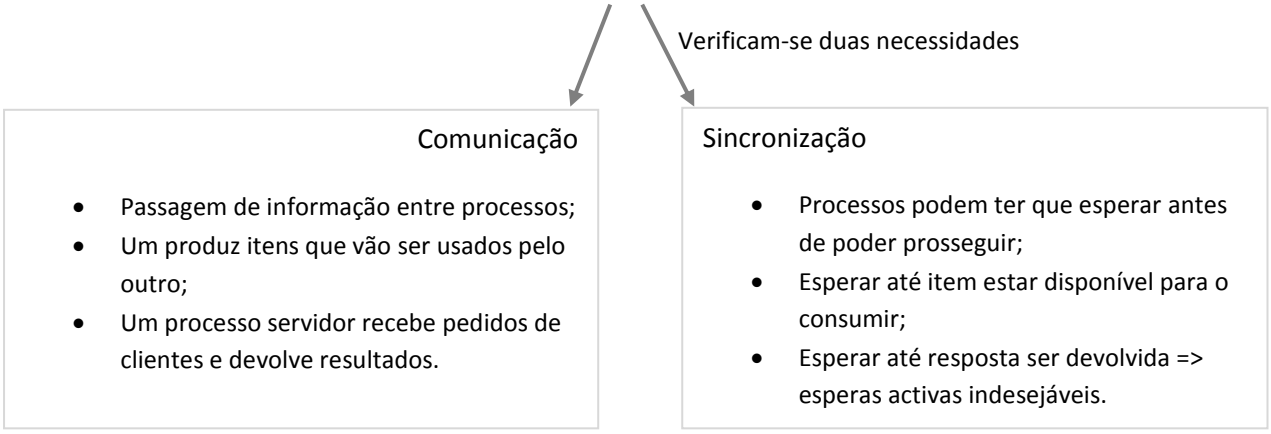
Sistemas Distribuídos: Introdução à Programação Concorrente

Concorrência de processos:

- Processos correm concorrentemente num sistema executando acções independentemente
- Com partilha de tempo, as acções são intercaladas de um modo imprevisível



Processos vs Threads

Processos

- Agrupamento de recursos necessários para a execução de um programa.
- Um processo é criado pelo sistema operativo e podem conter informações relacionadas com os recursos do programa e o estado de execução do mesmo. Possui o seu espaço de endereçamento privado.
- Inicia-se com uma thread de execução.

Threads

- Sequência de instruções que vão ser executadas num programa.
- São entidades escalonadas para a execução sobre o CPU.
- Procedimento que é executado dentro de um processo de uma forma independente. Um processo pode ter várias threads.
- Threads dos vários processos podem correr concorrentemente.
- Threads criadas dentro do mesmo processo:
  - Partilham memória (variáveis globais e heap) e recursos do processo
  - Mantêm o seu stack com variáveis locais privadas

- Pode conter um controlo de fluxo independente e ser escalonável, porque mantém própria:
  - Pilha
  - Propriedades de escalonamento
  - Dados específicos da thread

- Partilham os recursos de um processo:
  - É possível ler e escrever nas mesmas posições de memória, sendo por isso necessária uma sincronização explícita que tem de ser feita pelo programador.

Vantagens do uso de Threads:

- Criar e terminar uma thread é mais rápido do que um processo (menos sobrecarga para o sistema operativo);
- Ganho na performance dos programas;
- A gestão dos threads necessita de menos recursos que a gestão dos processos;
- Todas as threads num processo partilham o mesmo espaço de endereçamento. A comunicação entre threads é, portanto, mais fácil e eficiente;
- As aplicações com threads são mais eficientes e têm mais vantagens práticas:
  - Por exemplo, um programa tem situações onde necessita de efectuar operações de entrada/saída de dados muito demoradas: Enquanto uma thread espera que esta operação termine, o CPU pode ser utilizado para efectuar outras operações através de várias threads.

Race condition – corridas

Quando processos manipulam concorrentemente estrutura de dados partilhada, e o resultado depende da ordem dos acessos.

Exemplo: vários processos acedem a um contador partilhado: `contador = contador + 1;`

```
MOV contador, R0
ADD R0, 1
MOV R0, contador
```

Dado que a execução dos processos pode ser intercalada de diferentes modos, o que pode acontecer ao contador?

Secções Críticas

Um segmento de código que acede a recursos partilhados é uma secção crítica.

As secções crítica têm que ser submetidas a controlo de concorrência, caso contrário temos race conditions.

Uma secção crítica deve ser rodeada por código de entrada (que pede permissão para entrar) e saída.

```
...
codigo de entrada
  seccao critica
codigo de saida
...
```

Uma solução para o problema das secções críticas deve garantir:

exclusão mutua	Se um processo está a executar na sua secção crítica, mais nenhum o pode estar. Somente um processo de cada vez pode entrar na seção crítica.
ausência de deadlock	Se vários processos estão a tentar entrar na secção crítica, um deles deve inevitavelmente conseguir. (progresso) Se nenhum processo estiver na secção crítica e alguns processos quiserem entrar, apenas os processos que execução o código de sincronização podem participar na decisão de quem entra; esta decisão não pode ser adiada indefinidamente.
ausência de starvation	Se um processo tenta entrar na secção crítica inevitavelmente vai entrar. Nenhum processo pode esperar indefinidamente para acessar a secção crítica. (espera limitada) Existe um limite para que o numero de vezes que outros processos podem entrar passando à frente de um processo que já pediu entrada.

Exclusão mútua

A exclusão mútua deve afectar apenas os processos concorrentes somente quando um deles estiver a tentar aceder ao recurso compartilhado. A parte do código do programa onde é feito o acesso ao recurso compartilhado é denominada região crítica. Se for possível evitar que dois processos entrem nas suas regiões críticas ao mesmo tempo, ou seja, se for garantida a execução mutuamente exclusiva das regiões críticas, os problemas decorrentes da partilha serão evitados. Os mecanismos que implementam a exclusão mútua utilizam protocolos de acesso à região crítica. Sempre que um processo deseja executar instruções da sua região crítica, obrigatoriamente deverá executar antes um protocolo de entrada nessa região. Da mesma forma, ao sair da região crítica um protocolo de saída deverá ser executado. Os protocolos de entrada e saída garantem a exclusão mútua da região crítica do programa.

Nos algoritmos que empregam **Exclusão mútua com espera activa**, o processo que testa o acesso a uma região crítica em execução por outro processo, permanece em looping, testando uma condição, até que lhe seja permitido o acesso. As soluções são :

- **Solução de Peterson** (exemplo clássico de solução por software):
  - Na solução, dois itens são partilhados:

```
int vez;
boolean entrar[2];
```

  - O processo i executa:

```
entrar[i] = true;
vez = 1-i;
while (entrar[1-i] && vez == 1-i)
;
//
// seccao critica
//
entrar[i] = false;
```

- **Variáveis Lock**

```
adquirir lock
seccao critica
libertar lock
```

São instruções que o processador executa como uma operação indivisível, isto é, sem interrupções.

Implementação de *locks* com ajuda de hardware:

- inibir interrupções
- **instruções atómicas:**

  - **Test and Set:** instrução que atomicamente coloca o valor de uma flag a verdadeiro e devolve o valor que flag tinha previamente.
    Usando test-and-set é fácil implementar exclusão mútua:
    - flag lock começa a false
    - cada processo faz:

```
while (TestAndSet(&lock))
;
//
// seccao critica
//
lock = false;
```

- **Swap:** instrução que atomicamente troca o valor de duas posições de memória.

  Usando swap é fácil implementar exclusão mútua:
  - flag lock começa a false
  - cada processo faz:

```
key = true;
while (key == true)
Swap(&key, &lock);
//
// seccao critica
//
lock = false;
```

Esperas activas e o SO:

As soluções anteriores envolvem esperas activas, que pode demorar tempo considerável, enquanto outros processos executam as suas secções críticas – inaceitável.

Solução: o SO disponibilizar as primitivas de controlo de concorrência (ex: adquirir e libertar *lock*) para rodear as secções críticas.

Implementação de primitivas de concorrência pelo SO:

1. O kernel pode fazer um processo P passar ao estado bloqueado.
   O processo P, no estado bloqueado, não é escalonado, não consumindo tempo de processador.
2. Mais tarde, um processo Q invoca a libertação de *lock*.
   O código desta primitiva trata de mudar o estado de processo P para pronto.
3. O processo P pode então entrar na secção crítica da próxima vez que for escalonado.

Kernel é o componente central do sistema operativo. As responsabilidades do núcleo incluem a gestão os recursos do sistema.

Races nas estruturas no kernel:

Quando um processo não pode ser "desligado" enquanto corre em modo kerel (em uniprocessadores), não surgem race nas estruturas do kernel.

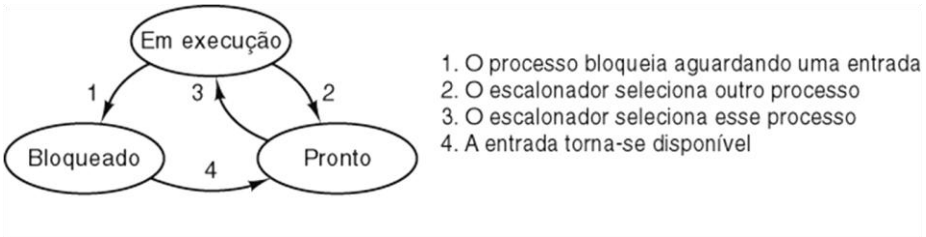
Quando é permitida a desafectação forçada em modo kernel,ou nos multiprocessadores, podem surgir races, levando a cuidados na escrita de código, como:

Breves inibições de interrupções

Esperas activas (raras e de muito curta duração) **->** são relativas a secções que manipulam estruturas do kernel, de duração curta e bem definida.

Os algoritmos que empregam **Exclusão mútua sem espera activa** são mais eficientes que os algoritmos com espera activa, pois os processos não competem pelo uso da CPU, permanecendo em estado de espera. As soluções são baseadas em duas primitivas (chamadas ao sistemas).

- sleep(): bloqueia o processo, até que outro processo o "acorde".
- wakeup(pid): acorda o processo cujo identificador é pid.



Exclusão Mútua vs Ordem de Execução

Exclusão Mútua	Ordem de execução
Quando vários processos concorrem no acesso a recursos partilhados: ○ Processo apenas é impedido de prosseguir temporariamente	Quando existem padrões de cooperação e dependência entre acções de processos: ○ Um processo pode não poder prosseguir até uma dada acção de outro; ○ Bloqueiam-se voluntariamente; ○ São libertados por outros processos.

Sistemas Distribuídos: Semáforos

Um semáforo é um tipo de dados com dois componentes e duas operações atómicas.

Componentes	Operações
o ***v***, valor do semáforo, inteiro não negativo (nunca). o ***l***, conjunto de processos bloqueados no semáforo. o um semáforo é iniciado com um valor v inteiro, e l vazio. o os componentes são manipulados apenas pelas operações.	o são disponibilizadas duas operações wait e signal: estas operações são vistas como atómicas; o wait (P, down ou acquire): tenta decrementar o valor do semáforo, bloqueando o processo no semáforo caso o valor seja 0 – processo corrente; o signal (V, up ou release): se houver processos bloqueados no semáforo, liberta um arbitrariamente; senão, incrementa o valor do semáforo. Esta operação nunca bloqueia.

A implementação das operações é tal que estas têm um comportamento equivalente a serem atómicas:

wait(S):	**signal(S):**

`if S.v > 0:`
`  S.v = S.v - 1`
`else:`
`  S.l.add(p)`
`  suspend()`
	`if S.l = {}:`
	`  S.v = S.v + 1`
	`else:`
	`  q = S.l.pop()`
	`  ready(q)`

e.g. não é possível que, com o semáforo a 1, e dois processos a fazer wait, ambos decidam decrementar o valor do semáforo e retornar, ficando este a -1. Neste caso, o wait retornaria num processo, o valor do semáforo ficaria 0, e

Onde:
- o p é o processo actual que executa
- o `add(e)`: adiciona elemento e ao conjunto
- o `pop()`: remove e devolve um elemento do conjunto
- o `suspend()`: bloqueia o processo corrente e escalona outro processo
- o `ready(p)`: torna o processo p pronto a correr

→ **Semáforo binário:** pode apenas tomar os valores 0 e 1. É também chamado de mutex, pois é apropriado para obter exclusão mútua.

Cada processo que quer entrar numa secção crítica faz:

```
wait(S);
//
// seccao critica
//
signal(S);
```

- No primeiro processo, wait coloca o semáforo a 0 e retorna.
- Se outros processos fizerem wait, este bloqueia os processos no semáforo.
- Quando um processo faz signal, um dos processos é desbloqueado; o valor do semáforo mantém-se a 0.
- O último processo, ao fazer signal, coloca o semáforo a 1.

→ Invariantes sobre semáforos

Sendo:

- k o valor inivial do semáforo S,
- #signal(S) e #wait(S) o número de operações signal e wait concluídas sobre S, temos:
  S.v >= 0 pela definição de semáforos
  S.v = k + #signal(S) – #wait(S) porque:
    - se alguma operação termina mudando S.v, , preserva o invariante;
    - se signal liberta um processo, também termina um wait, S.v fica na mesma, mas também o lado direito da equação.

- Sendo #SC o número de processos na secção crítica: **#SC = #wait(S) – #signal(S)**.
- Pelo segundo invariante sobre semáforos: **#SC + S.v = 1**.
- Este algoritmo garante exclusão mútua: Como **S.v >= 0,** temos que **#SC <= 1**.
- Este algoritmo garante ausência de deadlock:
  em deadlock estariam os processos no wait, **S.v = 0 e #SC = 0;**
  isto contradiz **#SC + S.v = 1**.

- Algoritmo de Udding (sem starvation)

```
Semáforos G1(1), G2(0), M(1); inteiros nG1=0, nG2=0;
wait(G1);
nG1 = nG1 + 1;
signal(G1);
wait(M);
wait(G1); // primeira gate
nG1 = nG1 - 1;
nG2 = nG2 + 1;
if (nG1 > 0)
   signal(G1);
else
   signal(G2);
signal(M);
wait(G2); // segunda gate
nG2 = nG2 - 1;
//
// seccao critica
//
if (nG2 > 0)
signal(G2);
else
signal(G1);
```

→ Semáforos fortes

Semáforos fortes têm uma fila de processos bloqueados (em vez de um conjunto); estes são libertados por ordem de chegada.

Tal permite, por exemplo, evitar starvation na solução simples para secções críticas com N processos.

wait(S): (sendo p o processo actual que executa)

```
if S.v > 0:
S.v = S.v - 1
else:
S.l.append(p) // acrescenta no fim da
fila
suspend()
```

signal(S):

```
if S.l = []:
S.v = S.v + 1
else:
q = S.l.pop(0) // remove e devolve o
primeiro da fila
ready(q)
```

→ **Exemplo: produtor-consumidor com bounded-buffer**

Em geral,

- Utilização de dois semáforos: items e slots.
- Contagem dos itens no buffer e das posições livres.

- Buffer como tipo abstracto de dados com controlo de concorrência:
  - ✓ Exclusão mútua é garantida internamente pelo buffer – buffer contém **mutex** interno.
  - Operações put e take *adquirem* e *libertam* mutex:

```
class Buffer {
 Semaphore mut(1);

 int take() {
  wait(mut);
  x = ...
  signal(mut);
  return x
 }

 put(int x) {
  wait(mut);
  ... x ...
  signal(mut);
 }

}
```

- Acesso directo ao buffer:
  - ✓ Controlo de concorrência no produtor e consumidor.
  - Variáveis: `int buffer[N], itake=0, iput=0;`
  - Dois semáforos para exclusão mútua: um para produtores e outro para consumidores.

Consumidor:

```
while (...) {
 wait(items);
 wait(mutcons);
 x = buffer[itake];
 itake = (itake + 1) % N;
 signal(mutcons);
 signal(slots);
 consume(x);
}
```

Produtor:

```
while (...) {
 x = produce():
 wait(slots);
 wait(mutprod);
 buffer[iput] = x;
 iput = (iput + 1) % N;
 signal(mutprod);
 signal(items);
}
```

Sistemas Distribuídos: Monitores

Monitor é uma técnica para sincronizar duas ou mais tarefas que compartilham um recurso em comum. Com um modelo de concorrência baseado em monitores, o compilador ou o interpretador podem inserir mecanismos de exclusão mútua transparentemente em vez do programador ter acesso às primitivas para tal, tendo que realizar o bloqueio e desbloqueio de recursos manualmente.

- Primitiva estruturada de controlo de concorrência.
- Tipo de dados com operações, que encapsula estado (semelhança com objectos).
- Acesso concorrente é controlado internamente.
- Clientes podem simplesmente invocar operações.
- Apenas um processo pode estar "dentro" num dado momento.
- Exclusão mútua é obtida implicitamente.
- Disponibiliza variáveis de condição (*condition variables* ou ainda *condition queues*).
  - Permitem processos bloquearem-se voluntariamente: processos testam predicado sobre variáveis de estado do monitor e decidem se bloqueiam.
  - Usadas em problemas de ordem-de-execução (mas não só).
    Exemplo: consumidor não pode prosseguir se o buffer estiver vazio.
  - São declaradas explicitamente. Por tradição, o nome deverá sugerir uma condição (predicado), que se verdadeira permite ao processo prosseguir. e.g.:
    condition notEmpty;
    condition notFull;
  - Operações sobre variáveis de condição (monitores clássicos):
    A cada variável de condição é associada uma fila *f* de processos bloqueados. Sendo p o processo actual que executa num monitor mon:

Primitiva waitC bloqueia processo na v.c.	Primitiva signalC liberta processo bloqueado na v.c.

`waitC(cond):`
`cond.f.append(p)`
`signal(mon.mut) // libertar mutex (entidade a qual foi feito lock)`
`suspend() // dirige-se para a fila (dos prontos a executar)`
`wait(mon.mut) // necessito de ter novamente o lock para poder continuar, por isso temos que esperar.`
 	
waitC liberta mutex antes de bloquear processo.
	`signalC(cond):`
	`if cond.f != []:`
	`q = cond.f.pop(0)`
	`ready(p)`
	
	Se não existir processo bloqueado, o signalC "perde-se" (ao contrário dos semáforos).

→ **Exemplo: bounded-buffer (bloqueante) como monitor**

É um tipo abstracto de dados com controlo de concorrência.
A exclusão mútua é garantida internamente pelo buffer: este contém um mutex implícito (não declarado). Cada operação adquire e liberta o mutex, implicitamente. No entanto, pode ser necessário bloquear uma operação se buffer vazio ou cheio.

```
monitor Buffer {
   condition notEmpty;
   condition notFull;
   int a[N], nitems, ...;
   int take() {
      if (nitems == 0)
         waitC(notEmpty);
         x = ...
         nitems--;
         signalC(notFull);
      return x;
   }
      put(int x) {
         if (nitems == N)
         waitC(notFull);
         ...
         nitems++;
         signalC(notEmpty);
}
```

Nota: Uma vez que o buffer trata da exclusão mútua e ordem-de-execução, o caso do produtor e consumidor fica trivial. Sendo buffer um monitor do tipo Buffer atrás:

Consumidor:

```
while (...) {
  x = buffer.take();
  consume(x);
}
```

Produtor:

```
while (...) {
  x = produce():
  buffer.put(x);
}
```

→ **Exemplo: implementar semáforos (fortes) com monitores clássicos**

```
monitor Semaphore {
   int v;
   condition notZero;
   wait() {
       if (v == 0)
          waitC(notZero);
          v = v - 1;
   }
   signal() {
       v = v + 1;
       signalC(notZero);
   }
}
```

A seguir ao signalC a execução continua no waitC (se existir algum processo bloqueado).

Semáforos (fracos) vs. Monitores

Semáforos (fracos)	Monitores
▪ wait pode ou não bloquear ▪ signal tem sempre um efeito ▪ signal desbloqueia processo arbitrário ▪ processo desbloqueado com signal continua imediatamente	▪ waitC bloqueia sempre ▪ signalC perde-se se fila vazia ▪ signalC desbloqueia primeiro processo da fila ▪ processo desbloqueado com signalC necessita readquirir lock

Execução de signalC e waitC

Falha na implementação usando ifs para testes de predicados….
…se um waitC não prosseguir imediatamente a seguir ao signalC, um outro processo pode alterar o estado do monitor. Consequentemente, o predicado que o waitC esperava pode ficar outra vez falso.

Candidatos a prosseguir aquando um signalC:

- processos que fizeram signalC (S) (caso este não prossiga logo);
- processos desbloqueados, à espera de retornar do waitC (W);
- processos à espera de entrar (E), adquirindo o lock;

Monitores mais em uso actualmente, e.g. Java e Pthreads, têm: E = W < S (como variante).
Ou seja: primeiro continua o processo que faz signalC; depois pode correr o processo acordado ou pode correr um terceiro processo que estivesse a querer entrar;
Como um terceiro processo pode ter mudado o estado do monitor, o predicado pode já não ser verdadeiro depois do waitC.
Conclusão: Utilização de testes de predicados com while:

```
while (!predicado())
wait(cond);
```

Outro fenómeno vai, obriga também o uso de while: os **spurious wakeups**.

Para obter implementações eficientes de monitores em multiprocessadores, um waitC pode, embora muito raramente, desbloquear mesmo sem ninguém ter feito signalC.

→ **Exemplo: leitores e escritores** (caso mais geral de exclusão mútua).

- Duas classes de processos:
  - readers: querem fazer operações de leitura sobre um recurso;
  - writers: querem fazer operações de escrita sobre um recurso;

- Um bloco de operações de leitura ou escrita é rodeado de código de sincronização; assim existem 4 operações:
  - `startRead` e `endRead` para rodear bloco de leitura;
  - `startWrite` e `endWrite` para rodear bloco de escrita.

- Requisitos de segurança:
  - podem estar vários processos a ler;
  - se um processo estiver a escrever, mais nenhum pode estar a ler ou escrever.

Implementação de Leitores e escritores (com monitores modernos)

- **starvation de escritores**

```
monitor RW { // E = W < S
  int readers = 0, writers = 0;
  condition OKread, OKwrite;

  startRead() {
    while (writers != 0) waitC(OKread);
    readers++;
    signalC(OKread);
  }

  endRead() {
    readers--;
    if (readers == 0) signalC(OKwrite);
  }

  startWrite() {
    while (writers != 0 || readers != 0)
    waitC(OKwrite);
    writers++;
  }

  endWrite() {
    writers--;
    signalC(OKread);
    signalC(OKwrite);
  }
}
```

- **starvation de leitores**

```
monitor RW { // E = W < S
  int readers = 0, writers = 0, wantWrite = 0;
  condition OKread, OKwrite;

  startRead() {
    while (writers != 0 || wantWrite > 0)
    waitC(OKread);
    readers++;
    signalC(OKread);
  }

  endRead() {
    readers--;
    if (readers == 0) signalC(OKwrite);
  }

  startWrite() {
    wantWrite++;
    while (writers != 0 || readers != 0)
    waitC(OKwrite);
    wantWrite--; writers++;
  }

  endWrite() {
    writers--;
    signalC(OKread);
    signalC(OKwrite);
  }
}
```

- **sem starvation**

```
monitor RW { // E = W < S
   int readers = 0, writers = 0, wantRead = 0, wantWrite = 0, turn = R;
   condition OKread, OKwrite;

   startRead() {
      wantRead++;
      while (writers != 0 || (turn != R && wantWrite > 0)) waitC(OKread);
      wantRead--; readers++;
      if (wantRead != 0) signalC(OKread);
      else turn = W;
   }

   endRead() {
      readers--;
      if (readers == 0) signalC(OKwrite);
   }

   startWrite() {
      wantWrite++;
      while (writers != 0 || readers != 0) waitC(OKwrite);
      wantWrite--; writers++;
   }

   endWrite() {
      writers--; turn = R;
      if (wantRead != 0) signalC(OKread);
      else signalC(OKwrite);
   }
}
```

Sistemas Distribuídos

Introdução aos sistemas distribuídos

Sistema Distribuído: conjunto de computadores ligados em rede, com software que permite a partilha de recursos e a coordenação de actividades, oferecendo idealmente um ambiente integrado.
Um sistema distribuído é um sistema em que a falha de uma máquina da qual nunca se ouviu falar pode tornar a nossa própria máquina inútil.

- Entidades de processamento independentes que comunicam por troca de mensagens.
- Possibilidade de falhas independentes.
- Heterogeneidade de hardware e software: a utilização de diferentes máquinas diminui a probabilidade de uma máquina falhar, se anteriormente outra tiver falhado, pela mesma razão.
- Nos sistemas assíncronos não existe:
  - limite para as velocidades relativas de processamento,
  - limite para o tempo de comunicação,
  - relógio global.
- Software (tem de ser) projectado tendo em conta a escalabilidade.
- Exigem fiabilidade e segurança mesmo em presença de falhas de computadores e canais de comunicação.

Características de um SD

Sistema e aplicações deverão ser concebidos de forma a garantir estas características:

Partilha de recursos

Hardware: impressora, discos, scanner, etc;
Software: ficheiros, bases de dados, compiladores, etc.
A partilha de recursos reduz custos e é necessária como suporte ao trabalho cooperativo.
Verifica-se o encapsulamento dos recursos pela máquina que os contém. Os gestores de recursos oferecem interfaces standard para aceder aos recursos.
Motivam modelo cliente/servidor / modelo baseado em objectos.

Abertura

Extensibilidade do software e hardware com possível heterogeneidade.
É obtida especificando e documentando interfaces.
O UNIX é exemplo de um sistema concebido com a Abertura em mente.
Um sistema distribuído aberto possui um mecanismo uniforme de comunicação entre processos.

Concorrência

Existe e deve ser suportada em SD's por vários motivos:

- Vários utilizadores/aplicações podem invocar comandos simultaneamente.
- Um servidor deve poder responder concorrentemente a vários pedidos que lhe cheguem.
- Vários servidores devem poder correr concorrentemente colaborando na resolução de pedidos.

Escalabilidade

O software deve ser pensado de modo a funcionar eficaz e eficientemente em sistemas de escalas diferentes, isto é, o trabalho envolvido no processamento de acessos a um recurso partilhado deverá ser independente do tamanho do sistema. Tanto o sistema como as aplicações não deverão necessitar de alterações com o aumento da escala.
Devem ser evitados algoritmos e estruturas de dados centralizadas.
Nenhum recurso deve ser escasso - em caso de necessidade deve ser possível aumentar o número de recursos de forma a satisfazer a procura.
As soluções passam por replicação de dados, caching, algoritmos distribuídos.

Tolerância a faltas

Os sistemas por vezes falham. Na presença de falhas, os programas podem produzir resultados errados ou terminar prematuramente. As soluções passam por redundância de hardware e/ou de software (servidores/serviços). Portanto, quando um componente falha, apenas o trabalho que usa este componente deve ser afectado e deve poder reiniciar ou continuar usando outro componente.
Em SD a disponibilidade pode ser maior que em sistemas centralizados, mas exige maior complexidade do software.
Motiva paradigmas mais avançados que cliente servidor (eg. comunicação em grupo, transacções).

Transparência

O sistema deve ser visto como um todo e não como uma colecção de componentes distribuídos.
Tipos de transparência: acesso, localização, concorrência, replicação, falhas, migração, desempenho e escalabilidade. Transparência de acesso + localização = transparência de rede.
Exemplo: email. Contra-exemplo: login.

Aspectos da concepção de um SD

Estes aspectos de concepção são consequência directa da distribuição:
(Outros, mais genéricos, continuam a ser importantes.)

Nomes

Os SD são baseados na partilha de recursos e na transparência da sua distribuição. O significado deve ser global e independente da localização do recurso.
Existe necessidade de resolução de nomes (eg. hostname → IP).
O mecanismo de tradução deve ser escalável e eficiente.
O espaço de Nomes pode ser simples (flat) ou estruturado (hierárquico).
Os nomes são resolvidos dentro de Contextos.
Serviço de Nomes:

- Traduz Nomes de um Espaço de Nomes para outro.
- Um Nome é resolvido quando é obtido um identificador que permite a invocação de uma acção sobre o recurso a que diz respeito o Nome.

Comunicação

Os processos de um SD encontram-se lógica e fisicamente separados interagindo por troca de mensagens. A comunicação entre dois processos envolve a transferência de dados do espaço de endereçamento do emissor para o do receptor. Por vezes é necessária a sincronização do receptor com o emissor, de forma que o emissor ou o receptor e bloqueado até o outro o permitir libertar.

Número e complexidade das camadas de software.

Compromisso entre desempenho e modelo de programação de alto nível.

Necessidades: transferência de dados e sincronização.

Primitivas de comunicação:

- send e receive realizam a passagem de mensagens entre pares de processos.
- envolve a transmissão de dados (mensagem) através dum meio de comunicação especificado (canal) e a recepção da mensagem.

A transmissão pode ser:

- assíncrona: a mensagem e enviada e o emissor prossegue.
- síncrona: o emissor espera ate a mensagem ser recebida.

Padrões de comunicação: Send-Receive básico; Cliente-Servidor (prestação de serviços; pedido-execução-resposta).

Difusão: destinatário e um grupo de processos e a um send corresponde um receive por membro do grupo.

Diferentes modelos de programação incluem primitivas básicas tipo send/receive (eg. sockets), invocação remota de procedimentos (eg. RPC), invocação de operações em sistemas de objectos distribuídos (eg. Java RMI, CORBA), e padrões de comunicação de mais alto nível assegurando ordem, atomicidade, domínios de filiação.

Estrutura do software

Num sistema centralizado, o SO ó tipicamente monolítico e oferece uma interface fixa.

Em sistemas distribuídos, o kernel pode assumir um papel mais restrito, gerindo os recursos mais básicos. É dado ênfase a serviços, que podem ser fornecidos por processos em modo utilizador.

Categorias de software num SD:

- Aplicações
- Sistema operativo
  - gestão básica de recursos (alocação e protecção de memória; criação e escalonamento de processos; comunicação entre processos; gestão de periféricos)
  - abstracções normais de programação
  - garantias de confiabilidade e segurança do SO
- Serviços
  - Funcionalidades comuns necessárias a programação de aplicações distribuídas
  - Oferece modularidade e adaptabilidade ao sistema
  - Serviços básicos (ficheiros) até de alto nível (email)
- Suporte à programação
  - Funcionalidades em runtime de suporte a primitivas / construções oferecidas pelas linguagens/ambientes de programação.

Alocação de carga

Modelos: Workstation-Server, Processor pool e Uso de workstations desocupadas

- Workstation-Server

Alocação da carga nas estações de trabalho (perto do utilizador) - privilegia aplicações interactivas.
Capacidade da estação de trabalho determina o tamanho da maior tarefa do utilizador.
Não optimiza o uso das capacidades (processamento e memoria) através do SD.
Um utilizador não consegue obter mais recursos dos que os locais.

- Processor pool (eg. Amoeba)

Alocação dinâmica de processadores a utilizadores num modelo baseado no Workstation-Server.
O parque de processadores consiste numa colecção (heterogenia) de computadores de baixo custo. Cada parque tem uma ligação independente a rede.
Os processadores são alocados aos processos até estes terminarem.

- Uso de workstations desocupadas

As estações de trabalho com baixa carga ou desocupadas são em determinados períodos alocadas a tarefas de outros utilizadores (remotos).
É uma mistura dos modelos anteriores.

Coerência

Coerência de: Actualizações, Replicação, Cache, Falhas e Interface com o utilizador.

- Actualizações:

Acontece quando vários processos acedem e actualizam os mesmos dados concorrentemente.
A actualização de um conjunto de dados não e executada instantaneamente.
A actualização deve, no entanto, ser atómica. Deve parecer instantânea.

- Replicação:

Se os dados de determinado item forem copiados para vários computadores e posteriormente alterados num ou em vários deles, então surge a possibilidade de inconsistências dados os vários valores para o item nos diversos computadores.

- Cache:

Surge sempre que os dados de determinado item remoto são guardados em cache e o valor do item e alterado (similar a replicação).

- Falhas:

Quando um componente crucial ao sistema falha, os outros inevitavelmente falharão (mesmo que forçados). Dependendo da altura em que cada componente falhe, os seus estados podem divergir.

- Interface com o utilizador:

Incoerências provocadas pela latência de input/ processamento/output e de comunicação.

Expectativas de utilização:

Funcionalidade

Mínimo - a funcionalidade oferecida pelo SD deve ser no mínimo a esperada de um único computador.

Esperado - o SD deve oferecer uma melhoria sobre o serviço de qualquer computador isolado - eg. uma maior variedade de serviços e recursos.

Reconfiguração

Capacidade de acomodar alterações sem por em causa o sistema.

Os SD devem poder suportar:

- alterações rápidas, eg. falha de um processo ou canal de comunicação, variações de carga, migração de recursos ou dados.
- de médio e longo termo, eg. mudanças de escala, acomodação de novos e diferentes componentes, mudança de função dos componentes.

Qualidade de serviço (QoS)

Desempenho: tempos de resposta.

Fiabilidade: desvios da especificação.

Disponibilidade: resiliência.

Segurança: privacidade e integridade.

Sistemas Distribuídos

Arquitectura de sistemas distribuídos

É a forma de descrever a estrutura do sistema distribuído.
O objectivo de uma arquitectura para sistemas distribuídos e a standarização dos componentes do sistema, das formas como interagem e de como são geridos.

- Interoperabilidade: aplicações que usem a mesma arquitectura deverão interagir sem dificuldades.
- Facilita o desenvolvimento: incentiva o desenvolvimento por refinamentos.
- Confiabilidade: a modularidade incentiva o isolamento de falhas.

Aspectos de concepção. As arquitecturas para sistemas/aplicações distribuídas privilegiam os seguintes aspectos:

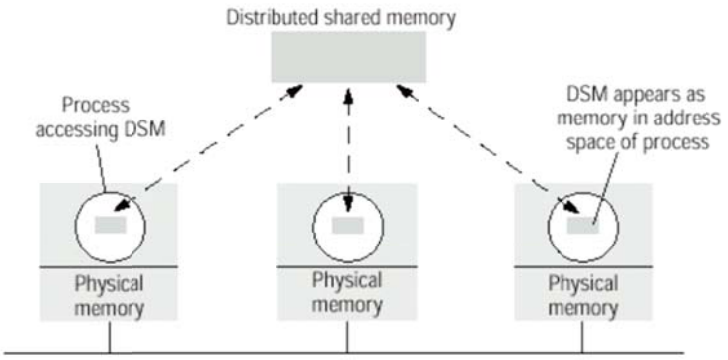
- Independência de localização
- Gestão de conectividade
- Escalabilidade
- Performance
- Segurança
- Equilíbrio de carga
- Tolerância a faltas

Paradigmas e modelos de programação.
Sistema centrado no processador:

- Data shipping: é arquitectura de distribuição onde os objectos de dados são recuperados do servidor, e são armazenados e operados nos nodos dos clientes. Esta arquitectura reduz a latência da rede e aumenta a utilização de recursos no cliente.
  - Sessão remota
  - Transferência explícita de ficheiros
  - Sistema de ficheiros em rede.
- Problemas: desempenho; controlo de concorrência; confiabilidade, etc.

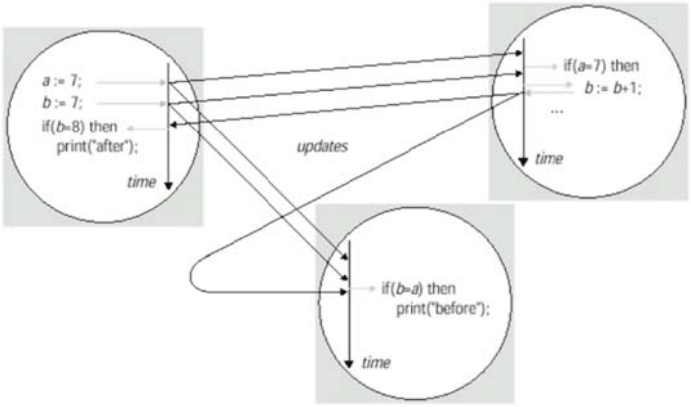
Sistema centrado no armazenamento:

- DSM (Distributed Shared Memory): é um conceito em ciência da computação que se refere a uma ampla classe de implementações de software e hardware, em que cada nó de um cluster tem acesso à memória compartilhada, além de memória não compartilhada de cada nó privado.



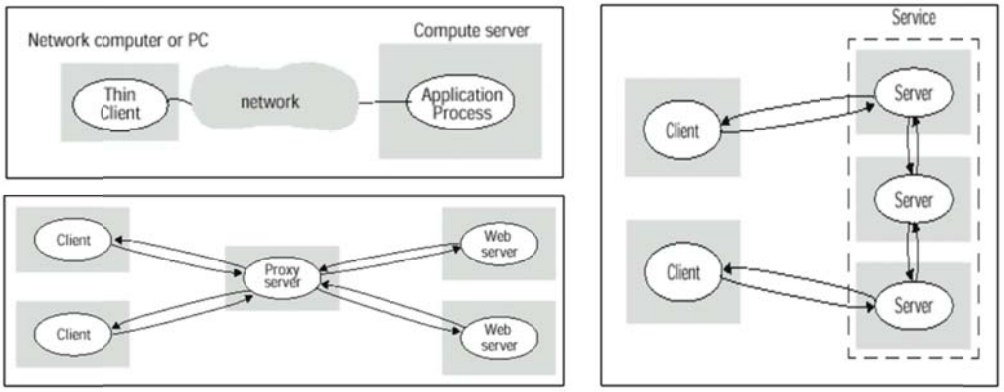
+ Abstracção, adaptação de aplicações
- Complexidade, modelo de coerência, desempenho

- Function shipping: é o processo de transferência de serviços lógico para as áreas físicas de um sistema onde eles podem ser perfeitamente executados. Quando o tráfego de comunicação entre os serviços está congestionado, muitas vezes é benéfico para os serviços a residir na mesma máquina, embora uma configuração ideal é geralmente específica para a implementação real do sistema e utilização.
  - Passagem de mensagens; Cliente / Servidor



Sistema centrado na comunicação:
- Generalização de cliente/servidor
- Ênfase em serviços
- Barramento de dados e funções
- Barramento de objectos - mediador de pedidos

Cliente/Servidor



- Servidor:
  grande capacidade de armazenamento e/ou processamento.
  periféricos especiais. Ex. impressão, salvaguarda, ...
  serviços particulares. Ex. pesquisa, autenticação, ...

  Exemplos de servidores vulgares: ficheiros, base de dados, informação de rede, nomes, email, autenticação.

- Cliente: interface com o utilizador; caching.

→ Servidor *stateless*

Servidor diz-se sem estado (stateless) se, de uma invocação para a seguinte, o servidor não mantém nenhuma informação acerca do estado do cliente (eg. não "sabe" se o cliente guarda algo em cache).

O cliente é o único responsável pelas suas acções, devendo portanto confirmar sempre se o que tem em cache ainda é válido.

Exemplo: Web proxies, web servers, NFS.

Servidor stateless executa a operação pedida e "esquece" o cliente

- recupera muito depressa após um crash.
- tipicamente precisa de mais argumentos em cada invocação.
- o "estado" está no cliente e é passado ao servidor, quando necessário (eg. cookies).

→ Servidor *stateful*

Servidor diz-se com estado (stateful) se o servidor mantém informação sobre os clientes com "sessões em aberto" (eg. o servidor é dono da cache e faz chamadas aos clientes quando os dados são alterados.

O cliente pode confiar na validade da cache.

Exemplo: TP monitors, database servers.

Servidor stateful sabe "tudo" acerca dos clientes

- pode ser mais eficiente, pois tem mais informação
- é complicado recuperar o estado perdido após uma falha: pedir a outro(s) processo(s) que lho passem? Quem são? Clientes? São de confiança? Estão acessíveis? Grupo de servidores replicados? E se foi uma falsa suspeita de falha?
- estado em memória persistente demora a actualizar e a recuperar

Sistemas Operativos Distribuídos.

O objectivo de um SOD é permitir programar um sistema distribuído de forma a que este possa ser usado pelo mais vasto leque de aplicações.

Este objectivo consegue-se oferecendo às aplicações abstracções, gerais e orientadas ao problema em mãos, dos diversos recursos do sistema. Exemplo disto é oferecer às aplicações processos e canais de comunicação em vez de processadores e uma rede de dados.

Um SOD é constituído por um conjunto de núcleos (kernels) e de servidores (processos em modo utilizador).

Este conjunto de núcleos e de servidores forma uma infra-estrutura genérica e transparente em termos de rede de comunicação, para a gestão de recursos.

Encapsulamento de recursos. Um SOD deve tornar os seus recursos acessíveis em toda a rede de comunicação salvaguardando:

- Encapsulamento: cada recurso deve ter uma interface bem definida para acesso pelos clientes. Todos os detalhes de implementação e gestão do recurso devem ser escondidos.
- Concorrência: os clientes devem poder partilhar e aceder concorrentemente aos recursos.
- Protecção: os recursos devem ser protegidos contra acesso ilegítimos.

Acesso aos recursos. Os clientes acedem aos recursos identificando-os em argumentos de chamadas ao sistema. Ao acesso a um recurso chamamos invocação. A invocação tem associadas tarefas que competem ao SOD:

- resolução de nomes
- comunicação
- escalonamento de acessos

Núcleos monolíticos. O UNIX é normalmente considerado monolítico:

- Executa todas as operações básicas (algumas que não requerem necessariamente privilégios especiais).
- Consiste num único bloco de código de funcionalidades indiferenciadas.
- A falta de modularidade faz com que a alteração e adaptação de componentes individuais seja difícil e se repercuta nos resto do núcleo.

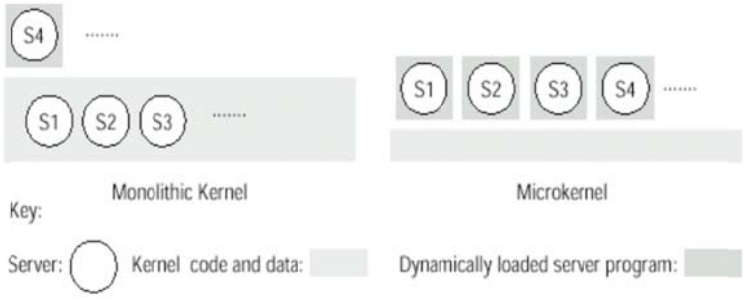
Micro-núcleos. Um micro-núcleo oferece apenas as funcionalidades mais básicas: gestão de processos, gestão de memória e comunicação entre processos. Todos os outros serviços do sistema são providos por servidores activados dinamicamente. Estes serviços são activados nas máquinas das quais se espera o serviço.
Os serviços do sistema são acedidos através de invocações (normalmente por RPC).

Arquitectura de micro-núcleos:
Os micro-núcleos devem ser transportáveis entre diferentes arquitecturas de computadores.
Os seus componentes principais incluem:

- Gestor de processos: criação e gestão a baixo-nível de processos.
- Gestor de threads: criação, sincronização e escalonamento.
- Gestor de memória: gestão da memória fisíca.
- Supervisor: processamento de interrupções e chamadas ao sistema.

Exemplos de micro-núcleos: Mach, Chorus, Amoeba, Clouds

Núcleos monolíticos vs. Micro-núcleos



Os micro-núcleos destacam-se pela sua abertura e modularidade salvaguardando a protecção de memória.	Em núcleos monolíticos é muito difícil extrair módulos de forma a poderem executar remotamente noutra máquina.
A (grande) vantagem dos núcleos monolíticos é o desempenho.	

Comunicação distribuída.

A abstracção básica da comunicação distribuída é a passagem de mensagens.
Para a maior parte dos sistemas/aplicações, o uso directo de primitivas send/ receive revela-se de demasiado baixo nível.
A abstracção para comunicação distribuída por excelência é a invocação remota de procedimentos - RPC.

Mapeamento da estrutura de dados. Os dados num programa são representados por estruturas mais ou menos complexas. Os dados nas mensagens são sequenciais, em cadeia.
As estruturas de dados têm que ser "achatadas" no envio e reconstruídas na recepção.
Os dados numa mensagem podem ser de diferentes tipos.

Nem todos os computadores representam os valores (inteiros, caracteres) da mesma forma.
A troca de dados entre dois computadores:

- requer que os valores sejam convertidos antes do envio para um formato externo pré-acordado e convertidos para o formato local na recepção.
- ou o envio dos dados na sua forma original, acompanhados pelo tipo de arquitectura do emissor. O receptor deverá ser capaz de fazer a transformação necessária.
- não requer a conversão entre computadores do mesmo tipo.

Primitivas de envio e recepção.

A passagem de mensagens é suportada por duas primitivas: enviar e receber.
Um processo envia uma mensagem para o destinatário e outro processo, no destino, recebe a mensagem.
A cada destinatário é associada uma fila: o emissor coloca mensagens na fila e o destinatário remove-as.

Comunicação síncrona

- O emissor e receptor sincronizam em cada mensagem.
- O envio e a recepção são operações bloqueantes.
- Aquando do envio, o emissor bloqueia até à recepção da mensagem.
- Aquando da recepção, o receptor bloqueia até chegar uma mensagem.

Comunicação assíncrona

- O envio não é bloqueante: o emissor pode continuar a sua execução logo após o envio da mensagem (que é copiada para um buffer local).
- A transmissão da mensagem processa-se em paralelo com a execução do emissor.
- A recepção pode ou não ser bloqueante.

Identificação do destinatário

O envio de uma mensagem requer a indicação do identificador do(s) seu(s) destinatário(s).
Identificadores independentes da localização:

- Quando o identificador do destinatário é independente da localização o identificador é, aquando o envio, mapeado para um endereço de baixo nível onde de facto é entregue a mensagem.
- O uso de identificadores independentes da localização permite a recolocação de serviços sem aviso prévio aos clientes.

Destinatários das mensagens: Processos, Portos, Sockets, Grupos de processos, Grupos de portos, Objectos.

Fiabilidade

Entrega não fiável:

- a mensagem é transmitida uma única vez, sem espera de confirmação (ack).
- não há garantia de que a mensagem enviada seja de facto entregue (eg. UDP).
- ao utilizar este serviço, fica a cargo dos processos assegurar a fiabilidade de que precisam.

Entrega fiável:

- pode ser implementada sobre a entrega não fiável através de confirmações de recepção.
- cada mensagem deverá ter um identificador único.

Comunicação Cliente/Servidor

Suporte ao paradigma cliente/servidor

Primitivas:

- DoOperation: usado pelos clientes para a invocação de operações remotas.
- GetRequest: usado pelos servidors para obter pedidos de serviço.
- SendReply: usado pelos servidores para enviar respostas aos clientes.

A resposta do servidor funciona como confirmação da invocação.

Protocolos de invocação remota de procedimentos (RPC):

- R: Request – pode ser usado quando o procedimento não devolve nada e o cliente não pretende confirmação da recepção. O cliente não bloqueia.
- RR: Request-Reply – comunicação cliente/servidor típica.
- RRA: Request-Reply-Acknowledgement – o cliente confirma a resposta do servidor.

Comunicação em grupo

Na comunicação entre um grupo de processos é utilizada a difusão de mensagens.

A difusão de mensagem é utilizada na implementação de sistemas distribuídos com as seguintes características:

- Tolerância a faltas baseada em serviços replicados – um serviço replicado consiste num conjunto de servidores. As invocações dos clientes são difundidas a todos os processos do grupo que executam a mesma operação.
- Localização de processos num sistema distribuído
- Melhor desempenho através de dados replicados
- Notificações múltiplas

Propriedades dos protocolos de comunicação em grupo:

- Difusão fiável: Todos os processos que não falhem recebem a mensagem.
- Difusão atómica: Difusão fiável com ordem total na entrega das mensagens.

Sistemas Distribuídos

Coordenação distribuída

Ordenação de eventos

Causalidade

Relação Happened-before (→)

- Se A e B são eventos do mesmo processo e A foi executado antes de B, então A→B.
- Se A é o evento de envio de uma mensagem por um processo e B o evento de recepção dessa mensagem por outro processo, então A→B.
- Se A→B e B→C então A→C

Realização de →

Associa-se um etiqueta temporal a cada evento do sistema de forma a que se A→B então a etiqueta de A é menor que a etiqueta de B.

- Cada processo tem associado um relógio lógico. O relógio é um contador que é incrementado entre cada dois eventos sucessivos do processo.
- Cada mensagem enviada transporta o instante lógico em que foi enviada.
- Ao receber uma mensagem, o processo acerta o seu relógio com o instante da mensagem se este último for mais recente.

Exclusão mútua distribuída

- Assumpções: o sistema consiste de n processos; cada processo no seu processador. Cada processo tem uma zona crítica que requer exclusão mútua.
- Requisitos: se um processo se encontra a executar a sua zona crítica, então mais nenhum processo se encontra a executar a sua.

Exclusão mútua distribuída: alg centralizado

- Um processo é escolhido para coordenar o acesso à zona crítica.
- Um processo que queira executar a sua zona crítica envia um pedido ao coordenador.
- O coordenador decide que processo pode entrar na zona crítica e envia a esse processo uma resposta.
- Quando recebe a resposta do coordenador, o processo inicia a execução da sua zona crítica.
- Quando termina a execução da sua zona crítica, o processo envia uma mensagem a libertar a zona.

Exclusão mútua distribuída: alg distribuído

- Quando um processo Pi pretende aceder à sua zona crítica gera uma etiqueta temporal TS e envia um pedido (Pi, Tsi) a todos os processos.
- Quando um processo recebe um pedido pode responder logo ou adiar a sua resposta.
- Quando um processo recebe respostas de todos os processos no sistema pode então executar a sua zona crítica.

- Depois de terminar a execução da sua zona crítica, o processo responde a todos os pedidos aos quais adiou a resposta.
- A decisão de Pj responder logo a um pedido (Pi, TSi) ou adiar depende de três factores:
  - Se Pj estiver na sua zona crítica, adia.
  - Se Pj não pretender aceder à sua zona crítica, responde.
  - Se Pj pretender aceder à sua zona crítica (e já enviou também um pedido) então compara a etiqueta temporal do seu pedido TSj com TSi:
    - Se TSj > TSi então responde logo (Pi pediu primeiro)
    - Senão adia a resposta.
- Comportamento desejado da solução distribuída:
  - Não há deadlocks.
  - Não há starvation. Garantido pela ordem total dada pelos relógios lógicos.
- Problemas da solução distribuída:
  - Cada processo tem de conhecer todos os outros.
    Conjunto estático.
  - Não tolera a falha de nenhum processo.

Atomicidade

Dada uma sequencia distribuída de operações, ou são todas executadas ou não é nenhuma - a esta sequencia chamamos transacção.
A atomicidade num sistema distribuído requer um coordenador de transacções responsável por:

- Iniciar a transacção.
- Distribuir a transacção pelos respectivos nodos.
- Coordenar a terminação da transacção que resulta no sucesso (commit) ou falha (abort) por todos os nodos.

Compromisso em duas fases

Protocolo de terminação de transacções (Twophase commit). O protocolo é iniciado após o fim da última operação da transacção e envolve todos os nodos em que se executa a transacção.

fase 1

Seja T uma transacção e Ci o coordenador:

- Ci adiciona <preparar T> ao log
- Ci envia <preparar T> a todos os nodos
- Quando recebe <preparar T> cada nodo responde consoante o sucesso das operações por si executadas:
  - Se falhou adiciona <no T> ao log e responde a Ci com <abort T>
  - Se teve sucesso adiciona <yes T> ao log, grava o log em disco, e responde a Ci com <commit T>
- O coordenador Ci recebe as respostas:
  - Se todos os nodos respondem <yes T> então a decisão é commit.
  - Basta que um nodo responda <no T> e a decisão é abort.
  - Basta que um nodo demore a responder (time out) e a decisão é abort.

fase 2

- O coordenador Ci :
  - Torna a decisão persistente escrevendo-a em disco.
  - Envia a a decisão a todos os nodos.
- A decisão é irrevogável.
- Todos os nodos actuam de acordo com a decisão.

Deadlocks:

→ prevenção

Controlo de recursos centralizado – um processo é incumbido de ceder controlo a todos os recursos.
Ordenação de recursos – requer uma ordem total entre os recursos: cada recurso é numerado univocamente; um processo só pode requisitar um recurso se não tiver controlo sobre um recurso com valor maior.
Processos com prioridades e preempção: a cada processo é atribuída uma prioridade. Um processo mais prioritário ganha o controlo do recurso ao menos prioritário.
Processos com idade: cada processo tem uma data de nascimento.

- Wait-die: não preemptivo prioridade aos mais novos
- Would-wait: preemptivo prioridade aos mais velhos

→ Detecção

Algoritmo centralizado

- Cada nodo mantém um grafo de espera local. Os vértices do grafo correspondem aos processos que detêm ou pretendem recursos do nodo.
- Um processo coordenador solicita periodicamente os grafos de espera locais e construi um global com vértices para todos os processos no sistema.
- Qualquer ciclo no grafo global é um deadlock.

Algoritmo distribuído

- Cada nodo mantem um grafo de espera local. Os grafos locais têm um vertice adicional Pex extra.
- Se o grafo local tiver um ciclo que não envolva Pex então há um deadlock.
- Se o grafo local tiver um ciclo que envolva Pex então há provavelmente um deadlock. Inicia-se um algoritmo de detecção distribuído.

Acordo e Eleição

Determinar quando e onde um novo coordenador é necessário. Cada processo tem uma prioridade única e distinta. O coordenador é sempre o processo com mais alta prioridade.

Algoritmo Bully

- O algoritmo é despoletado por um processo Pi que julga que o coordenador falhou.
- Pi envia uma mensagem de eleição a todos os processos com maior prioridade que Pi.
- Se num intervalo T Pi não receber qq resposta, então auto-elege-se coordenador.
- Se receber alguma resposta então Pi espera durante T' por uma mensagem do novo coordenador.
- Se não receber nenhuma mensagem então reinicia o algoritmo.
- Se Pi não é o coordenador então, a qualquer momento pode receber uma mensagem...

Algoritmo em anel

- Aplicável a sistemas organizados física ou logicamente em anel.
- Assume que os canais de comunicação são unidireccionais.
- Cada processo mantém uma lista de activos que consiste nas prioridades de todos os processos activos quando este algoritmo terminar.
- Quando um processo Pi suspeita a falha do coordenador, Pi cria uma lista de activos vazia, envia uma mensagem de eleição m(i) ao seu vizinho e insere i na lista de activos.
- Se Pi recebe uma mensagem de eleição m(j) responde de uma de três formas:
    - Se foi a primeira mensagem de eleição que viu, então cria uma lista de activos com i e j. Envia as mensagens de eleição m(i) e m(j) (nesta ordem) ao vizinho.
    - Se $i \neq j$ então junta j à sua lista de activos e reenvia a mensagem ao vizinho.
    - Se $i=j$ então a sua lista de activos já contem todos os processos activos no sistema e Pi pode determinar o coordenador.

Aula 1

```
import java.lang.Thread;
import java.util.*;


class Contador
{
        public long i=0;

        public void inc()
        { i+=100000000; }
}


class HelloRunnable implements Runnable
{
  Contador cont;

  public HelloRunnable (Contador cont)
  {
    this.cont = cont;
  }

  public void run ()
  {
    cont.inc();
    System.out.println ("incrementei p/ " + cont.i);
  }
}

public class app
{
  public static void main (String args[])
  {
    Contador c = new Contador();

    for (int i = 0; i < 10; i++)
      {
        try
        {
          Thread t = new Thread (new HelloRunnable (c));
            t.start ();
            t.join ();
        }
        catch (Exception e)
        {
          System.out.println ("hum?" + e.getMessage ());
        }
      }

    System.out.println ("incrementei p/ " + c.i);
}
}
```

Aula 2

Sincronização por operação

```
import java.lang.Thread;
import java.util.*;

class Conta
{

  public float saldo = 0;
}

class Banco
{

  ArrayList < Conta > contas = new ArrayList < Conta > ();

  public Banco ()
  {
    for (int i = 0; i < 10; i++)
      contas.add (new Conta ());
  }


  public synchronized void debito (int conta, float valor)
  {
    contas.get (conta).saldo -= valor;
  }

  public synchronized void credito (int conta, float valor)
  {
    contas.get (conta).saldo += valor;
  }

  public synchronized void transferencia (int source, int dest, float valor)
  {
    contas.get (dest).saldo += valor;
    contas.get (source).saldo -= valor;
  }

  public void mostra ()
  {
    for (int i = 0; i < 10; i++)
      System.out.println ("Conta " + i + " tem saldo " +
                          contas.get (i).saldo);
  }
}

class Agencia implements Runnable
{

  private Banco b;

  public Agencia (Banco b)
  {
    this.b = b;
  }

  public void run ()
  {
    // credito 1000euros a cada conta seguido de um debito de 700€
```

```
    for (int i = 0; i < 10; i++)
      {
        b.credito (i, 1000);
        b.debito (i, 700);
      }

    // transferencia dos 300euros restantes das primeiras 5 contas para as
ultimas

    for (int i = 0; i < 5; i++)

      b.transferencia (i, 5 + i, 300);



  }
}

public class app_banco
{

  public static void main (String args[])
  {
    Banco b = new Banco ();

      ArrayList < Thread > lista = new ArrayList < Thread > ();

      try
    {
      for (int i = 0; i < 1000; i++)
        {
          Thread t = new Thread (new Agencia (b));
            t.start ();
            lista.add (t);
        }

      for (Thread t:lista)
          t.join ();
    }
    catch (Exception e)
    {
      System.out.println ("Exepção: " + e.getMessage ());
    }
    b.mostra ();

  }
}
```

Sincronização por acesso a conta

```
import java.lang.Thread;
import java.util.*;

class Conta
{

  public float saldo = 0;

  public synchronized void debito (float valor)
  {
```

```
    saldo -= valor;
  }

  public synchronized void credito (float valor)
  {
    saldo += valor;
  }
}

class Banco
{

  ArrayList < Conta > contas = new ArrayList < Conta > ();

  public Banco ()
  {
    for (int i = 0; i < 10; i++)
      contas.add (new Conta ());
  }


  public void debito (int conta, float valor)
  {
    contas.get (conta).debito (valor);
  }

  public void credito (int conta, float valor)
  {
    contas.get (conta).credito (valor);
  }

  public synchronized void transferencia (int source, int dest, float valor)
  {
    contas.get (dest).credito (valor);
    contas.get (source).debito (valor);
  }

  public void mostra ()
  {
    for (int i = 0; i < 10; i++)
      System.out.println ("Conta " + i + " tem saldo " +
                          contas.get (i).saldo);
  }
}

class Agencia implements Runnable
{

  private Banco b;

  public Agencia (Banco b)
  {
    this.b = b;
  }

  public void run ()
  {
    // credito 1000euros a cada conta seguido de um debito de 700€
    for (int i = 0; i < 10; i++)
      {
        b.credito (i, 1000);
        b.debito (i, 700);
```

```
      }

    // transferencia dos 300euros restantes das primeiras 5 contas para as
ultimas

    for (int i = 0; i < 5; i++)

      b.transferencia (i, 5 + i, 300);



  }
}

public class app_banco
{

  public static void main (String args[])
  {
    Banco b = new Banco ();

      ArrayList < Thread > lista = new ArrayList < Thread > ();

      try
    {
      for (int i = 0; i < 1000; i++)
        {
          Thread t = new Thread (new Agencia (b));
            t.start ();
            lista.add (t);
        }

      for (Thread t:lista)
          t.join ();
    }
    catch (Exception e)
    {
      System.out.println ("Exepção: " + e.getMessage ());
    }
    b.mostra ();

  }
}
```

Capítulo 1

Pergunta 1

```
import java.lang.*;
import java.io.*;


class InToOut extends Thread
{

  BufferedReader in = new BufferedReader (new InputStreamReader (System.in));
  BufferedWriter out =
    new BufferedWriter (new OutputStreamWriter (System.out));

  public void run ()
  {
    try
    {
      String s;
      while (true)
        {
          s = in.readLine ();
          System.out.println (s);
          //out.write(s); <- Porque nao da?
        }
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }
  }

}

class CountSleep extends Thread
{

  int c = 0;

  public void run ()
  {
    while (true)
      {
        try
        {
          c++;
          this.sleep (1000);
          System.out.println (c + "\n");
        }
        catch (Exception e)
        {
          e.printStackTrace ();
        }
      }
  }
}

public class um
{
```

```
  public static void main (String args[])
  {
    try
    {
      Thread t1 = new InToOut ();
      Thread t2 = new CountSleep ();
        t1.start ();
        t2.start();
        t1.join ();
        t2.join();
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }
    System.out.println("Adios.");
  }
}
```

Pergunta 2

```
import java.util.*;
import java.lang.*;

class Contador
{

  private int c = 0;

  public void inc ()
  {
    c++;
  }

  public int get ()
  {
    return c;
  }
}

class Child extends Thread
{
  Contador c;

  public Child (Contador c)
  {
    this.c = c;
  }

  public void run ()
  {
    for (int i = 0; i <= 10000000; i++)
      c.inc ();
  }
}

public class dois
{
  public static void main (String args[])
  {
```

```
    ArrayList < Thread > ts = new ArrayList < Thread > ();
    Contador c = new Contador ();

      try
    {


      for (int i = 0; i <= 10; i++)
        {
          Thread t = new Child (c);
            ts.add (t);
            t.start ();
        }

      for (int i = 0; i <= 10; i++)
        {
          ts.get (i).join ();
        }
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }

    System.out.println (c.get ());
  }
}
```

Capítulo 2

Pergunta 1

```
import java.util.*;
import java.lang.*;

class Contador
{

  private int c = 0;

  public synchronized void inc ()
  {
    c++;
  }

  public int get ()
  {
    return c;
  }
}

class Child extends Thread
{
  Contador c;

  public Child (Contador c)
  {
    this.c = c;
  }
```

```
  public void run ()
  {
    for (int i = 0; i < 10000000; i++)
      c.inc ();
  }
}

public class dois
{
  public static void main (String args[])
  {
    ArrayList < Thread > ts = new ArrayList < Thread > ();
    Contador c = new Contador ();

      try
    {


      for (int i = 0; i < 10; i++)
        {
          Thread t = new Child (c);
            ts.add (t);
            t.start ();
        }

      for (int i = 0; i < 10; i++)
        {
          ts.get (i).join ();
        }
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }

    System.out.println (c.get ());
  }
}
```

Pergunta 2 3 4

```
import java.util.*;
import java.util.concurrent.locks.*;

class Conta
{

  private int saldo;
  private int num;
  private Lock lock = new ReentrantLock ();

  public Conta (int saldo, int num)
  {
    this.saldo = saldo;
    this.num = num;
  }

  public void credito (int val)
  {
    saldo += val;
  }
```

```
  public void debito (int val)
  {
    saldo -= val;
  }

  public int saldo ()
  {
    return saldo;
  }

  public void lock ()
  {
    lock.lock ();
  }

  public void unlock ()
  {
    lock.unlock ();
  }

}

class Banco extends Thread
{
  Random rand = new Random ();

  private ArrayList < Conta > cs;

  public Banco (ArrayList < Conta > cs)
  {
    this.cs = cs;
  }

  public void transf (int ori, int dest, int val)
  {
    Conta cori = cs.get (ori);
    Conta cdest = cs.get (dest);

    if (ori < dest)
      {
        cori.lock ();
        cdest.lock ();
        cori.debito (val);
        cdest.credito (val);
        cdest.unlock ();
        cori.unlock ();
      }

    else
      {
        cdest.lock ();
        cori.lock ();
        cori.debito (val);
        cdest.credito (val);
        cori.unlock ();
        cdest.unlock ();
      }
  }

  public void debito (int ori, int val)
  {
```

```
    Conta c = cs.get (ori);
    c.lock ();
    c.debito (val);
    c.unlock ();

  }

  public int balanco ()
  {
    int val = 0;

  for (Conta c:cs)
      {
        c.lock ();
        val += c.saldo ();
      }

  for (Conta c:cs)
      c.unlock ();

    return val;
  }
}
```

Capítulo 3

Pergunta 1

```
import java.util.*;
import java.util.concurrent.locks.*;

class Conta
{

  private int saldo;
  private int num;
  private Lock lock = new ReentrantLock ();
  private Condition positivo = lock.newCondition();

  public Conta (int saldo, int num)
  {
    this.saldo = saldo;
    this.num = num;
  }

  public void credito (int val)
  {
    saldo += val;
    if(saldo > 0)
            positivo.notifyAll();
  }

  public void debito (int val)
  {
    try{
    while(saldo - val < 0)
            positivo.wait();
    }
    catch(Exception e) {e.printStackTrace();}
```

```
    saldo -= val;
  }

  public int saldo ()
  {
    return saldo;
  }

  public void lock ()
  {
    lock.lock ();
  }

  public void unlock ()
  {
    lock.unlock ();
  }

}

class Banco extends Thread
{
  Random rand = new Random ();

  private ArrayList < Conta > cs;

  public Banco (ArrayList < Conta > cs)
  {
    this.cs = cs;
  }

  public void transf (int ori, int dest, int val)
  {
    Conta cori = cs.get (ori);
    Conta cdest = cs.get (dest);

    if (ori < dest)
      {
        cori.lock ();
        cdest.lock ();
        cori.debito (val);
        cdest.credito (val);
        cdest.unlock ();
        cori.unlock ();
      }

    else
      {
        cdest.lock ();
        cori.lock ();
        cori.debito (val);
        cdest.credito (val);
        cori.unlock ();
        cdest.unlock ();
      }
  }

  public void debito (int ori, int val)
  {
    Conta c = cs.get (ori);
    c.lock ();
```

```
    c.debito (val);
    c.unlock ();

  }

  public int balanco ()
  {
    int val = 0;

  for (Conta c:cs)
      {
        c.lock ();
        val += c.saldo ();
      }

  for (Conta c:cs)
      c.unlock ();

    return val;
  }
}
```

Barreira

```
import java.util.concurrent.locks.*;


public class Barreira
{

  private int n;
  private int count;
  private Lock l = new ReentrantLock ();
  private Condition barreira = l.newCondition ();

  public Barreira (int n)
  {
    this.n = n;
    this.count = 0;
  }

  public void esperar ()
  {
    l.lock ();
    count++;
    if (count > n){
      barreira.notifyAll ();
      count = 1;
    }

    try
    {
      while (count < n)
        barreira.wait ();
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }
    l.unlock ();
  }
}
```

BoundedBuffer

```
import java.util.*;
import java.util.concurrent.locks.*;

public class MyBoundedBuffer < T >
{
  private ArrayList < T > buffer;
  private int size;
  private int count;
  private Lock l = new ReentrantLock ();
  private Condition notEmpty = l.newCondition ();
  private Condition notFull = l.newCondition ();

  public MyBoundedBuffer (int size)
  {
    this.buffer = new ArrayList < T > (size);
    this.size = size;
    this.count = 0;
  }

  public void put (T obj)
  {
    l.lock ();
    try
    {
      while (count > size)
        notFull.wait ();
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }
    buffer.add (obj);
    count++;
    notEmpty.notify ();
    l.unlock ();
  }

  public T get (int pos)
  {
    l.lock ();
    try
    {
      while (count < 1)
        notEmpty.wait ();
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }
    T obj = buffer.get (pos);
    buffer.remove (pos);
    count--;
    notFull.notify ();
    l.unlock ();
    return obj;
  }

}
```

MySemaphore

```
import java.util.concurrent.locks.*;


class MySemaphore
{
        private int value;
        private Lock l = new ReentrantLock();
        private Condition c = l.newCondition();

        public MySemaphore(int value)
        {
                this.value = value;
        }

        public void up()
        {
                l.lock();
                value++;
                c.notify();
                l.unlock();
        }

        public void down()
        {
                l.lock();
                try{
                while(value < 1)
                        c.wait();
                }
                catch(Exception e){e.printStackTrace();}
                value--;
                l.unlock();
        }
}
```

Capítulo 5

EchoCliente

```
import java.io.*;

import java.net.*;

import java.util.*;


public class EchoClient
{

  public static void main (String args[])
  {
    try
    {

      BufferedReader keyboard =
        new BufferedReader (new InputStreamReader (System.in));
      BufferedReader in;
```

```
      PrintWriter out;
      String s;

      while (true)
        {
          s = keyboard.readLine ();
          Socket c = new Socket ("localhost", 6666);
            in =
            new BufferedReader (new InputStreamReader (c.getInputStream ()));
            out = new PrintWriter (c.getOutputStream (), true);

            out.println (s);
            System.out.println (in.readLine ());
        }

    }
    catch (Exception e)
    {
      e.getStackTrace ();
    }
  }
}
```

EchoServer

```
import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;


public class EchoServer
{
  public static void main (String args[])
  {
    try
    {
      ServerSocket s = new ServerSocket (Integer.parseInt (args[0]));
        s.setReuseAddress (true);
      Socket cliente;

      while (true)
        {
          cliente = s.accept ();

          BufferedReader in =
            new BufferedReader (new
                                InputStreamReader (cliente.getInputStream
                                                   ()));
          PrintWriter out =
            new PrintWriter (cliente.getOutputStream (), true);

          String l;

          while ((l = in.readLine ()) != null)
            {

              out.println (l);
            }
          System.out.println ("One down.");
          cliente.close ();
          sum = 0;
        }
```

```
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }

  }
}
```

SumClient

```
import java.io.*;
import java.net.*;
import java.util.*;

public class SumClient
{

  public static void main (String args[])
  {
    try
    {

      BufferedReader keyboard =
        new BufferedReader (new InputStreamReader (System.in));
      BufferedReader in;
      PrintWriter out;
      String s;
        s = keyboard.readLine ();
      Socket c = new Socket ("localhost", 6666);
        in = new BufferedReader (new InputStreamReader (c.getInputStream ()));
        out = new PrintWriter (c.getOutputStream (), true);

      while (!s.equals ("done"))
        {


          out.println (s);
          s = keyboard.readLine ();
        }
      out.println ("done");
      System.out.println ("sum: " + in.readLine ());
    }
    catch (Exception e)
    {
      e.getStackTrace ();
    }
  }
}
```

SumServer

```
import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;


public class SumServer
{
  public static void main (String args[])
  {
    try
```

```
    {
      ServerSocket s = new ServerSocket (Integer.parseInt (args[0]));
        s.setReuseAddress (true);
      Socket cliente;
      Integer sum = 0;

      while (true)
        {
          cliente = s.accept ();

          BufferedReader in =
            new BufferedReader (new
                                InputStreamReader (cliente.getInputStream
                                                   ()));
          PrintWriter out =
            new PrintWriter (cliente.getOutputStream (), true);

          String l;

          while (!(l = in.readLine ()).equals ("done"))
            {
              sum += Integer.parseInt (l);
            }
          out.println (sum.toString ());
          cliente.close ();
        }
    }
    catch (Exception e)
    {
      e.printStackTrace ();
    }

  }
}
```

Online Banking

```
import java.util.*;
import java.util.concurrent.locks.*;
import java.net.*;
import java.io.*;

public class OnlineBanking
{


        public static void main(String args[])
        {
                Socket client;
                ArrayList<Conta> contas = new ArrayList<Conta>(10);
                Banco b = new Banco(contas);

                try{
                ServerSocket server = new
ServerSocket(Integer.parseInt(args[0]));

                while(true)
                {
                        client = server.accept();
                        BufferedReader in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
                        PrintWriter out = new
```

```
PrintWriter(client.getOutputStream(),true);
                        String s = in.readLine();
                        out.println(b.threat(s));
                        client.close();

                }
                }
                catch(Exception e){e.printStackTrace();}
        }

}
```