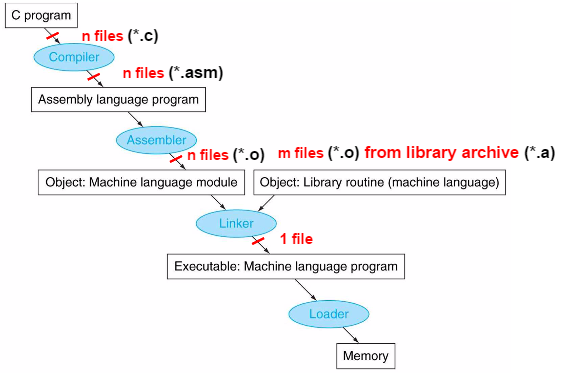
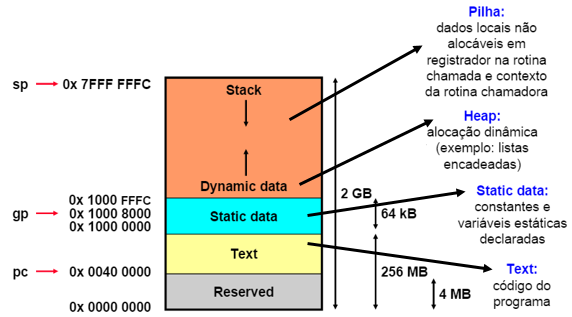
**Organização de Computadores**

-Cadeia de ferramentas



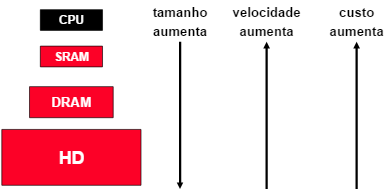
-Uso de memória: layout



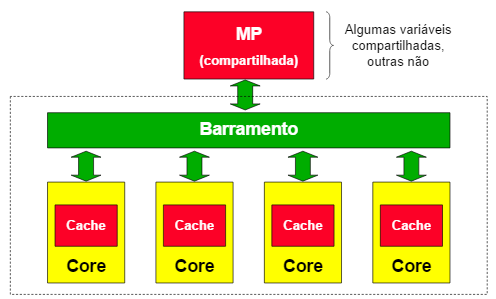
-Princípio da Localidade

* Temporal:
  + ‘Um item tende a ser novamente referenciado em breve. ’
    - Laços
    - Procedimentos
* Espacial:
  + ‘Itens com endereços próximos a um item referenciado tendem a ser referenciados em breve. ’
    - Instruções
    - Estruturas c/ elementos contíguos (arranjos, etc.)

-Hierarquia de memoria



-CMP: ‘on-chip multiprocessing’



**Sistemas Operacionais I**

-Principais gerenciadores do SO

* Gerenciador de memória, processador, dispositivos e arquivos.

-Tipos de SO

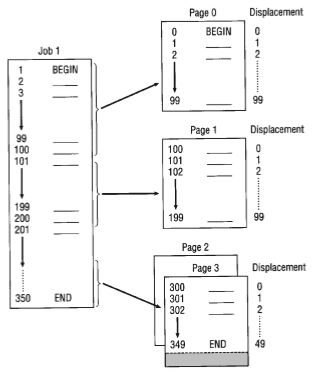
* Sistemas ‘em lote’ (batch), interativos, de tempo real, híbridos e embarcados.

-História dos SOs

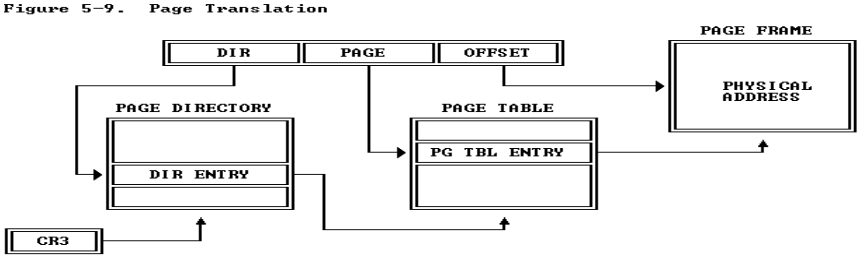
* Primeira geração
  + Tubos de vácuo; Engenheiros fazem tudo; Reservas de tempo para uso dos computadores; Computadores ociosos entre reservas.
* Segunda geração
  + Transistores; separação de operadores, programadores, pessoal de manutenção; ‘mainframes’; ‘Jobs’; ciclos de carregar cartão, rodar, impressão, fim; carrega cartões em uma fita, imprime em outra fita.
  + Conceitos de ‘**bootstrap’**, monitor, ‘**spool’** (utilizar um dispositivo rápido como intermediário para um dispositivo lento), ...
* Terceira geração
  + Circuitos integrados; carregar diversos programas de uma vez; um programa usa I/O, outro usa CPU.
  + Multiprogramação passiva, CPU espera interrupções dos programas requisitando uso de I/O ou indicando o fim do programa.
  + Multiprogramação ativa, cada programa usa uma fatia de tempo predefinido da CPU. Quando o tempo termina, o processo é interrompido e outro processo começa a executar.
  + Memória Virtual, não é mais necessário que todo o programa esteja na memória.
* Quarta geração
  + Multiprocessores; ‘network’; gerenciador de dispositivos; WWW.
  + Network SO: Cada máquina roda seu próprio SO e tem seus próprios usuários. Permitem controle remoto do sistema.
  + SOs Distribuídos: Usuários podem pensar que estão trabalhando com um sistema típico de 1 processador, quando na verdade, estão conectados em um ‘cluster’ de muitos processadores trabalhando juntos.
* Quinta geração
  + Multicore; dispositivos moveis; processadores embarcados.
  + Virtualização, criação de partições em um único servidor, com cada partição suportando um SO diferente.

-Esquemas de memória

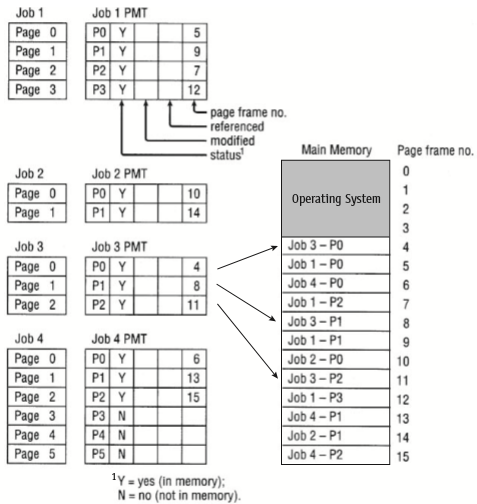
* Single-User Contiguous Scheme
  + Cada programa para ser processado, é carregado inteiramente na memória e aloca-se tanto espaço continuo na memória quanto necessário.
* Partições fixas
  + Uma partição é dada para cada processo.
  + O tamanho das partições é designado quando o sistema é ligado. Uma vez que o sistema esteja em operação, o tamanho das partições permanece estático.
  + Também requer que o programa esteja totalmente e continuamente alocado na memória.
  + Causa **fragmentação interna**.
* Partições dinâmicas
  + Cada processo recebe somente a quantidade de memória necessária para rodar.
  + Também requer que o programa esteja totalmente e continuamente alocado na memória.
  + Causa **fragmentação externa**.
* Paginação
  + Programas são divididos em **páginas** de tamanho **fixo** e devem ser alocados em ‘**page frames’** na memória.
  + Ainda requer que todo o programa esteja na memória, mas agora ele não precisa estar continuo.
  + Faz uso de tabelas para manter informações sobre os page frames e a tradução da página para page frame.



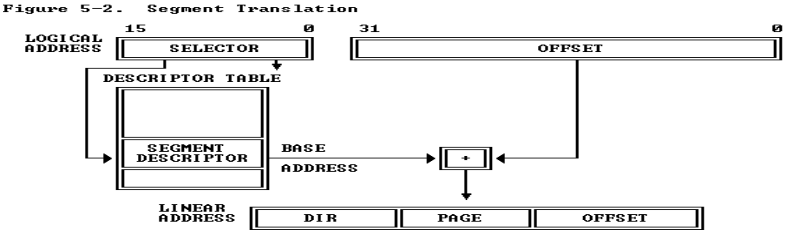
* + A chave para o sucesso deste esquema é o tamanho das páginas. Um tamanho de página muito pequeno gerará ‘PMTs’ muito longas enquanto tamanhos muito grandes vão resultar em fragmentação interna excessiva.



* Paginação sob demanda
  + Esquema que carrega apenas uma parte do programa na memória principal para processamento.

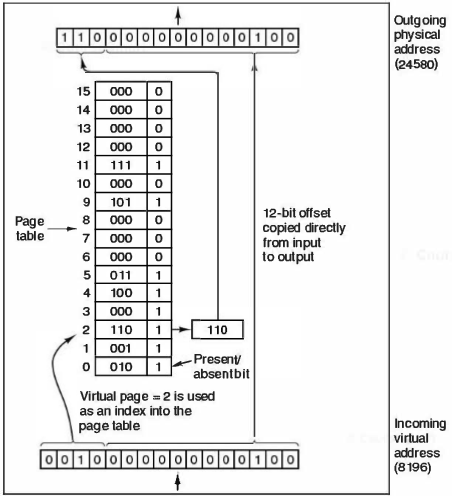


* + Quando uma página requisitada não se encontra na memória, o Gerenciador de Memória tem que gerar uma interrupção e chamar o ‘**Page fault handler’**.
  + O ‘Page fault handler’ determina se tem ‘page frames’ livres na memória para que então ele possa trazer imediatamente a pagina requisitada para a memória. Caso todos os ‘page frames’ estejam ocupados, ele deve decidir qual ‘page frame’ deve ser colocado no armazenamento secundário para abrir espaço para a página requisitada.
* Segmentação
  + Cada processo é dividido em vários **segmentos** de tamanhos **diferentes**.
  + Diferentemente da paginação, a memória principal não é mais dividida em ‘page frames’ porque o tamanho de cada segmento é diferente do outro. Por isso, a memória é alocada de uma maneira dinâmica.
  + Como a memória é particionada dinamicamente, vai haver fragmentação externa na memória.

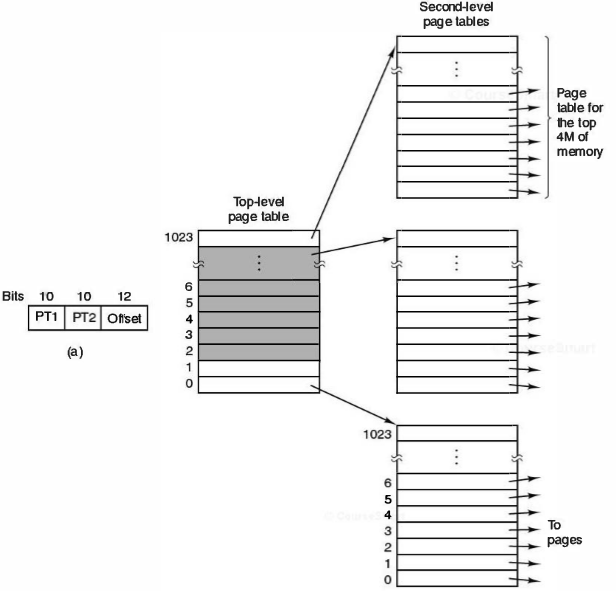


* Segmentação paginada
  + Este esquema de alocação não mantém cada segmento como uma única unidade continua, mas subdivide ele em páginas de tamanho igual, menores que a maioria dos segmentos, e mais facilmente manipuláveis do que um segmento inteiro.
  + Para acessar uma localização na memória, o sistema precisa localizar o endereço, que é composto de 3 entradas:
    - SEGMENT\_NUMBER & PAGE\_NUMBER & DISPLACEMENT

-Mapeamento feito pela MMU

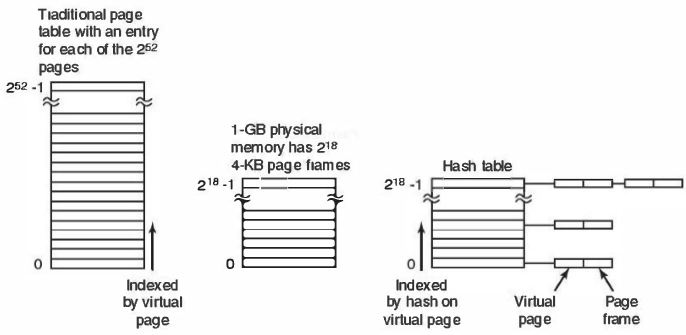


-Tabelas de páginas de multinível



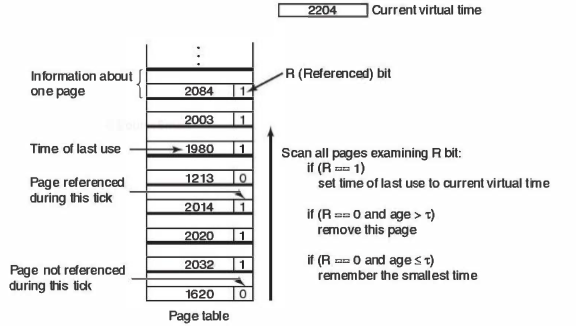
-Tabela de páginas invertida

* Há apenas uma entrada por ‘page frame’ na memória real, em vez de uma entrada por pagina do espaço de endereços virtuais.
* Este método salva grandes quantidades de memória, mas agora para achar um endereço físico, é preciso percorrer toda a tabela para encontra-lo.
* Isto pode ser contornado, usando uma **TLB**.
* Para acelerar a procura por endereços quando ocorrer uma falta na TLB, também pode-se usar uma ‘**Hash Table’.**



-Algoritmos de substituição de pagina

* The Not Recently Used Page Replacement Algorithm
  + Quando uma falta de página ocorre, o SO inspeciona todas as páginas e as divide em 4 categorias baseando-se nos valores dos **bits R e M**:
    - Classe 0: não referenciada, não modificada
    - Classe 1: não referenciada, modificada
    - Classe 2: referenciada, não modificada
    - Classe 3: referenciada, modificada
  + O algoritmo **NRU** remove uma página randomicamente dentre as classes não vazias de número mais baixo.
* The First-in, First-Out (**FIFO**) Page Replacement Algorithm
  + O SO mantém uma lista de todas páginas atualmente na memória, com as páginas mais recentes na ‘cauda’ da lista e as mais antigas na ‘cabeça’ da lista.
  + Quando uma falta de página ocorre, a página na ‘cabeça’ da lista é removida e a nova página é adicionada ao final da lista.
* The Second-Chance Page Replacement Algorithm
  + É uma simples modificação do FIFO, onde o algoritmo verifica o bit R da página na ‘cabeça’ da lista, caso este bit seja 0, a página é removida imediatamente. Caso contrário, seu bit R é setado para 0 e a página é colocada no final da lista e então a procura por páginas continua.
* The Clock Page Replacement Algorithm
  + É uma simples modificação do algoritmo acima, onde a lista agora é uma lista circular.
* The Least Recently Used (**LRU**) Page Replacement Algorithm
  + Um jeito de implementar este algoritmo é tendo um **contador** de 64-bits incrementado automaticamente após cada instrução. Além disso, cada entrada da tabela de páginas deve ter um campo longo o suficiente para conter o contador. Quando ocorre uma falta de página, o SO examina todos os contadores na tabela de página e encontra o menor. Esta página então é a escolhida para ser removida.
  + Outro jeito de implementar este algoritmo é usando uma **matriz**. Para uma máquina com n ‘page frames’, o hardware do LRU mantém uma matriz de n X n bits, inicialmente todos setados para 0. Quando um ‘page frame’ k é referenciado, o hardware primeiro seta todos os bits da linha k para 1, então seta todos os bits da coluna k para 0. A qualquer instante de tempo, a linha com o menor valor binário é a menos recentemente usada.
* Simulating LRU in Software
  + Uma solução usando software é o algoritmo chamado **NFU** (Not Frequently Used). Este algoritmo requer um contador em software para cada página, inicializado em 0. Para cada página, o bit R é adicionado ao contador. Quando uma falta de pagina acontece, a página com o menor contador é escolhida para ser removida.
    - O principal problema desta técnica é que ela nunca esquece de nada.
  + Com uma pequena modificação ao NFU, é possível simular o LRU. Primeiro, os contadores são deslocados para a direita 1 bit antes do bit R ser adicionado. Depois, o bit R é adicionado ao bit mais à esquerda em vez do bit mais à direita.
    - O algoritmo é conhecido como ‘**Aging’**.
    - Quando ocorre uma falta de página, a página com o menor contador é removida.
* The Working Set Page Replacement Algorithm
  + A ideia básica deste algoritmo é encontrar uma página que não esteja no ‘**Working Set’** e remover ela.
  + A quantidade de tempo de CPU que um processo realmente usou desde que ele começou é chamado ‘current virtual time’.
  + O ‘working set’ de um processo é o conjunto de páginas que ele referenciou durante os últimos τ segundos de tempo virtual.
  + Cada entrada contém, pelo menos, dois itens chaves de informação: o tempo, aproximado, da última referência à página e o bit R.
  + A ‘**idade’** de uma página é o: (tempo virtual atual – o tempo de ultimo uso da página).



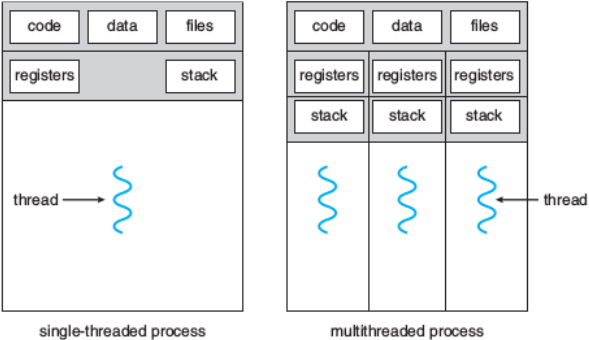
* The WSClock Page Replacement Algorithm
  + É necessária uma lista circular de ‘page frames’, inicialmente vazia. Quando uma página é carregada, ela é adicionada à lista. Cada entrada contém o campo ‘Time of last use’, assim como o bit R e M.
  + A cada falta de página, a página apontada pela ‘mão’ do ‘relógio’ é checada primeiro. Se o bit R for igual a 1, o bit R é setado então para 0, a ‘mão’ do ‘relógio’ é avançada para a próxima página, e o algoritmo repete para esta página.
  + Agora se o bit R for igual a 0 e a ‘idade’ da página for maior que o τ e a página não foi modificada, ela pode ser retirada imediatamente. Por outro lado, se a página foi modificada, é feito então um ‘agendamento’ de escrita em disco para esta página e então a ‘mão’ do ‘relógio’ é movida e o algoritmo continua com a próxima página.
* Algoritmos de substituição de páginas local: correspondem a alocar cada processo em uma fração fixa de memória. Isto é, quando ocorrer uma falta de página, o algoritmo vai escolher uma página do próprio processo para ser retirada.
* Algoritmos de substituição de páginas global: alocam ‘page frames’ dinamicamente entre todos os processos executando. Com isso, o número de ‘page frames’ atribuído a cada processo varia no tempo.

-Processos

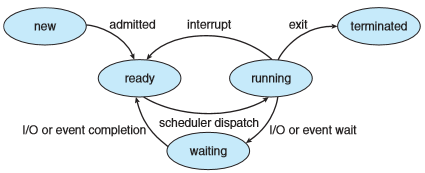
* Um processo suspenso consiste de seu espaço de endereçamento, geralmente chamado de ‘**core image’**, e sua entrada na tabela de processos, que contém o conteúdo dos seus registradores e muitos outros itens necessários para reiniciar o processo depois.
* Há quatro principais eventos que causam a **criação** de processos:
  + Inicialização do sistema.
  + Execução de uma chamada de sistema para criação de processos por um processo já em execução.
  + Um usuário requisita a criação de um novo processo.
  + Inicialização de um ‘batch job’.
* Há quatro condições para um processo **terminar**:
  + Término normal (voluntario)
  + Término por causa de erros (voluntario)
  + Erro fatal (involuntário)
  + Terminado por outro processo (involuntário)
* Suponha que um processo usa uma fração ‘p’ do seu tempo esperando por I/O. Com ‘n’ processos em memória de uma vez, a probabilidade de que todos os ‘n’ processos estejam esperando por I/O é de ‘p**n**’. A utilização da CPU é dada pela formula:
  + **Utilização da CPU** = 1 - p**n**

-Threads

* Processos são usados para agrupar recursos; threads são entidades escalonadas para execução na CPU.



* Há várias maneiras de implementar threads como:
  + Threads em **nível de usuário**, o programa em si cuida de suas threads.
  + Threads em **nível de Kernel**, o kernel do SO gerencia as threads.
  + Threads **hibridas**, usam threads em nível de kernel e então multiplexam threads de nível de usuário em alguma ou em todas as threads de nível de kernel.
* Enquanto um processo roda, ele muda de **estados**. O estado de um processo é definido em parte pela a atual atividade do processo. Um processo pode estar em um dos seguintes estados:
  + ‘New’. O processo está sendo criado.
  + ‘Running’. Instruções estão sendo executadas.
  + ‘Waiting’. O processo está esperando pela ocorrência de algum evento.
  + ‘Ready’. O processo está esperando para ser atribuído a um processador.
  + ‘Terminated’. O processo terminou sua execução.



-Escalonador e troca de contexto

* Uma **troca de contexto** basicamente consiste de:
  + Trocar de modo usuário para modo kernel.
  + Salvar o contexto do processo atualmente rodando na CPU.
  + O algoritmo de escalonamento deve escolher um novo processo para assumir a CPU.
  + A MMU deve ser carregada com os limites do novo processo.
  + E finalmente o novo processo deve ser inicializado.
  + E além disso, a troca de contexto normalmente invalida toda a memória cache, forçando o algoritmo a recarregar dinamicamente ela duas vezes da memória principal (uma na entrada do modo kernel e uma na saída).
* Um escalonador **‘não-preemptivo’** escolhe um processo para rodar e então deixa ele rodar até que ele se bloqueie (seja por I/O ou esperando outro processo) ou até que ele deixe a CPU voluntariamente.
* Um escalonador ‘**preemptivo’** escolhe um processo e deixa ele rodar por no máximo um tempo fixo. Se o processo ainda estiver rodando ao final do intervalo de tempo, ele é suspenso e o escalonador escolhe outro processo para rodar (se um outro estiver disponível).
* Um escalonador ‘preemptivo’ necessita que uma interrupção de relógio ocorra ao final do intervalo de tempo para dar controle da CPU de volta ao escalonador.

-Algoritmos de escalonamento

* Não-Preemptivos
  + First-Come First-Served
    - Os processos ‘ganham’ a CPU na ordem que eles requisitam ela.
  + Shortest Job First
    - A Job com menor ‘rajada de CPU’ roda antes.
    - O tempo estimado das Jobs é calculado com:
      * **Próxima estimativa** = α \* tempo estimado agora + (1-α) \* tempo real executado
  + Priority
    - Processos com maior prioridade rodam primeiro.
* Preemptivos
  + Round-Robin
    - Cada processo recebe um intervalo de tempo, chamado ‘**quantum’**, durante o qual o é permitido executar.
    - Caso o processo não tenha terminado de executar antes do final do intervalo, ele é suspendido e um novo processo é escolhido para rodar.
    - Um problema deste algoritmo é a escolha do ‘quantum’. Deve-se manter uma certa proporção entre tempo de execução dos programas e tempo para troca de contexto.
  + Shortest Remaining Time Next
    - O processo com menor tempo restante roda antes.
    - Quando um novo processo chega, o escalonador verifica se o tempo de execução deste novo processo é menor que o tempo restante do processo atualmente rodando, caso o tempo seja menor, o escalonador faz a troca de contexto para o novo processo.
  + Priority
    - Processos com maior prioridade rodam primeiro.
    - Caso um processo com maior prioridade chegue, é feita a troca de contexto entre o processo atualmente rodando e o novo processo.
    - Para prevenir que um processo de alta prioridade rode **indefinidamente**, o escalonador pode diminuir a prioridade do processo atualmente rodando a cada interrupção do relógio.
  + Filas Multilevel
    - Existem várias filas de ‘classes de prioridade’ no sistema, cada fila tem um certo ‘quantum’ de tempo em que os processos nela rodam.
    - Quando um processo utiliza todo o seu quantum de tempo, ele é colocado em uma fila de uma classe de prioridade menor e consequentemente com um quantum maior.
* Escalonadores de tempo real
  + Os eventos que um sistema de tempo real deve responder podem ser categorizados como ‘**periódicos’** (ocorrendo em intervalos regulares) ou ‘**não periódicos’** (ocorrem de forma imprevisível).
  + Se existem ‘m’ eventos periódicos e um evento ‘i’ ocorre com período Pi e requer Ci segundos de tempo de CPU para lidar com cada evento, os ‘m’ eventos só podem ser gerenciados se:



* + Um sistema de tempo real que satisfaz este critério é dito ser **escalonavel**.
  + Algoritmos de escalonamento para sistema de tempo real podem ser estáticos ou dinâmicos. Os estáticos fazem as decisões de escalonamento antes do sistema iniciar, e o dinâmico faz as decisões em tempo de execução.
  + Rate-Monotoc Scheduling
    - Se um processo com baixa prioridade está rodando e um processo com alta prioridade fica disponível para rodar, o escalonador vai preemptar o processo com baixa prioridade. Ao entrar no sistema, cada processo periódico recebe uma prioridade inversamente proporcional ao seu **período**. Quanto menor o seu período, maior será sua prioridade.
    - A perda de Deadline é previsível e em ordem.
    - Este algoritmo tem uma limitação: a utilização da CPU é limitada, e nem sempre é possível de maximizar os recursos da CPU. O pior caso da utilização da CPU escalonando N processos é:



* + Earliest Deadline First Scheduling
    - Cada processo recebe uma prioridade de acordo com sua **deadline**. Quanto mais cedo é a deadline, maior a prioridade do processo.
    - Quando um processo se torna disponível para rodar, ele deve anunciar sua deadline para o sistema.
    - Prioridades podem ser ajustadas para refletir a deadline de novos processos disponíveis para rodar.
    - A perda de Deadline é imprevisível e em cascata.

-Troca de mensagens

* Este método de comunicação entre processos usa duas primitivas, **send** e **receive**.
  + send(destination, &message);
  + receive(source, &message);

-Design Issues for Message-Passing Systems

* Para se proteger contra a **perda** **de mensagens**, o remetente e o destinatário podem concordar que assim que receberem a mensagem, o destinatário deve mandar de volta uma mensagem especial de reconhecimento (**ACK**). Se o remetente não receber o ACK dentro de um certo intervalo de tempo, ele irá retransmitir a mensagem.
* Para se proteger da perda do ACK, as mensagens devem possuir um tipo de **ID**, uma sequência de números consecutivos. Se o destinatário receber uma mensagem com um ID igual ao da última mensagem que ele atendeu, ele simplesmente manda de novo o ACK [só precisa comparar com a última?].
* Sistemas de mensagens também devem lidar com a pergunta de como processos são nomeados, para que o processo especificado no send e receive não é ambíguo.
* Autenticação também é um problema para sistemas de mensagens: como o cliente tem certeza que está se comunicando com o verdadeiro server, e não com um impostor?

-How messages are addressed

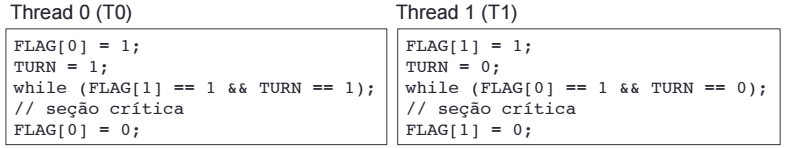
* Um jeito é atribuir um endereço único para cada processo e então fazer as mensagens serem endereçadas a processos.
* Outro jeito diferente é de inventar uma nova estrutura de dados, chamada de **mailbox**. Uma mailbox é um lugar para armazenar um certo número de mensagens, tipicamente especificado quando a mailbox é criada. Quando mailboxes são usadas, o parâmetro de endeço das chamadas send e receive são mailboxes, não processos. Quando um processo tenta enviar para uma mailbox que está cheia, ele é suspenso até que uma mensagem seja removida da mailbox.

-Deadlock e Livelock

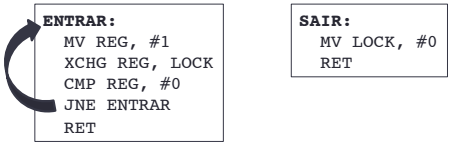
* **Deadlock** pode ser definido formalmente como:
  + Um conjunto de processos está em ‘deadlock’ se cada processo no conjunto está esperando por um evento que apenas outro processo no conjunto pode causar.
* **Condições** para Deadlock
  + Condição de exclusão mútua
  + Condição ‘hold and wait’
  + Condição de não preempção
  + Condição de espera circular
* Estratégias para **lidar** com Deadlocks
  + Apenas ignore o problema
  + Detecção e recuperação. Deixe deadlocks ocorrerem, detecte eles, toma uma ação.
  + Evitação dinâmica alocando os recursos cuidadosamente.
  + Prevenção, negando estruturalmente uma das quatro condições requeridas.
* **Livelock** é uma situação aonde dois ou mais processos continuamente mudam seus estados em resposta a mudanças em outro(s) processo(s) sem fazer nada de útil. Isto é parecido com Deadlock, mas a diferença é que os processos estão sendo educados e estão deixando outros fazerem o trabalho. Isto pode acontecer quando um processo tenta evitar um Deadlock.

-Sistemas de **sincronização**

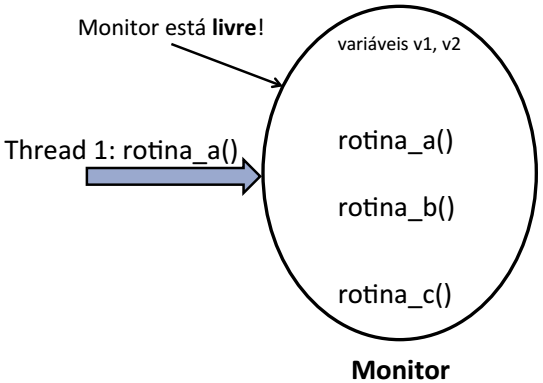
* Desativar interrupções
  + Problemas:
    - Desempenho: periféricos não podem ser atendidos.
    - Não funciona em maquinas paralelas.
* Algoritmo de **Peterson**



* + Problemas:
    - Espera ocupada.
    - Complexidade: Fácil para 2 thread, mas e para mais?
* **Spin-lock**

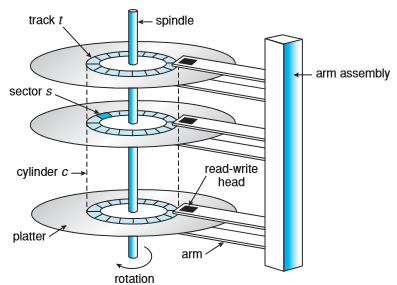


* + Problemas:
    - Espera ocupada.
* **Mutex**
  + É um tipo abstrato de dados composto de:
    - Um valor lógico
    - Uma fila de threads: thread bloqueadas nesta fila não fazem uso da CPU
  + Uma variável do tipo Mutex pode assumir um dos 2 valores: livre ou ocupado
  + Duas operações são permitidas em um Mutex “m”: lock(m) e unlock(m).
* **Semáforos**
  + É um tipo abstrato de dados composto de:
    - Um valor inteiro não negativo;
    - Uma fila de threads (FIFO)
  + Duas operações são permitidas em um semáforo “S”:
    - P(S) ou Down(S) → passar
    - V(S) ou Up(S) → liberar
* **Troca de mensagens**
  + Permite a comunicação entre processos sendo executados em computadores diferentes
  + Funções de comunicação:
    - send(msg, destinatário) e receive(msg, remetente).
* **Monitor**
  + É uma coleção de rotinas, variáveis e estruturas de dados agrupadas em um tipo especial de ‘modulo’ ou ‘pacote’
  + Somente uma thread pode estar ativa em um monitor em um dado momento
    - As demais threads aguardam em uma fila para acessar o monitor

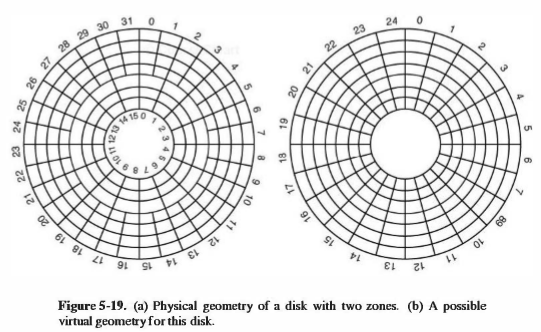


-Discos rígidos

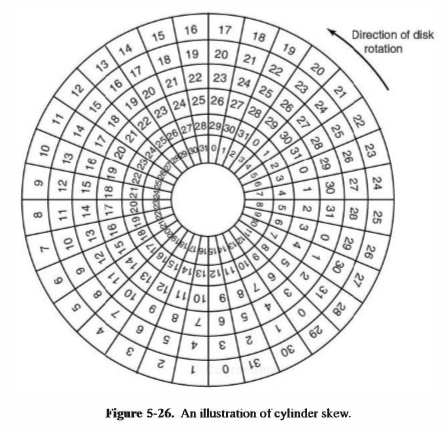
* Discos modernos são endereçados como um grande conjunto unidimensional de ‘**blocos lógicos’**, aonde o bloco lógico é a menor unidade de transferência. O tamanho de um bloco lógico é normalmente de **512bytes**, mas podem ser de 1KB.
* O conjunto unidimensional de blocos lógicos é mapeado sequencialmente nos setores do disco. O setor 0 é o primeiro setor da primeira ‘trilha’ no cilindro mais externo do disco.



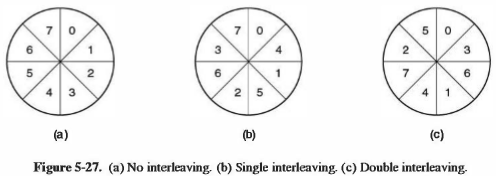
* Sempre que um processo precisa de I/O de/para o disco, ele chama uma chamada de sistema do SO. A requisição especificá várias informações:
  + Se a operação é de escrita ou leitura de dados.
  + Qual é o endereço do disco para a transferência
  + Qual é o endereço de memória para a transferência
  + Qual é o número do setor para ser transferido
* Se o driver de disco e seu controlador estão disponíveis, a requisição pode ser servida imediatamente. Se o driver de disco ou o controlador estiverem ocupados, toda nova requisição é colocada em uma lista de requisições **pendentes** para o disco.
* A estrutura de dados para um setor tipicamente consiste de um **cabeçalho**, uma **área de dados** (geralmente 512bytes), e um ‘**trailer’**. O cabeçalho e o ‘trailer’ contêm informações usadas pelo controlador de disco, como o número do setor e um código de correção de erro (**ECC**).
* Antes que um disco possa guardar dados, ele deve ser dividido em setores que o controlador de disco possa ler e escrever. Este processo é chamado ‘**formatação de baixo nível**’, ou ‘**formatação física’**.
* Além disso, o SO precisa gravar sua própria estrutura de dados no disco. Isto é feito em dois passos. O primeiro passo é particionar o disco em um ou mais grupos de cilindros. O SO pode tratar cada partição como se fosse um disco separado. Por exemplo, uma partição pode ter uma cópia do código executável do SO, enquanto outra pode ter arquivos de usuário. O segundo passo é a **‘formatação logica’**, ou criação de um sistema de arquivos. Estas estruturas podem conter ‘mapas’ de espaços livres e ocupados e um diretório inicial vazio.
* Discos mais sofisticados são mais espertos na recuperação de blocos ruins. O controlador mantém uma lista de blocos ruins no disco. Esta lista é inicializada durante a formatação de baixo nível e atualizada durante a vida útil do disco. A formatação de baixo nível também deixa de lado setores de reserva não visíveis ao SO. O controlador pode ser mandado para trocar logicamente cada setor ruim por um setor de reserva. Este esquema é conhecido como ‘**sector sparing’** ou ‘**forwarding’**.
* Discos modernos são divididos em zonas com mais setores em zonas exteriores do que em zonas interiores.
* Para esconder os detalhes de quantos setores cada trilha tem, a maioria dos discos modernos tem uma geometria virtual que é apresentado ao SO. O software é instruído para agir como se tivessem *x* cilindros, *y* cabeçotes e *z* setores por trilha. O controlador então mapeia uma requisição **(x, y, z)** em um cilindro, cabeçote e setor real.



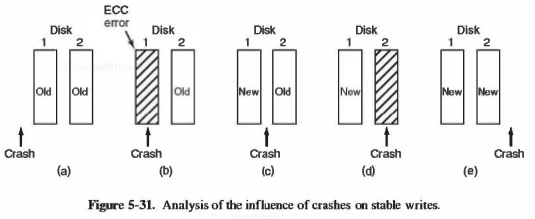
* A posição do setor 0 em cada trilha é deslocado da trilha anterior quando é feito a formatação de baixo nível. Este deslocamento, chamado de ‘**cylinder skew’**, é feito para improvisar performance. A ideia é de permitir que o disco leia múltiplas trilhas em uma única operação continua sem perder dados.



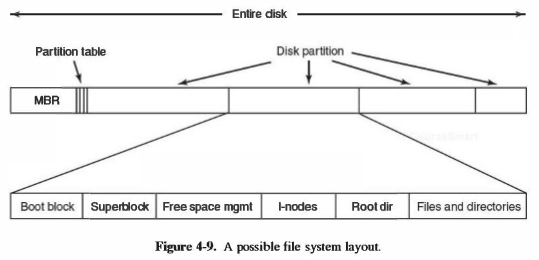
* A quantidade de ‘cylinder skew’ depende da geometria do disco.



* Para algumas aplicações, é essencial que os dados nunca sejam perdidos ou corrompidos, mesmo quando ocorrer erros de disco ou CPU. O que se pode fazer é um subsistema de disco que tem a seguinte propriedade: quando uma escrita é emitida, o disco ou escreve os dados corretamente ou não faz nada, deixando os dados existentes intactos. Tal sistema é chamado de ‘**stable storage’** e é implementado em software.



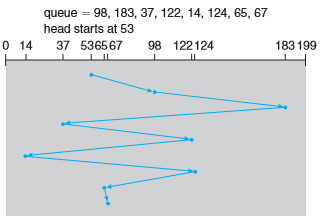
* O setor 0 do disco é chamado de **MBR (Master Boot Record)** e é usado para inicializar o computador. O final do MBR contém a tabela de partições. A tabela dá o endereço de início e de fim de cada partição. Uma partição da tabela é marcada como ativa. Quando o computador é inicializado, a **BIOS** lê e executa o MBR. A primeira coisa que o programa MBR faz é localizar a partição ativa, ler seu primeiro bloco, chamado de ‘**boot block’**, e executa ele. O programa no ‘boot block’ carrega o SO contido nesta partição. Por uniformidade, toda partição começa com um ‘boot block’, mesmo que ela não contenha um SO inicializável.
* Além de começar com um ‘boot block’, o layout de uma partição do disco varia muito de sistema de arquivo para sistema de arquivo. Frequentemente o sistema de arquivos vai conter alguns dos itens mostrados na figura abaixo:



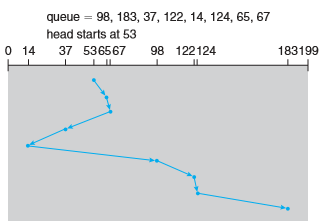
* O primeiro é o ‘**superblock’**. Ele contém todos os parâmetros chaves sobre o sistema de arquivos e é lido em memória quando o computador é inicializado ou o sistema de arquivo é primeiramente ‘tocado’. Informações típicas de um ‘superblock’ incluem o número magico que identifica qual é o tipo do sistema de arquivo, o número de blocos no sistema de arquivos e outras informações administrativas chaves.
* Em seguida pode vir informações sobre os blocos livres no sistema de arquivos. Isto pode ser seguido pelos ‘i-nodes’, um conjunto de estrutura de dados, um por arquivo, dizendo tudo sobre o arquivo. Após isso, pode vir o diretório ‘root’, que contém o topo da arvore do sistema de arquivos. Finalmente, o resto do disco contém todos os outros diretórios e arquivos.

-Algoritmos de escalonamento de disco

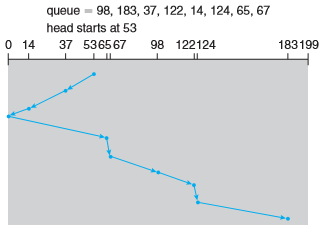
* First-come, first-served (**FCFS**) Scheduling
  + Este algoritmo é intrinsecamente justo, mas geralmente não é o que fornece o serviço mais rápido.



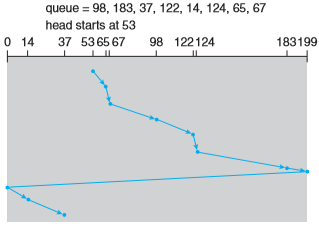
* + Como pode-se ver, este algoritmo faz com que o cabeçote se mova muito entre cilindros.
* Shortest-seek-time-first (**SSTF**) Scheduling
  + Este algoritmo seleciona a requisição com o menor tempo de busca da posição atual do cabeçote.
  + Este algoritmo pode causar ‘**starvation’**. O cabeçote tende a ficar no centro do disco.



* **SCAN** Scheduling
  + Neste algoritmo, o braço do disco começa em uma extremidade do disco e move-se em direção a outra extremidade, servindo as requisições assim que ele chega em cada cilindro, até ele chegar a outra extremidade. Na outra extremidade, a direção de movimento do cabeçote é revertida, e o ‘servimento’ continua.
  + Este algoritmo algumas vezes é chamado de **‘Elevator algorithm’**, já que ele funciona como um elevador em um prédio, primeiro servindo todas as requisições indo para cima e então revertendo para servir as requisições no outro sentido.
  + Se uma requisição chegar bem afrente do cabeçote, ela será servida quase que imediatamente; agora se uma requisição chegar atrás do cabeçote, ela terá que esperar até que o cabeçote se mova para o fim do disco, reverta sua direção, e volte em sua direção.



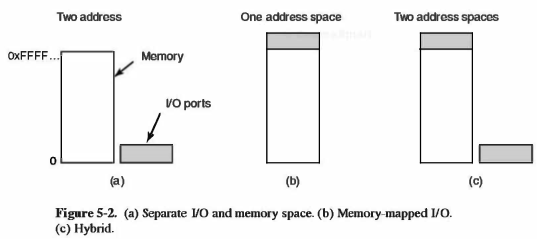
* Circular SCAN (**C-SCAN**) Scheduling
  + Este algoritmo é uma variação do SCAN designado para fornecer um tempo de espera mais uniforme. Como o SCAN, o C-SCAN move o cabeçote de uma extremidade para a outra, servindo as requisições pelo caminho. Quando o cabeçote chega a outra extremidade, entretanto, ele retorna imediatamente ao começo do disco sem servir nenhuma requisição na volta.



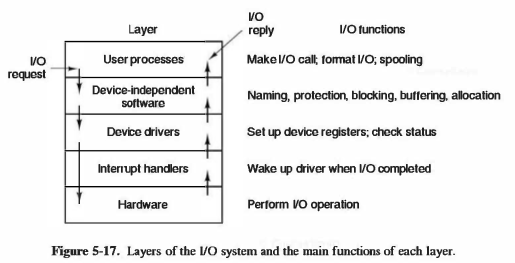
* **LOOK** Scheduling
  + Como descrito antes, ambos SCAN e C-SCAN movem o braço do disco através de todo o disco. Na pratica, nenhum dos dois algoritmos é implementado deste jeito. Normalmente o braço vai apenas até a última requisição em cada direção. Então, reverte a direção imediatamente, sem ir até a extremidade do disco. As versões do SCAN e C-SCAN que seguem este padrão são chamados de LOOK e **C-LOOK** Scheduling, porque eles procuram por uma requisição antes de continuar movendo em uma dada direção.

-Dispositivos

* Dispositivos de I/O podem ser divididos em duas categorias: **dispositivos de blocos** e **dispositivos de caracteres**.
* Um dispositivo de bloco guarda as informações em blocos de tamanho fixo, cada um com seu próprio endereço. A propriedade essencial de um dispositivo de bloco é que é possível ler ou escrever em cada bloco independentemente de todos os outros blocos. HDs, CD-ROMs e conectores de USB são exemplos desses dispositivos.
* Um dispositivo de caracter entrega ou recebe uma sequência de caracteres, sem levar em consideração nenhuma estrutura de bloco. Eles não são endereçáveis e não tem operações de busca. Impressoras e interfaces de Network são exemplos desses dispositivos.
* Há duas maneiras da CPU se comunicar com os dispositivos. Na primeira abordagem, cada registrador de controle é atribuído um número de ‘porta I/O’, um inteiro de 8 a 16bits. O conjunto de todas as portas de I/O formam um ‘**espaço de portas de I/O**’ e é protegido para que programas de usuários ordinários não possam acessar.
* A segunda abordagem é de mapear todos os registradores de controle no espaço de memória. Cada registrador de controle é atribuído um endereço de memória único no qual não é salvo nenhum dado de memória. Este sistema é chamado de ‘**memory-mapped I/O**’. Geralmente, os endereços usados estão no topo do espaço de endereçamento. É possível usar um sistema hibrido também, aonde os endereços de memória são usados para buffer de dados e as portas de I/O separadas são usados como registradores de controle. [acho que é isso...]

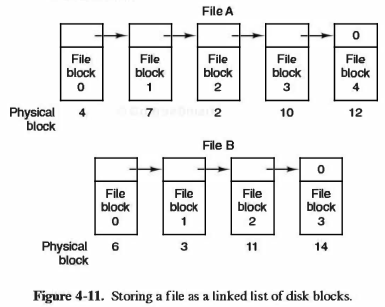


* A forma mais simples de I/O é fazer a CPU cuidar de todo o trabalho. Este método é chamado de ‘**programmed I/O’**.
* Neste esquema o SO fica checando o registrador de status do dispositivo e fica passando os dados para ele. Isto é chamado de ‘**polling’** ou **‘espera ocupada’**.
* Outra forma de I/O é usando **interrupções**. Nesta forma, a CPU manda o primeiro dado, e vai fazer outra coisa. Então quando o dispositivo estiver preparado para receber mais dados, ele gera uma interrupção.
* Uma desvantagem obvia do I/O usando interrupção é que as interrupções vão ser geradas a cada caracter passado ao dispositivo. Uma solução é usar o **DMA**. O DMA é que fica passando os dados em vez da CPU e só gera uma única interrupção ao final do processo.
* Cada dispositivo de I/O anexado a um computador precisa de algum código especifico para dispositivo para o controlar. Este código, chamado de ‘**device driver’**, é geralmente escrito pelo fabricante do dispositivo e entregue junto com o dispositivo.
* Cada driver de dispositivo normalmente cuida de um tipo de dispositivo, ou no máximo, uma ‘classe’ de dispositivos intimamente relacionados.
* A maioria dos SOs definem uma interface padrão que todos os dispositivos de bloco devem suportar e uma segunda interface padrão que todos os dispositivos de caracteres devem suportar. Estas interfaces consistem de um número de procedimentos que o resto do SO pode chamar para fazer o driver trabalhar para eles. Tipicamente os procedimentos são de ler um bloco (dispositivos de blocos) ou escrever um caracter (dispositivos de caracter).
* Camadas e suas principais funções:

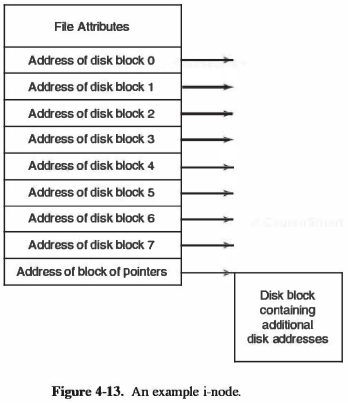


-Formas de alocação de arquivos no disco:

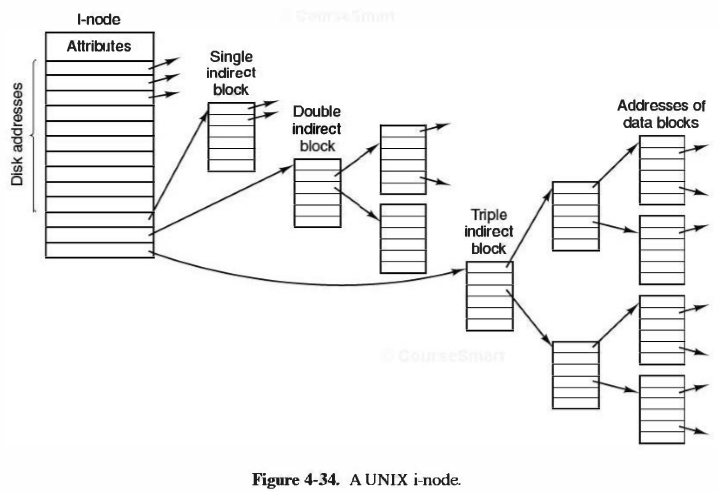
* Alocação contigua
  + O esquema de alocação mais simples é de guardar cada arquivo como uma contigua linha de blocos do disco.
* Alocação usando lista encadeada
  + O segundo método para guardar arquivos é de manter cada um como uma lista encadeada de blocos de disco. A primeira ‘word’ de cada bloco é usada como um ponteiro para o próximo bloco. O resto do bloco é usado para dados.



* Alocação usando I-Nodes (Arvore)
  + O último método para manter controle de qual bloco pertence a qual arquivo é de associar com cada arquivo uma estrutura de dados chamada um ‘**i-node’** (nodo de indexes), que lista os atributos e os endereços de disco dos blocos dos arquivos.

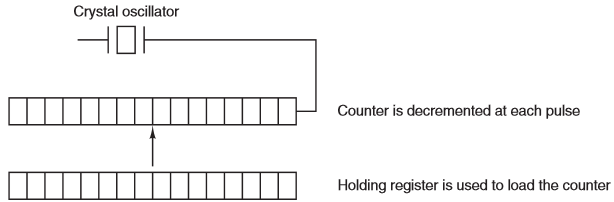


* + O problema deste esquema é que, a princípio, ele limita o tamanho máximo do arquivo. Para cuidar deste problema, deve-se utilizar uma estrutura de arvore, como mostrado pelo professor na aula e mostrado na imagem abaixo.

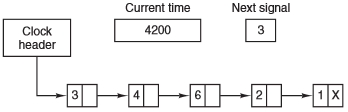


-Clocks/Timers

* Clock Hardware
  + Dois tipos:
    - Clocks ligados a linha de força de 110- ou 220-volt que causam interrupções a cada ciclo de voltagem, a 50 ou 60Hz.
    - Clocks construídos com 3 componentes: um **cristal** **oscilador**, um contador e um registrador de apoio, também chamados de clocks programáveis.
      * O contador do clock programável é feito para contar até zero e quando ele chega em zero, ele causa uma interrupção para a CPU.



* Clock Software
  + Em muitos sistemas, um processo pode solicitar que o sistema operacional lhe dê um aviso após um certo intervalo. O aviso geralmente é um sinal, **interrupção**, mensagem ou algo similar.
  + Se muitos sinais são esperados, é mais eficiente simular múltiplos clocks encadeando todas as solicitações juntos, ordenados pelo tempo, numa lista encadeada. Cada entrada da lista diz quantos ticks de clock seguindo o último elemento deve-se esperar antes de causar um sinal.
  + Na figura abaixo, a próximo interrupção deve ocorrer em 3 ticks. Em cada tick, Next Signal é decrementado. Quando ele chega a 0, o sinal correspondente ao primeiro elemento na lista é causado, e o elemento é removido da lista. Então o Next Signal é setado com o valor do elemento na cabeça da lista, neste exemplo, 4.



**Sistemas Operacionais II**

-Bootstrapping

* Computadores devem implementar um processo de ‘**bootstrap’** para carregar o sistema operacional
* Computadores modernos tipicamente utilizam ‘read-only Memory’ (**ROM**) contendo o código inicial para carregar o sistema operacional
  + Processadores são frequentemente implementados para iniciar executando um código em um **endereço especifico**, imediatamente após eles serem reiniciados
* O advento das ROMs revolucionou o ‘bootloading’
  + Fabricantes incluíram programas sofisticados na placa-mãe do computador para lidar com o processo de inicialização do sistema operacional
  + Software até incluem drivers básicos para disco, vídeo, etc.
* A combinação de memória persistente e o programa armazenado nela é chamado de **firmware**

-IA32 Bootstrap

* Processadores IA32 são projetados para iniciar executando instruções no endereço **0xFFFFFF0** imediatamente após uma reinicialização
* Fabricantes de computadores colocam uma ROM neste intervalo de endereços para iniciar o processo de boot
  + ROM é tipicamente muito maior que **16 bytes**
* CPUs IA32 também inicializam em modo de proteção 0 (**modo kernel**)
  + Isto permite que o bootloader tenha total acesso a todo o sistema para preparar o hardware para rodar o sistema operacional
* Atualmente existem duas categorias de bootloaders:
  + **PC BIOS** – Basic Input/Output System
  + **EFI/UEFI** – [Unified] Extensible Firmware Interface

-PC BIOS

* Firmware original para computadores x86
* Providencia duas características cruciais, e uma terceira útil:
  + Um firmware bootloader para iniciar o processo de bootstrap
  + Uma biblioteca básica de funções de input/output para interagir com o hardware do computador
  + Frequentemente inclui uma interface de usuário simples para a configuração do hardware
* Originada com o sistema operacional CP/M (1976)
  + Naquele ponto a BIOS era carregada do disco por um firmware
* O bootloader da BIOS segue um **processo** bem simples:
  + Se um reset do hardware foi feito, rode alguns diagnósticos no hardware
  + Identifique e configure periféricos do computador para uso básico
  + Percorra os dispositivos inicializáveis em alguma ordem, tentando carregar e inicializar a próxima etapa do processo de bootstrap
    - O primeiro setor de cada dispositivo inicializável é carregado na memoria
    - Se o setor terminar com a assinatura **0x55** ou **0xAA**, ele é utilizado como bootloader
* A BIOS carrega o setor de boot no endereço de memória **0x7C00** e então pula para este endereço
* O setor de boot que a BIOS carrega é de apenas **512 bytes !**
  + Historicamente, este era o tamanho dos setores do disco do x86
  + 0x200 bytes, então o bootloader está nos endereços **0x7C00 – 0x7DFF**
  + Tirando a assinatura, o setor de boot tem **510 bytes** para fazer o que precisa ser feito
* A BIOS passa alguns detalhes limitados para o bootloader
  + E.g. O registrador **%dl** contendo o ID numérico do disco de boot; isto permite que o bootloader obtenha mais dados do disco
* Bootloaders são geralmente escritos em linguagem assembly

-Bootloading e MBRs

* O primeiro setor de um disco rígido é um **master boot record** (MBR)
  + Especifica até **4 partições** do disco rígido, cada uma com seu próprio formato e uso
* Problema: MBR não corresponde a um sistema operacional especifico; o disco pode conter **múltiplos** SOs
* Uma partição contendo um sistema operacional frequentemente especifica seu próprio bootloader no primeiro setor da partição
* Os detalhes das partições no MBR também utilizam um pouco de espaço...
  + 4 partition-table entries x 16 bytes per entry = **64 bytes**
* Formato antigo de MBR:
  + 512 bytes – 2 bytes de assinatura – 64 bytes da tabela de partição = **446 bytes** para bootloading
* Novas MBRs incluem detalhes adicionais que reduzem o tamanho para **434-436 bytes**, partidos em duas partes.
* Nota: Bootloaders avançados como LILO e GRUB claramente não cambem neste espaço pequeno
  + O bootloader em si é quebrado em múltiplas **etapas**
    - Etapa 1 é de 512 bytes, e é responsável por carregar a Etapa 2 na memoria
    - Etapa 2 é muito maior, 100KB+
* O loader do MBR deve carregar o bootloader da partição...
* E deve fazer isso de um jeito que o bootloader da partição não saiba que ele não foi carregado diretamente pela BIOS
  + O loader do MBR deve emular o mecanismo de bootloading da BIOS
* MBR loaders usam um mecanismo chamado de ‘**chain loading’**
* MBR carrega o próximo bootloader no endereço **0x7C00** e então pula para este endereço e começa a roda-lo
  + Ele também mantém os valores dos registradores da BIOS, ou seja, o %dl possui o mesmo valor que a BIOS passo para o Loader do MBR
* Claro, o Loader do MBR também foi carregado no endereço 0x7C00...
  + Ele deve copiar a si mesmo para outra posição, e então pular para lá antes de continuar com suas próximas tarefas
  + ‘Chain loaders’ frequentemente se copiam para o endereço **0x0600**

-BIOS Labrary Functions

* Nesta fase inicial, os bootloaders dependem das funções da BIOS para interagir com o hardware do computador
* Os bootloaders possuem o mesmo problema das aplicações de usuário: eles não sabem os endereços de operações da BIOS
* Todas as funções da BIOS são invocadas via **interrupções de software**
* Exemplo: **int $0x10** é usada para serviços de vídeo

-IA32 Memory Addressing

* A arquitetura IA32 suporta diversos mecanismos diferentes de endereçamento de memoria
  + Alguns providenciam maneiras avançadas para os sistemas operacionais modernos gerenciarem memoria
  + Outros são necessários por motivos de compatibilidade com versões anteriores
* Uma consequência disso é que quando um Core i7 inicializa, é necessário que ele aja **exatamente** igual a um 8086

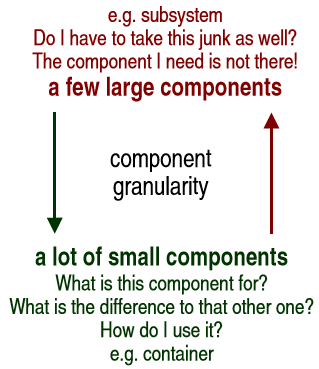
-IA32 Real-Addressing Mode

* O IA32 possui um modelo de memória segmentada
  + O que um “segmento” significa depende do modo de endereçamento de memória que o processador esta
* Inicialmente, processadores IA32 inicial no modo de endereçamento real (isto é, “**real mode**”)
* No modo de endereçamento real, o processador suporta um espaço de endereços físicos de **20-bit** (1MiB)
* Para gerar este endereço de 20 bits, o processador combina um valor de segmento de 16 bits e um valor de deslocamento de 16 bits
  + Endereço = segmento << 4 + deslocamento

-IA32 Protected Mode

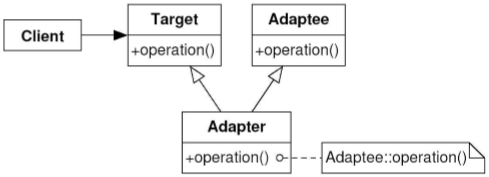
* O IA32 também possui o modo protegido de gerenciamento de memoria
* Modo protegido também usa os registradores de segmento
* No modo protegido, os seletores de segmento não são diretamente usados para computar endereços lineares
  + Os comportamentos dos dois modos são completamente diferentes
* Em vez disso, os seletores de segmentos são índices para descritores de segmentos na ‘Global Descriptor Table’ (**GDT**)
* Todos os acessos a memória no modo protegido do IA32 usam endereços lógicos
  + Segment\_selector : offset, aonde segment\_selector é de 16 bits e o offset é de 32bits
* Estes endereços lógicos são mapeados para endereços lineares usando os descritores de segmentos na tabela de descritores
  + Linear\_address = gdt[segment\_selector].base\_address + offset
* Este endereço linear é um endereço virtual; ele deve então ser mapeado para um endereço físico usando a estrutura de páginas de diretório
* Para **trocar para** o modo protegido, pelo menos uma tabela de descritores, a GDT, e dois descritores (code e data) devem ser criados.
* Endereços base e limite para a GDT devem ser carregados no registrador GDTR usando instruções LGDT
* Uma IDT e um ‘gate’ para o manipulador de interrupções NMI (Non Maskable Interrupts) precisam ser criados
* Endereços base e limite para a IDT devem ser carregados no registrador IDTR usando instruções LIDT

-Granularidade de componentes de software

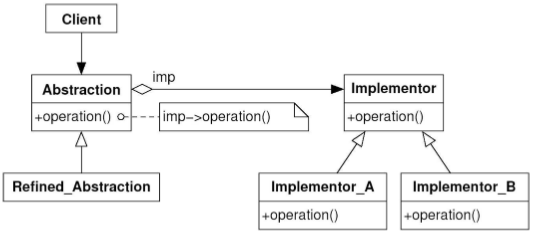


-Design Patterns

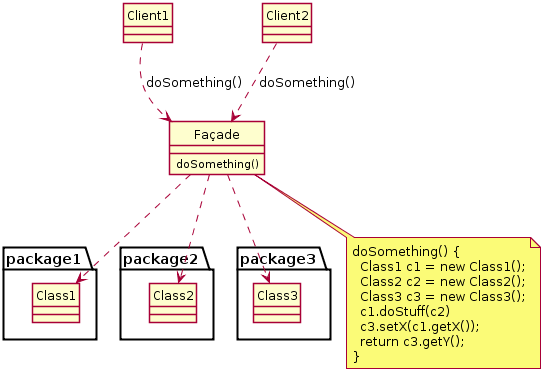
* São um catalogo de soluções para problemas recorrentes no cenário de orientação a objetos
* É uma descrição ou um template de como resolver um problema que pode ser usado em diversas situações.
* <https://en.wikipedia.org/wiki/Software_design_pattern>
* **Adapter** pattern
  + O padrão de projeto Adaptador permite que classes incompatíveis trabalhem juntas, convertendo a interface de uma classe na interface esperada pelos seus clientes.



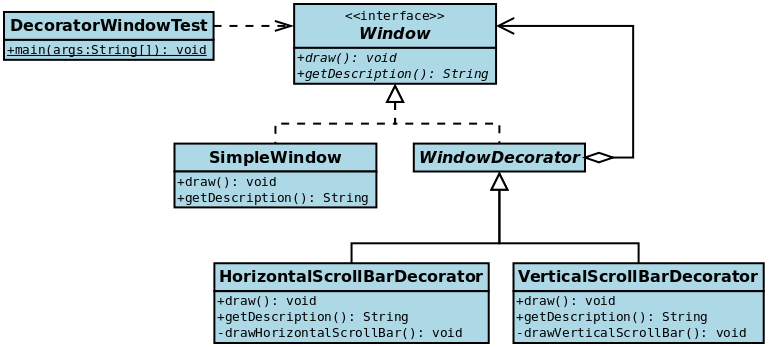
* **Bridge** pattern
  + O padrão de projeto Bridge é utilizado para desacoplar uma abstração de sua implementação para que as duas possam variar independentemente.



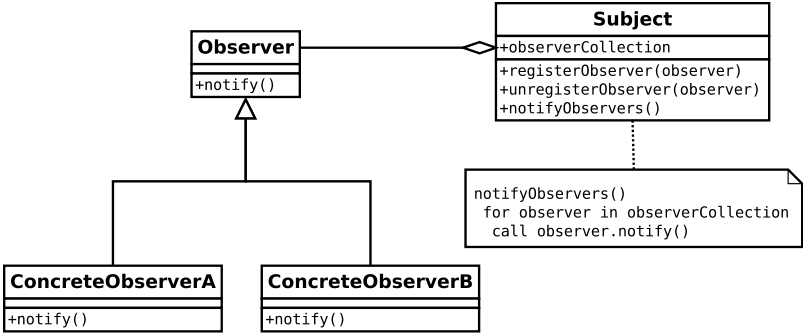
* Facade (façade) pattern
  + Um Façade é um objeto que provê uma interface simplificada para um corpo de código maior, como por exemplo, uma biblioteca de classes.
  + Este padrão esconde as complexidades de um sistema maior e provê uma interface simplificada ao cliente.



* Decorator pattern
  + Permite que comportamentos sejam adicionados a um objeto individual, estaticamente ou dinamicamente, sem afetar o comportamento de outros objetos da mesma classe.

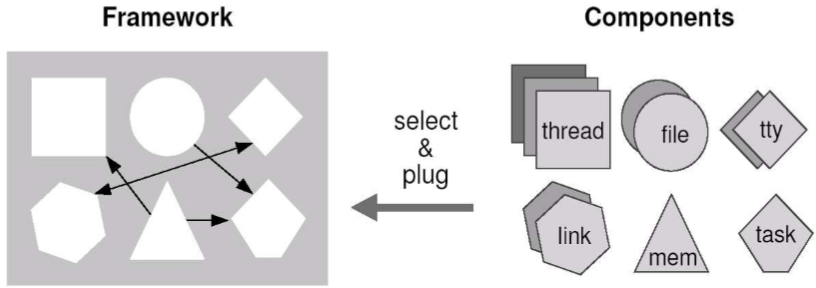


* Observer pattern
  + Define uma dependência um-para-muitos entre objetos de modo que quando um objeto muda o seu estado, todos os seus dependentes são notificados e atualizados automaticamente.
  + Permite que objetos interessados sejam avisados das mudanças de estados ou outros eventos que ocorrerem em um outro objeto.



-Frameworks

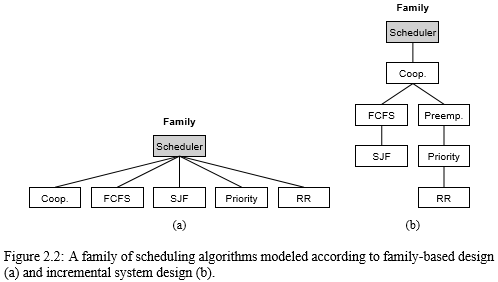
* Um Framework é um conjunto de classes com objetivo de reutilização de arquitetura de software, provendo um guia para uma solução em um domínio específico de software.
* Framework se diferencia de uma simples biblioteca, pois está se concentra apenas em oferecer implementação de funcionalidades, sem definir a reutilização de uma solução de arquitetura.
* Frameworks são constituídos de *frozen spots* e *hot spots*:
  + **Frozen spots** são as partes fixas de um framework, também conhecidos como *hook points*. São serviços já implementados pelo framework. Normalmente realizam chamadas indiretas aos hot spots.
  + **Hot stops** são as partes flexíveis de um framework. São pontos extensíveis, necessitam de complementação por funcionalidades/serviços que devem ser implementados.



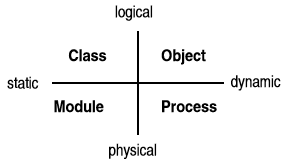
* **White box** framework: It based on inheritance of classes and overriding the methods baseclasses.
* **Black box** framework: It based on composition and delegation where delegation means that an object relies on another object to do some task, and these objects that we rely on are the black box framework.

-Design/Modeling/Programming

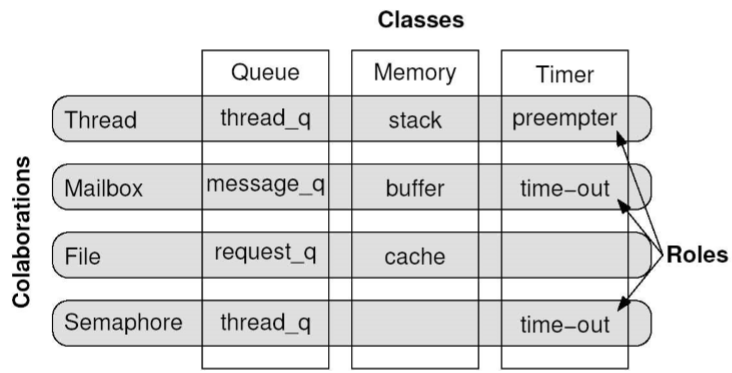
* Family-Based Design
  + É estabelecido em torno de dois conceitos complementares: ***commonality*** e ***variability***.
    - Commonality é o critério básico de agrupamento, entidades que compartilham aspectos em comum considerados relevantes para o projetista são agrupadas para formar famílias.
    - Inversamente, variability traz as diferenças que identificam cada um dos *membros* da família.
  + Este conceito engloba, mas não se restringe a, hierarquias de classes em projetos orientados a objetos, com uma *classe base* caracterizando a família (abstração) e subclasses capturando as variações que distinguem membros da família.



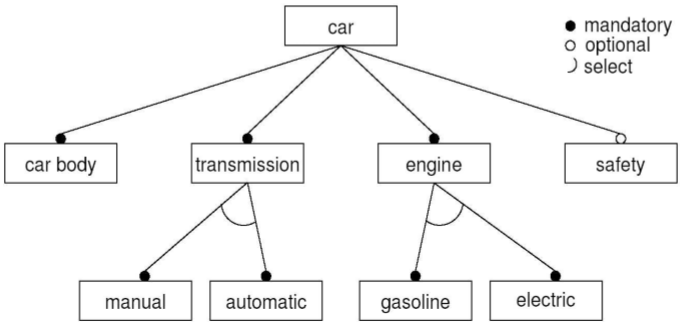
* + A figura acima mostra uma família de algoritmos de escalonamento modelados de acordo com: (a) o método original de Family-based design, e (b) a extensão a este método chamado de *incremental system design*. Como pode ser observado, o projeto incremental de sistemas tende a gerar hierarquias mais profundas, já que as variações são organizadas uma sobre a outra, com o mais primordial perto da raiz.
* Object-Oriented Design
  + Analise de domínio e decomposição
    - Objetos abstraem entidades do domínio
    - Commonality => classes
    - Variability => hierarquias de classe (subclasses)
  + Modelos:



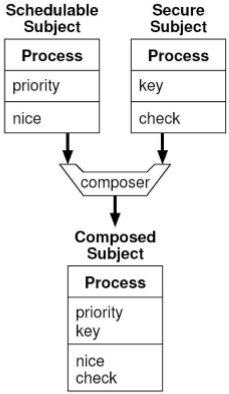
* Collaboration-Based Design
  + Estende o projeto orientado a objetos para expressar que um objeto pode desempenhar diferentes papéis em um sistema, e que uma série de papéis cooperantes (colaboração) pode ser uma melhor unidade de reuso e composição do que uma classe.
  + Uma *colaboração* pode ser definida como um conjunto de objetos e um protocolo de interação que especificam os seus papeis na colaboração. Da mesma forma, um *papel* pode ser interpretado como a parte de um objeto que reforça o protocolo de interação.



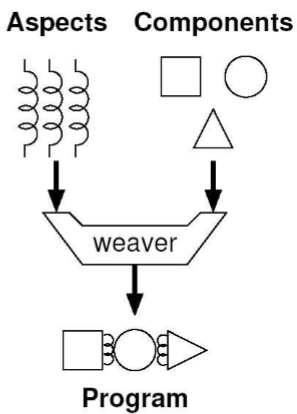
* **Feature**-Based Modeling
  + Características permitem que o processo de design seja abordado a partir de diferentes níveis de detalhe
    - Subcaracterísticas fornecem um método para ver características como uma agregação de várias, mais primitivas características



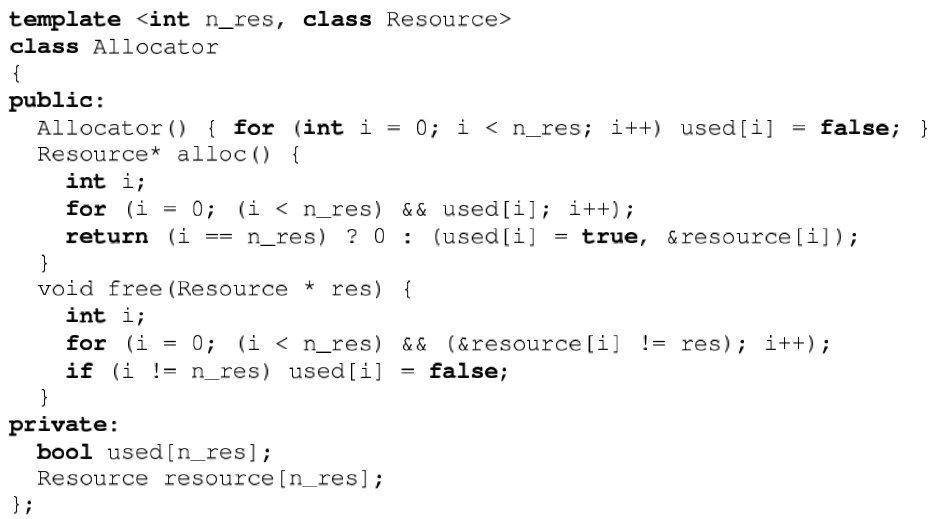
* **Subject**-Oriented Programming
  + Estende o projeto orientado a objetos para lidar com uma “multiplicidade de pontos e vista subjetivos” de objetos sendo modelados. Perspectivas subjetivas de um objeto originam do fato que algumas de suas propriedades podem ser mais interessantes para alguns programas do que para outros.
  + Na programação orientada ao sujeito, *sujeitos* são coleções de classes, ou fragmentos de classe, que modelam um ponto de vista subjetivo de um domínio. Estes pontos de vista subjetivos são mais tarde são reconciliados durante a composição de sujeito.



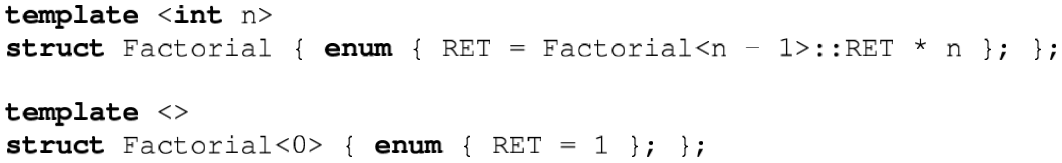
* **Aspect**-Oriented Programming
  + Lida com propriedades não funcionais de sistemas baseados em componentes.
    - Substitui fragmentos de código espalhados por vários componentes por aspectos reusáveis
  + Um dos maiores problemas da programação orientada a aspectos é a dificuldade de separar os aspectos dos componentes durante a fase de design, e subsequentemente implementando componentes independentes de aspectos.



* Generic Programming
  + Reusabilidade por meio de parametrização
    - Desacoplando algoritmos de estruturas de dados
  + Componentes genéricos
    - Extremamente ajustáveis (parâmetros)
  + C++ Standard Template Library (STL)



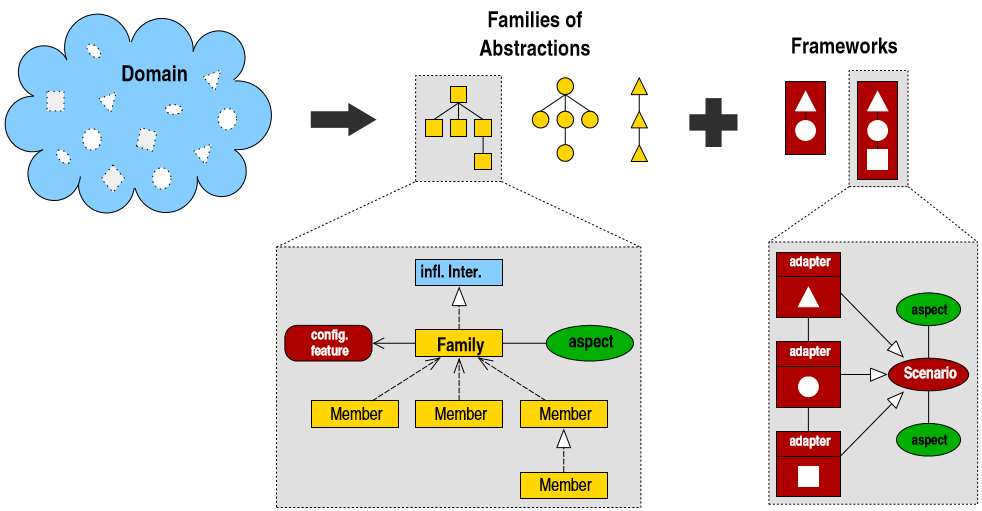
* Static Metaprogramming
  + Linguagens multinível
    - Partes do programa de entrada são avaliados em tempo de compilação
  + Transformação de componentes e composição



* Generative Programming
  + Engenharia de domínio
    - Familias
  + Conhecimento de configuração
    - Componentes em produtos
  + Geradores:
    - Programação orientada a aspecto
    - Programação orientada a sujeito
    - Metaprogramação estática
* Multiparadigm Design
  + Um único paradigma não pode cobrir particularidades de todos os domínios, eles devem ser combinados.

-Application-Oriented Operating Systems

* Um sistema operacional orientado a aplicação só é definido em relação a(s) aplicação(ões) correspondente(s) para o qual ele implementa o suporte de tempo de execução necessário que é entregue conforme solicitado.

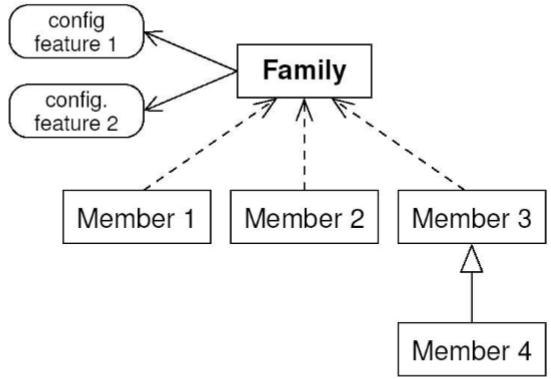


-Application-Oriented Domain Decomposition

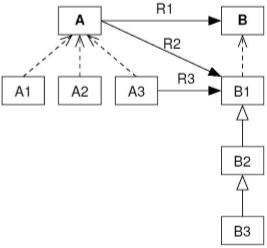
* Abstrações modelam entidades do domínio
* Analise de Commonality
  + Constrói famílias de abstrações
* Analise de Variability
  + Forma membros de famílias (subclasses ou não)
  + Separa aspectos de cenário
* Factorização
  + Características configuráveis
* Relações entre famílias
  + Propriedades do sistema
  + Arquiteturas reutilizáveis

-Scenario-Independent Abstractions

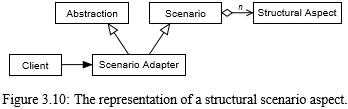
* Pode ser reutilizada em uma variedade de cenários
* Famílias
  + Hierarquia de classes
  + Classes cooperando
  + **Package comum**
    - Classe básica ou classes de utilidade
    - Características configuráveis



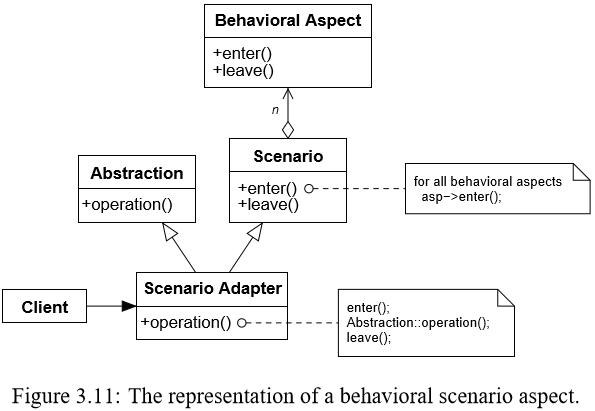
* Inter-Family Relationships
  + Formam as regras de composição do framework
  + Evite:
    - Regras restritivas
    - Regras frouxas
    - Relações por razão de reutilização
  + Famílias podem depender de outras famílias ou membros de outras famílias e membros de famílias podem depender de membros de outras famílias. A dependência pode ser de **existência** ou de **exclusão**.



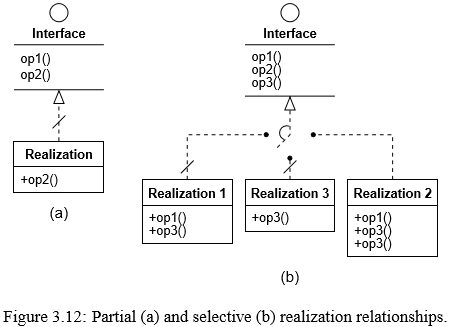
* Scenario Aspects
  + Propriedades que transcendem o escopo das abstrações
    - Dependências de cenário
    - Propriedades não funcionais
  + Podem ser organizados como famílias
  + Aplicação a abstrações
    - AOP Weaver
    - Adaptadores de cenário
  + Aspectos de cenário podem ser tanto **estruturais** como **comportamentais**:
    - O primeiro tipo modifica a estrutura das abstrações, adicionando alguma estrutura de dados especifica do cenário a elas. O identificador global único que é atribuído a uma abstração para suportar invocação remota em um cenário distribuído é um exemplo de aspecto estrutural.



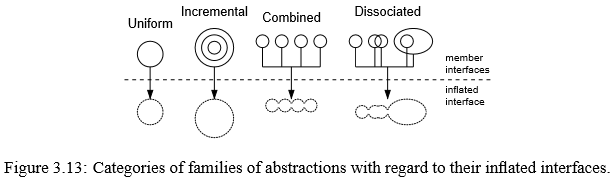
* + - O segundo tipo de aspecto de cenário modifica o comportamento das abstrações, reforçando semânticas especificas de cenário nelas. Fixando um *lock* as abstrações para que invocações concorrentes de suas operações sejam coordenadas é um exemplo de aspecto comportamental.



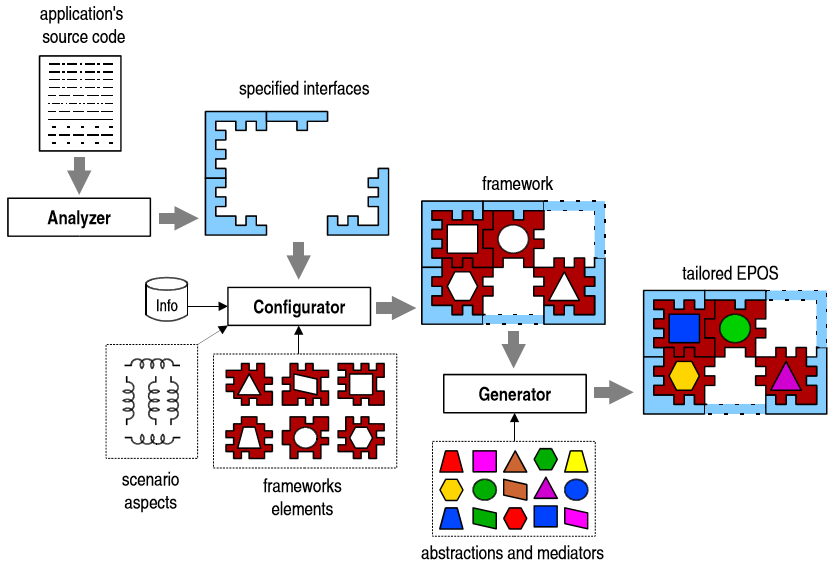
* **Configurable** **Features**
  + Características configuráveis diferem de aspectos em que:
    - Eles são específicos para uma única família de abstrações
    - Eles não são transparentes para as abstrações
      * Mas encapsulam implementações de programação genérica de algoritmos e estruturas de dados associadas a característica que pode ser reusada por abstrações quando a características for ativada
* **Inflated** **Interfaces**
  + Exportam famílias de abstrações para aplicações como se elas fossem uma única abstração
  + **Partial** and **Selective** Realization
    - Realização parcial implementa apenas um subconjunto especifico da interface inflada.
    - Realização seletiva significa que apenas uma das realizações pode ser vinculada a interface inflada por vez.



* + Inflated Interface Types
    - **Uniform**: uma família de abstrações na qual todos os membros compartilham a mesma interface;
    - **Incremental**: uma família de abstrações que tenha sido submetido a um projeto incremental, com cada membro sendo uma extensão do membro anterior;
    - **Combined**: uma família de abstrações na qual membros não apresentam nenhuma intersecção, mas que permitem membros serem automaticamente mesclados (via herança múltipla) para produzir novos membros com suas funcionalidades combinadas;
    - **Dissociated**: uma família de abstrações que não se enquadra em nenhuma das categorias precedentes.



-Tailoring EPOS



-EPOS Frameworkk Metaprogram

