# Paralelismo versus concorrência

- Execução paralela => hardware
  - Vários computadores, eg. cluster
  - Multiprocessamento, eg. um processo em cada CPU
  - CPU a executar instruções em paralelo com a operação de disco (que se manifesta através de uma interrupção, com prioridade superior à actividade no CPU)





# Concorrência

- Criada pelo SO ao repartir tempo de CPU pelas várias actividades, em resultado de esperas passivas ou desafectação forçada
- Também conhecida por pseudo-paralelismo





# Em geral...

- Seja num ambiente de paralelismo real ou simulado pelo SO
- Existem várias "actividades" em execução "paralela"
- Essas actividades não são independentes, há interacção entre elas
- Facto que levanta algumas questões...





# Papel do SO

- O sistema operativo tem a responsabilidade de
  - Fornecer mecanismos que permitam a criação e interacção entre processos
  - Gerir a execução concorrente (ou em paralelo), de acordo com as políticas definidas pelo administrador de sistemas

# Cooperação e Competição

- Em geral, um conjunto de actividades tem 2 tipos de interacção
  - Cooperam entre si para atingir um resultado comum
    - Processo inicia transferência do disco e aguarda que esta termine
    - O disco interrompe e a rotina de tratamento avisa o processo
  - Competem por recursos partilhados (CPU, espaço livre, etc.)
    - Há necessidade de forçar os processos a esperar até que o recurso fique disponível





# Sincronização

- Em ambos os casos, estamos perante uma questão de sincronização:
  - Cooperação (espera até que evento seja assinalado)
  - Competição (espera até que recurso esteja disponível)
- Sincronizar é atrasar deliberadamente um processo até que determinado evento surja
  - Convém que a espera seja passiva.





# Comunicação

- Para poder haver interacção tem de haver alguma forma de comunicação
  - Processos podem requisitar ao SO um segmento partilhado, podem escrever/ler ficheiros comuns, enviar/receber "mensagens", etc.
  - Threads do mesmo processo podem comunicar através de variáveis globais
- A comunicação pode ser tão simples como o assinalar a ocorrência de um evento (de sincronização), ou pode transportar dados

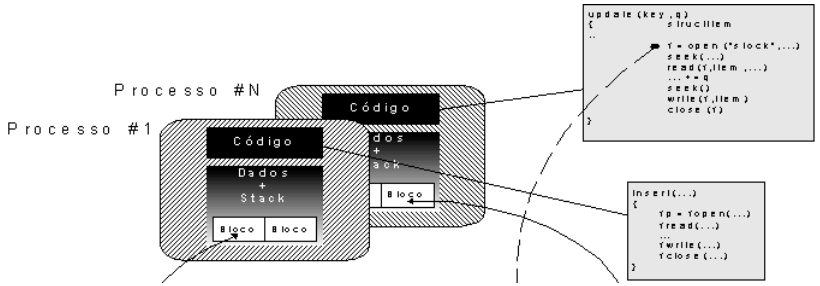




# Exemplos

- Acesso a ficheiros partilhados
- Mirroring de discos
- Impressão

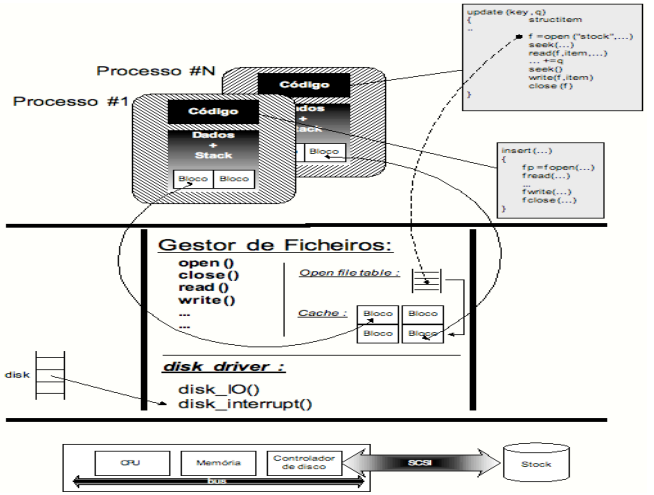
# Acesso a ficheiros partilhados




Dois programas a acederem simultaneamente ao mesmo ficheiro ou base de dados












# Programação Concorrente

- A possibilidade de execução "simultânea" leva ao acesso em concorrência a recursos partilhados.
- O acesso concorrente pode ser feito a zonas de endereçamento partilhadas ou a (genericamente) ficheiros.
- O acesso concorrente pode facilmente resultar na incoerência dos dados partilhados.





# Programação Concorrente

- Para garantir a **coerência** dos dados é necessário que os processos **cooperem** e acedam **ordenadamente** aos recursos partilhados.
- O SO fornece um conjunto de mecanismos que permitem aos processos **sincronizarem-se** e controlarem a ordem de acesso aos recursos partilhados
- *compulsory* ou *advisory*?
- Por exemplo, no caso de acesso a ficheiros
  - Pode usar-se a primitiva lockf para fazer o "lock" de partes de um ficheiros
  - Todos os processos têm de respeitar o protocolo de acesso

# Mais exemplos de concorrência

- Inspirados na vida real
  - Diálogo cliente/bar(wo)men
  - Acesso ao WC
  - Acesso a um parque de estacionamento ou sala de cinema sem marcação de lugar
- São exemplos, respectivamente, de
  - Sincronização
  - Exclusão mútua (caso particular em que capacidade = 1)
  - Controlo de capacidade





# Sincronização

BARMAN

```
while (n_copos_balcao ==
   MAX_BALCAO)
   /* aguarda por vaga no
   balcão*/;
n_copos = n_copos + 1;
pousar_copo_no_balcao();
....
```

CLIENTE

```
while (n_copos_balcao == 0)
   /* aguarda por copo cheio*/
   ;
n_copos = n_copos – 1;
tirar_copo_do_balcao();
…
```





# Regiões críticas

- Para um dado recurso partilhado, cada processo "declara" as regiões do seu código que acedem ao recurso como **regiões críticas**.
- A execução de uma região crítica (relativa a um recurso partilhado X) por parte de um processo está dependente do processo receber garantias de que nenhum outro processo executará a sua região crítica (relativa tb. a X).





# Regiões críticas

- Com mais rigor, deverá ser assegurado que:
  - Correcção: não podem estar dois processos a executar as suas regiões críticas.
  - Justiça: todo o processo que o pretenda deverá inevitavelmente poder executar a sua região crítica*.

*Poderá demorar algum tempo, mas "inevitavelmente" chegará a sua vez

# Regiões críticas

- Formas "pouco interessantes" de implementar regiões críticas:
  - Inibição de interrupções *
  - Test-And-Set *
  - Variáveis de guarda
  - Alternância estrita
  - Algoritmo de Peterson

*Interessante em casos muitos particulares, dentro do sistema operativo





# Exemplo: BA c/ alternância estrita

BARMAN

```
while (1) {
   while (vez != BARMAN)
   /* aguarda vez para por
   copo no balcão*/;
   pousar_copo_no_balcao();
   vez = CLIENTE;
}
```

CLIENTE

```
while (1) {
   while (vez != CLIENTE)
   /* aguarda vez para beber
   */;
   tirar_copo_do_balcao();
   vez = BARMAN;
}
```





# Exemplo: Algoritmo de Peterson

```
int vez;
int interessado[2];
entrar_regiao_critica(int processo)
{
    int outro;
    outro = 1 – processo;
    interessado[processo] = 1;
    vez = processo;
    while (vez == processo && interessado[outro])
      /* espera que o outro saia da região crítica */;
}
```

```
sair_regiao_critica(int processo)
{
    interessado[processo] = 0;
}
```

**Atenção à inversão de prioridades**





# Outros mecanismos de Sincronização

- Formas mais interessantes de implementar regiões críticas e não só:
  - Contagem de eventos
  - Monitores
  - Sleep / Wakeup
  - Semáforos
  - Mensagens

# Semáforos

- Imagine uma caixa com bolas, rebuçados, pedras ...
- E as operações seguintes:

P:

Se há bola(s) na caixa, retiro uma e continuo, senão aguardo (passivamente) que alguém deposite uma.

V:

Devolvo a bola à caixa; se há alguém bloqueado à espera, acordo-a





# Semáforos

- Servem para resolver problemas de sincronização e de exclusão mútua
- Apenas com 3 operações*:
  - Inicialização :

    s = cria_semáforo (valor_inicial)
  - P (s) ou Down (s)
  - V (s) ou Up (s)

* Na realidade há mais operações (e.g. Trylock)





# Semáforos

```
P(s)
{
    s = s - 1
    if (s < 0)
      then bloqueia(...)
}
```

```
V(s)
{
    s = s + 1
    if (s ≤ 0)
      then liberta(queue(s))
}
```

Quando S < 0, o seu valor absoluto |S| conta o número de processos bloquados no P()





# Sincronização com semáforos

- Para cada evento de sincronização, é criado um semáforo com **valor inicial zero**
- Um processo espera *passivamente* pelo evento, e só avança depois do evento acontecer

  P(s) /* se caixa vazia, espera; senão evento já ocorreu */
- Outro processo assinala ocorrência do evento

  V(s) /* se ninguém à espera, deixa bola na caixa para indicar que o evento já ocorreu*/

## BARMAN

```
/* aguarda por vaga no balcão */
/* e só depois vai... */

pousar_copo_no_balcao();

/* avisa que há +1 copo cheio) */
V(copo)
....
```

## CLIENTE

```
/* aguarda por copo cheio */
P(copo)

tirar_copo_do_balcao();

/* avisa que há vaga no balcão
   */
```

**Falta inicializar o semáforo copo a Zero**





# Capacidade

- Para aguardar por espaço no balcão, usa-se um semáforo inicializado à capacidade do recurso partilhado (e não a Zero)
- É um caso particular de sincronização:
  - Só bloqueia se o recurso estiver esgotado naquele instante
  - Equivale a inicializar o semáforo a 0 e de imediato executar tantos V() quantas as posições livres no balcão





## BARMAN

```
/* aguarda por vaga no balcão */
P(espaço)

pousar_copo_no_balcao();

/* avisa que há +1 copo cheio) */
V(copo)
....
```

## CLIENTE

```
/* aguarda por copo cheio */
P(copo)

tirar_copo_do_balcao();

/* avisa que há vaga */
V(espaço)
```

**Falta inicializar o semáforo copo a Zero e espaço à capacidade do balcão**





# Exclusão mútua

- Exclusão mútua é também uma forma de "sincronização"
- Talvez aqui a palavra seja (des)sincronização, pois queremos garantir que 2 ou mais processos não estão simultaneamente dentro da região crítica...
- Esqueleto da solução
  - Entrada: /* espera por região livre, e ocupa-a */
  - Código correspondente à região crítica
  - Saída: /* liberta região*/

# Exclusão mútua com semáforos

- Usando uma variável para descrever o estado da região crítica
- E aplicando a "receita" da Sincronização => inicializa_semáforo(espera, 0)

```
If  (ocupado)  then P(espera)        /*ocupado por outro */
               else Ocupado = true   /* ocupado por mim */
   < região crítica>
Ocupado = false                      /* agora está livre */
/* E falta avisar que a RC já está livre */
```





# Exclusão mútua com semáforos

- Como avisar que a Região Crítica já está livre?
- Não podemos usar apenas com a invocação de um V(espera). Porquê?





# Exclusão mútua com semáforos

- Porque vai dar asneira...
  - O V(espera) seria executado sempre que um processo sai da região crítica
  - E o P(espera) só é executado às vezes quando a RC está "ocupada"
- Se o número de P() não for igual ao número de V(), o semáforo espera deixa de servir para sincronização
  - Um V() incrementa sempre o valor do semáforo
  - E, para bloquear um processo, o semáforo não pode estar positivo quando esse processo executa o P()





# Solução?

- Temos de contar o número de P() e V()!
- Por outras palavras, só devemos executar um V() sabendo que antes foi executado um P()
  - Temos de contar os processos bloqueados
  - E só executar o V() se houver processos bloqueados

# Exclusão mútua com semáforos

Esquecendo por momentos que estamos a usar variáveis partilhadas, logo a criar novas regiões críticas...

```
If  (! Ocupado)
then Ocupado =: true
else {e++; P(espera); e--}
    < região crítica>
if (e==0)
then Ocupado := false
Else V(espera)
```





# Exclusão mútua com semáforos

- Se o código anterior fosse executado dentro do kernel, poderia usar instruções privilegiadas como as que inibem e permitem interrupções
- Mas não posso esquecer de voltar a permitir interrupções **antes** de bloquear o processo!
- E inibi-las novamente a seguir!!!

```
Disable Interrupts
If  (! Ocupado)
then Ocupado =: true
else {
    e++;
    Enable Interrupts;
    P(espera);
    Disable Interrupts;
    e--}
Enable Interrupts;
    < região crítica>
/* e no fim liberta a região */
```





# Percebeu a ideia?

- Manter o número de P() igual ao de V()
- E libertar a exclusão mútua caso um processo se bloqueie dentro de uma Região Crítica, caso contrário temos

DEADLOCK





# Exclusão mútua com semáforos

- O código anterior é meramente académico, e tinha em vista apenas mostrar o raciocínio e algumas das "boas práticas" subjacentes à programação concorrente com semáforos.
- Se pensarmos um bocado, reparamos que um semáforo "tem memória", isto é,

valor inicial + #V() $\geq$ #P() concluídos

# "Receitas" com semáforos

- Sabendo que valor inicial + #V() ≥ #P() concluídos
  - Sincronização:

    valor inicial = 0
  - Capacidade:

    valor inicial = N = capacidade do recurso
  - Exclusão mútua:

    valor inicial = 1





# Exclusão mútua com semáforos

- Para cada região crítica, é criado um semáforo s com valor inicial igual a **UM**
- No início da região crítica

  P(s) /* só avança se região está livre */
- No fim da região crítica

  V(s) /* assinala que a região está livre */





# Produtor/consumidor com semáforos

- Agora que resolvemos os aspectos de sincronização
- E já sabemos a "receita" da exclusão mútua
- É altura de reparar que o balcão é uma variável partilhada pelos vários processos (M barmen + N clientes)

  =>Falta garantir exclusão mútua no acesso ao balcão!





## Produtor(es)

```
P(espaço_livre);
   P(mutex);
      Buf[p++ % N] = px;
   V(mutex);
V(não_vazio);
```

## Consumidor(es)

```
P(não_vazio)
   P(mutex);
      cx = Buf[c++ % N];
   V(mutex);
V(espaço_livre)
```

**Falta inicializar os semáforos espaço_livre a N, não_vazio a ZERO, mutex a UM, e as variáveis p e c a ZERO.**

# Produtor Consumidor

- O exemplo anterior surge frequentemente na comunicação entre processos
  - Utilizam-se buffers múltiplos (porquê?)
  - Tem de ser modificado no caso do "produtor" ser uma rotina de tratamento de interrupções (porquê?)





## RTInt teclado

```
Disable Interrupts;
If livres == 0 then "PIP" Else
   livres--
   Buf[p++ % N] = px;
Endif
Enable Interrupts;
V(não_vazio);
```

## Consumidor(es)

```
P(não_vazio)
Disable_interrupts
cx = Buf[c++ % N];
livres ++
Enable_interrupts
```

**Falta inicializar o semáforo não_vazio a ZERO, as variáveis p e c a zero, e livres = capacidade do buffer.**





# Exercícios

- Barbeiro
- Filósofos
- Parque de estacionamento
- Lockf
- Implementação de monitores e variáveis de condição
- Spooler
- Escalonamento de pedidos de transferência de disco...





# Processos versus threads

- Até agora temos assumido que os processos têm espacos de endereçamento distintos, têm de ser protegidos uns dos outros
- O que complica a implementação do buffer partilhado no nosso algoritmo dos produtores/consumidores
  - Sockets
  - mmap
  - Shmops do Unix V