tempo em que o CPU nao esta a fazer trabalho util. Tambem melhora a taxa de utilizacao dos perifericos como se explicou acima. E aumenta o débito dos trabalhos pois que aproveitando melhor o tempo de CPU, consegue que os programas terminem mais depressa.

Q3 – int getchar()

Declaracoes de variaveis globais – note que estas variaveis nao podem ser locais à funcao de getchar pois nesse caso desapareciam do stack de cada vez que getchar retornasse. Ao defini-las como globais, mantem-se entre activacoes sucessivas de getchar

```
 int cont = 0; // contador de bytes no buffer
 int getptr = 0; // indice que indica posicao corrente para ler byte do buffer
 char B[512]; // buffer, vector de bytes onde ficam os bytes lidos do ficheiro
 int fd; // inteiro para guardar o numero de canal aberto para ler do ficheiro
 int primeira = 1; // se valer 1 indica que é a 1ª primeira vez que getchar é chamada, senao indica que o canal fd já foi aberto
 int fim = 0; // se valer 0 indica que ainda nao encontrou fim de ficheiro, irá ficar a 1 quando chegar ao fim de ficheiro

int getchar()
{ int c;  // para retornar o byte lido do buffer por cada invocacao de getchar
  int n; // numero de bytes lidos de ficheiro quando se invoca read—se n == 0 entao encontrou o fim de
  // ficheiro

 if (fim) return -1; // se já antes encontrou fim de ficheiro, retorna logo

 If (primeira) {
   fd = open (filename, O_RDONLY);   // se é a 1ª vez, abre um canal para o ficheiro dado
   primeira = 0; // indica que já abriu
}

if (cont == 0)  { // buffer B está vazio – tem de ir ler do ficheiro
  n = read(fd, B, 512); // pede para ler 512 bytes do ficheiro:: os bytes vão ficar guardados em B, nas
                        // posicoes desde B[0] até B[n-1]; n indica o numero de bytes lidos
  if (n==0) { fim = 1; return -1;} se o buffer estava vazio e encontrou fim de ficheiro, retorna logo
```

```
  cont = n;   // note que n pode ser menor ou igual a 512 – indica o numero de bytes no buffer B
// em geral n será igual a 512 mas quando o byte offset se estiver a aproximar do fim do ficheiro, da
// ultima vez o valor de bytes lidos pode ja´ ser menor do que 512; esses bytes devem ser lidos para o
// buffer
 getptr = 0;  // o buffer foi recarregado pelo que vamos voltar a ler desde a posicao B[0]
}

// uma vez garantido que o buffer B tem algum byte para ser lido, agora as accoes de ler o byte vêm a
// seguir  -- note que neste ponto do programa se tem já sempre o contador cont diferente de 0

c = B[getptr]; // este é o byte a ser retornado por getchar
getptr = getptr + 1; // da proxima vez o indice do proximo byte a ser lido
cont = cont – 1; // porque removeu um byte – o byte continua no buffer mas como avançamos o indice
// é como se o byte agora lido deixe de ficar acessivel

return c; // devolve o byte lido

}
```

Teste T1 – FSO 2012-13 – Sugestoes de resolucao – Q -4– 5-6

Consulte os slides e apontamentos de FSO----

Q4 e Q5

Veja os slides e os apontamentos

Q6.

a)

```
(1) int p1, p2, p3, p4;
(2) p1 = fork();
(3) p2 = getpid();
(4) p3 = fork();
(5) p4 = getpid();
(6) exit(0);
```

O Programa declara 4 variáveis de tipo inteiro.

A activação da execução do programa dá origem a um processo. Chamemos-lhe P0 (o processo Pai). Este processo vai executar todas as instruções desde 2 até 6.

Na instrução2, o processo P0 cria um processo filho. Chamemos-lhe P0-1 (o 1º filho de P0). A partir daqui tanto o processo P0 como o seu filho P0-1 vão executar todas as instruções desde a 3 até à 6. No contexto de P0 a variável p1 recebe o pid do processo criado P0-1. No contexto de P0-1 a variável p1 recebe o valor 0. Note que a partir do ponto 2 passam a existir dois processo (P0 e P0-1) em que cada um deles tem uma cópia privada das variáveis p1, p2, p3 e p4.

- o processo P0:

    executa 3: a chamada getpid( ) vai afectar o valor do identificador do processo P0 (pid) à sua variável privada p2.

    executa 4, pelo que o processo P0 cria um novo filho: chamemos-lhe P0-2 (o 2º filho de P0). A partir daqui, passa a existir uma nova cópia das variáveis p1, p2, p3, e p4 no contexto de P0-2. Estas variáveis, p1 e p2 herdam os valores que P0 tinha na altura do fork da instrução 4. A variável p3 vale 0 no contexto de P0-2;

    o processo P0-2 executa as instruções desde 5 e 6. Na instrução 5 a variavel p4 no contexto de P0-2, recebe o valor do pid do processo P0-2. Na instrução 6 o processo P0-2 termina a sua execução.

    no contexto de P0, a variável p3 vale o pid do processo P0-2.

    o processo P0 executa as instruções desde 5 e 6. Na instrução 5 a variavel p4 no contexto de P0, recebe o valor do pid do processo P0. Na instrução 6 o processo P0 termina a sua execução.

- o processo P0-1:

    executa 3: a chamada getpid( ) vai afectar o valor do identificador do processo P0-1 (pid) à sua variável privada p2.

    executa 4, pelo que o processo P0-1 cria um novo filho: chamemos-lhe P0-1-1 (o 1º filho de P0-1). A partir daqui, passa a existir uma nova cópia das variáveis p1, p2, p3, e p4 que P0-1 tinha, no contexto de P0-1-1.

Estas variáveis, p1 e p2 herdam os valores que P0-1 tinha na altura do fork da instrução 4. A variável p3 vale 0 no contexto de P0-1-1;

o processo P0-1-1 executa as instruções desde 5 e 6. Na instrução 5 a variavel p4 no contexto de P0-1-1, recebe o valor do pid do processo P0-1-1. Na instrução 6 o processo P0-1-1 termina a sua execução.

no contexto de P0-1, a variável p3 vale o pid do processo P0-1.

o processo P0-1 executa as instruções desde 5 e 6. Na instrução 5 a variavel p4 no contexto de P0-1, recebe o valor do pid do processo P0-1. Na instrução 6 o processo P0-1 termina a sua execução.

Conclusão: a execução deste programa cria 4 novos processos: P0, P0-1, P0-2 e P0-1-1.

b)

```
(1) int p1, p2, p3;
(2) p1 = fork();
(3) if (p1 > 0) {
(4) p2 = fork();
(5) } else { p3 = fork();}
(6) exit(0);
```

O Programa declara 3 variáveis de tipo inteiro.

A activação da execução do programa dá origem a um processo. Chamemos-lhe P0 (o processo Pai). Este processo vai executar todas as instruções desde 2 até 6.

Na instrução 2, o processo P0 cria um processo filho. Chamemos-lhe P0-1 (o 1º filho de P0). A partir daqui tanto o processo P0 como o seu filho P0-1 vão executar todas as instruções desde a 3 até à 6. No contexto de P0 a variável p1 recebe o pid do processo criado P0-1. No contexto de P0-1 a variável p1 recebe o valor 0. Note que a partir do ponto 2 passam a existir dois processo (P0 e P0-1) em que cada um deles tem uma cópia privada das variáveis p1, p2, p3.

- o processo P0:

Na instrucao 3, P0 testa o if: como p1 tem o pid do processo P0-1, é um valor positivo pelo que P0 vai executar a instrução 4.

Na instrução 4, o processo P0 cria um novo filho: chamemos-lhe P0-2 (o 2º filho de P0). A partir daqui, passa a existir uma nova cópia das variáveis p1, p2, p3 no contexto de P0-2. A variável p1 em P0-2 herda o valor que P0 tinha na altura do fork da instrução 4 ou seja um valor positivo igual ao pid de P0-1. A variável p2 vale 0 no contexto de P0-2;

o processo P0-2 executa a instrução 6 e termina.

no contexto de P0, a variável p2 vale o pid do processo P0-2.

o processo P0 executa a instrução 6 e termina.

- o processo P0-1:

Na instrucao 3, P0 testa o if: como p1 tem o valor 0 no contexto do processo P0-1, é um valor nulo pelo que P0-1 vai executar a instrução 5.

Na instrução 5, o processo P0-1 cria um novo filho: chamemos-lhe P0-1-1 (o 1º filho de P0-1). A partir daqui, passa a existir uma nova cópia das variáveis p1, p2, p3 no contexto de P0-1-1. A variável p1 em P0-1-1 herda o valor que P0-1 tinha na altura do fork da instrução 5 ou seja um valor nulo. A variável p2 em P0-1 e em P0-1-1 está indefinida; a variável p3 em P0-1-1 vale 0 e em P0-1 vale o pid do processo P0-1-1.

o processo P0-1 executa a instrução 6 e termina.

o processo P0-1-1 executa a instrução 6 e termina.

Conclusão: a execução deste programa cria 4 novos processos: P0, P0-1, P0-2 e P0-1-1.