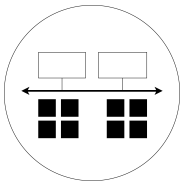
Multiprocessamento



30 de novembro de 2023

Sumário

Versão preliminar
Consulte o livro texto em caso de dúvidas.

1 Enquanto isso, no mercado doméstico...

Processadores multicore já haviam sido desenvolvidos no século passado para casos específicos e algumas aplicações pesadas; mesmo o Pentium Pro da Intel em 1995 já estava preparado para rodar em paralelo com outro CI idêntico conectado a ele, sem hardware adicional. Como vimos no estudo da Lei de Moore, o *Power wall* praticamente paralisou a aceleração do clock vista até então e obrigou os fabricantes a achar um contorno para manter o aumento contínuo de desempenho, imprescindível no mercado de microprocessadores.

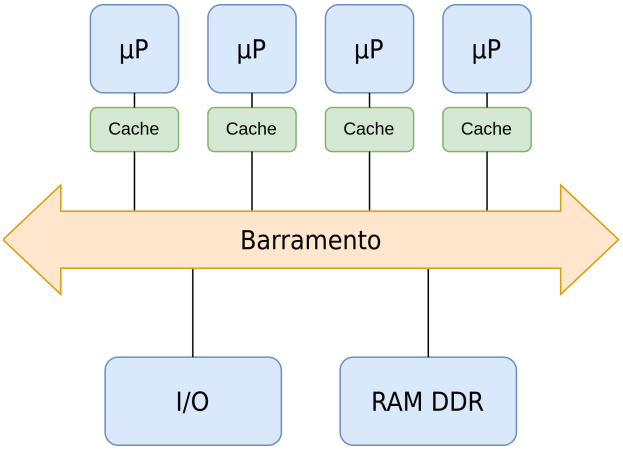


Figura 1: Quadcore em arranjo SMP.

A solução foi, claro, colocar mais de um processador no mesmo encapsulamento. Embora nem todos os casos de uso se beneficiem de "mais processadores no mesmo sistema," esta tendência se solidificou e multicores são utilizados hoje em dia até mesmo em casos menos óbvios, como certos equipamentos IoT.

2 Symmetric multiprocessing – SMP

Se quisermos transformar um sistema single core para um dual core, a primeira solução que vem à cabeça de qualquer pessoa é colocar um processador extra em paralelo (de alguma forma) ao original, de maneira que nenhum deles tenha prioridade sobre o outro. Obviamente teremos alguns conflitos (discutidos mais à frente), mas parece ser a saída mais simples.

Na figura 1 vemos esta organização em um quadcore típico. Os componentes principais (microprocessadores, periféricos e memória principal) são interconectados através de um barramento.

Perceba inicialmente que nenhum dos núcleos possui qualquer vantagem em relação aos demais. O acesso à memória RAM principal se dá através de um barramento, pelo qual as informações trafegam entre ela e as linhas de cache, e o acesso a periféricos se dá de forma similar. Como neste barramento há várias funcionalidades simultâneas e de naturezas diferentes, é necessário um circuito controlador, que irá decidir a ordem e momento das operações.

O multiprocessamento se iniciou na década de 1960, e multiprocessadores assimétricos eram mais comuns do que os simétricos: processadores diferentes da CPU principal eram utilizados para delegar tarefas secundárias, como trabalhos de impressão. Hoje em dia, usamos o temo SMP para indicar processadores com memória compartilhada que não possuem uma

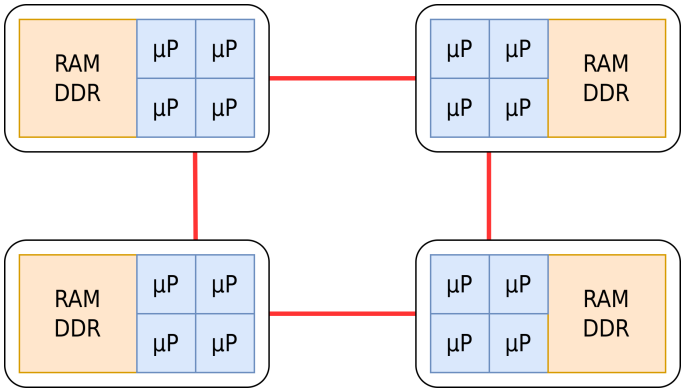


Figura 2: Dezesseis processadores em um arranjo NUMA.

hierarquia de acesso. [1]

É importante notar que podem haver conflitos no acesso a este barramento, já que temos vários agentes independentes funcionando em paralelo. Este conflito deve ser resolvido por algum critério de arbitragem pelo sistema.

No contexto de desempenho, uma disputa pelo barramento (*bus contention*) é quando estes conflitos se tornam excessivos, causando um impacto considerável no desempenho. É fácil imaginar que, se tivermos um SMP com 64 processadores, o barramento único vai se tornar um gargalo incontornável no sistema, já que todas as comunicações devem passar por ele.

3 Non-uniform memory access – NUMA

A solução para sistemas com muitos processadores é utilizar algo diferente de um barramento único; isso implica novas topologias de organização, sendo as mais usadas a topologia em anel e a rede totalmente conectada (*fully connected mesh* ou *fully connected network*), bastante comuns em processadores comerciais da Intel e AMD. Há alternativas mais exóticas, como conexões em cubo ou *crossbar* ou a hierarquia usada nas placas de vídeo.

Nosso exemplo de trabalho pode ser visto na figura 2. Neste arranjo específico, temos quatro processadores quadcore interligados em uma topologia de anel.

O ponto de maior interesse aqui (e que batiza estes sistemas) é que *o acesso à memória não é mais simétrico entre os processadores.* Como a RAM DDR está distribuída por cada nó quadcore, um processador que deseja acessar um dado local terá este acesso mais rápido do que se tentar acessar um dado em outro nó. Dito de outra forma, um dado x localizado em alguma RAM terá uma certa prioridade de acesso para os processadores

[1] O tempo AMP – *Asymmetric multiprocessing* não é usado contemporaneamente.

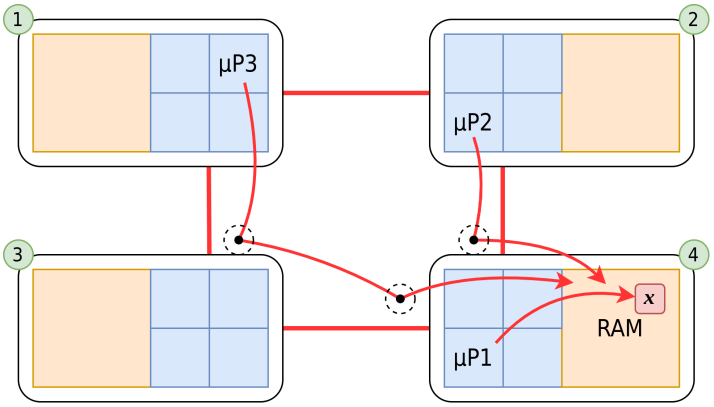


Figura 3: Tempo variável de acesso num arranjo NUMA.

locais e terá acesso mais lento para outros processadores do sistema, devido aos atrasos inerentes a este tipo de comunicação.

Isso cria uma hierarquia de acesso aos dados de acordo com qual processador vai acessá-lo, o que podemos expressar simplesmente por um valor de latência no acesso, que é dependente da localização na rede; ou seja, o acesso à memória não é uniforme entre os processadores. [2]

Usando a figura 3, é fácil perceber que o acesso à posição de memória x do nó 4 será praticamente imediato para o µP1, mas terá uma latência adicional significativa para o µP2, já que há um protocolo a seguir, roteamento a ser feito, esperas durante as etapas de transmissão e leituras, tudo isso inerente à troca de informações entre dois nós de uma rede. O µP3 terá esta latência adicional em dobro.

Pode-se perceber também que a topologia em anel utilizada força algumas restrições. No caso da figura 3, se o µP3 estiver acessando x pelo caminho indicado, nenhum dos processadores do nó 3 poderá acessar valores dos outros nós, já que a capacidade de comunicação do nó 3 está sendo utilizada para rotear o acesso externo.

Uma forma simples de evitar este problema é usar uma rede completamente conectada, com todos os nós conectados individualmente a todos os outros. Note, no entanto, que se tivermos um número grande de nós, o número de conexões cresce quadraticamente...

Portanto, se quisermos ter um *cluster* ou servidor com centenas ou milhares de processadores, o mais viável é ter módulos NUMA interconectados em um sistema com acesso por LAN ou WAN, talvez simplesmente usando a Ethernet e endereços de IP. Nestes casos, a distribuição de trabalho e balanceamento do workload pelo sistema operacional se torna crucial para

[2] Tecnicamente, processadores UMA com acesso uniforme são os que não têm latência variável para cada processador, como nossos SMP, mas este termo não é usado contemporaneamente.

evitar altas latências e ociosidades no sistema, que ocasionam perda de desempenho e de eficiência energética.

4 Coerência de cache

A partir do momento em que temos uma hierarquia de memória, precisamos examinar a consistência dos dados dentro da hierarquia, pois cada dado poderá ter várias cópias. Dentro de um sistema single core o problema é bem mais simples, já que apenas um programa está rodando em qualquer momento.

Se temos mais de um processador potencialmente acessando a mesma localidade de memória, e se esta localidade pode ter várias cópias espalhadas pelo sistema, entramos nos domínios de um sistema paralelo com dados distribuídos e precisamos tratá-lo de acordo.

Na prática, o que ocorre é que podemos ter uma posição de memória compartilhada por dois programas rodando em diferentes processadores, e um pode tentar alterar esta posição independentemente do outro, que por sua vez ficaria com uma cópia desatualizada da informação.

4.1 Protocolo MESI

Há vários métodos usados contemporaneamente para garantir coerência de cache. Vamos mostrar aqui apenas um deles para ilustrar, por ser de fácil compreensão.

Podemos simplesmente marcar cada uma das linhas de cache com um estado, indicando ao processador o que está acontecendo com os dados desta linha. Vamos utilizar quatro possibilidades, que correspondem às letras do MESI:

Exclusive quando a linha da memória foi copiada para uma única cache;

Shared quando a linha de memória foi copiada para duas ou mais caches, acessadas por processadores diferentes;

Modified quando o processador associado à cache alterou um dado na linha; ou

Invalid quando *outro* processador alterou uma cópia desta linha de cache.

O mecanismo então é o seguinte: quando uma linha da memória principal é acessada pela primeira vez (digamos pelo processador µP1), ela é copiada para a cache do µP1 e é marcada na cache como sendo exclusiva. Neste caso, o processador pode ler à vontade os dados. Se escrever algo, a linha fica marcada como modificada, mas nenhuma ação extra é necessária, já que a linha não é compartilhada.

Caso um outro processador µP2 tente acessar uma linha cuja cópia no µP1 esteja marcada como E, outra cópia da linha será feita para sua cache e *ambas as linhas de µP1 e µP2* serão marcadas como compartilhadas. Acessos para leitura continuam válidos, já que todas as cópias possuem os mesmos valores.

O foco do protocolo ocorre quando há uma escrita em alguma linha compartilhada, marcada com S. Neste caso, a linha escrita (digamos a do µP1) vai para o estado M, indicando que ela foi modificada em relação ao conteúdo original da memória principal. Ao mesmo tempo, todas as demais cópias desta linha em cache (neste exemplo, apenas na cache de µP2) deverão ser marcadas como inválidas, indicando que estão desatualizadas e não podem ser usadas.

Neste ponto, temos várias possibilidades. Escritas adicionais em uma linha M não provocam demais mudanças. Quaisquer linhas I, se descartadas da cache, são simplesmente ignoradas, não necessitando de ação adicional. Se uma linha M for descartada da cache, no entanto, ela deverá ser escrita na memória principal, o que é apenas nossa conhecida disciplina de escrita write-back.

No entanto, se houver leitura em uma linha marcada como inválida, digamos pelo µP2, faremos um processo de atualização, que consiste em realizar o write-back da linha M correspondente ao endereço acessado e proceder com a cópia desta linha para a cache do µP2. Ambas as linhas ficam marcadas com S, então. No caso de uma escrita posterior pelo µP2, a linha da cache de µP2 fica como M e as demais como I, em princípio.

4.2 Cache snooping

Podemos implementar o protocolo MESI utilizando mensagens sendo emitidas pelas controladoras de cache e pelo controlador do barramento, mas isto gera bastante processamento.

Uma alternativa é o método chamado de cache snooping ("bisbilhotagem da cache"), cujo diagrama é ilustrado na figura 4.

A ideia é que o bloco de cache snoop monitora as informações que trafegam no barramento; como ele tem acesso livre à cache local, ele pode dizer se as operações que estão ocorrendo no barramento são relevantes para o processador. De forma análoga, operações na cache que podem ser relevantes a outros processadores são "publicadas" no barramento.

Por exemplo, suponhamos que a cache do µP1 tem a linha com o endereço 0xB1FE F0F0 marcada no estado exclusivo:

1. se o processador µP2 realiza a leitura deste endereço, o snoop do µP1 vai "enxergar" esta operação no barramento e a linha será marcada como compartilhada, em ambos os processadores;

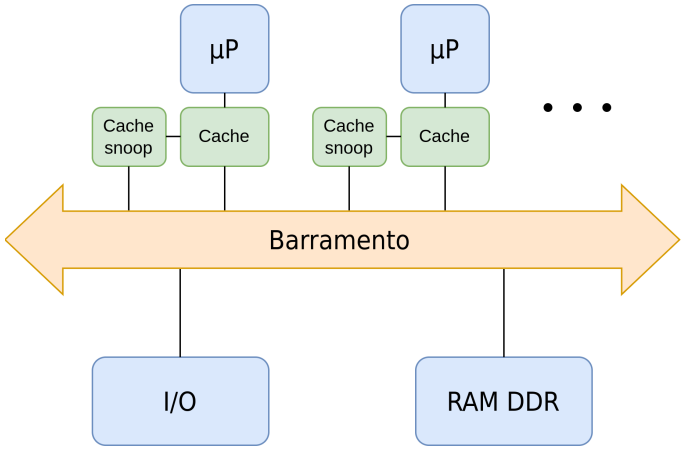


Figura 4: Diagrama do circuito de snooping.

2. se depois disso o µP2 escrever neste endereço, o snoop dele vai publicar a escrita na linha do 0xB1FE F0F0 no barramento e a sua linha ficará marcada como modificada;

3. o snoop do µP1 vai observar esta escrita no barramento, vai verificar que, sim, há uma cópia desta linha na sua cache e, finalmente, vai marcar esta linha da cache do µP1 como inválida.

Obviamente isto só vai ser eficiente em sistemas SMP ou similares, com todo o circuito no mesmo encapsulamento. Se tentássemos cache snooping num sistema NUMA com vários CI's fisicamente separados, as latências iriam impactar demais o desempenho, o que inviabiliza esta técnica neste caso.

5 Taxonomia de Flynn ainda n vi

A taxonomia de Flynn é um método clássico de classificação de processadores de acordo com o tipo de paralelismo, que só é útil pelo uso contemporâneo da sigla SIMD. Consiste em quatro categorias:

SISD Single Instruction Single Data;

SIMD Single Instruction Multiple Data;

MISD Multiple Instruction Single Data;

MIMD Multiple Instruction Multiple Data.

O "single"/"multiple" diz respeito a quantos "fluxos" de instruções ou dados o sistema é capaz de processar. Vamos examinar os casos mais de perto.

O SISD significa que temos um fluxo de instruções únicas agindo cada uma sobre um dado apenas, ou seja, é uma sequência de operações escalares, como num típico processador single core contemporâneo.

Já o SIMD implica que cada instrução do fluxo *age sobre um conjunto de dados*. Isso quer indicar para nós que este é um processador que realiza instruções vetoriais, como visto no capítulo de ILP. Normalmente, estas são descritas como "instruções SIMD" nas ISA's, e são implementadas como subconjuntos de instruções (como as AVX da Intel ou as NEON da ARM, dentre outras).

Nos processadores de placas gráficas (GPU's), além das instruções vetoriais, se usa um conceito similar, chamado de SPMD ou *Single Program Multiple Data,* no qual uma única instrução (vetorial ou não) é aplicada a uma série de dados distintos. Veja mais na seção 6.3, em multithreading fine grained.

O MISD é o esquisitão do grupo. Nele, um único dado é utilizado por mais de um fluxo de instruções – na prática, temos mais de um processador executando o mesmo programa (ou seja, mais de um fluxo em paralelo). Este modelo é usado para descrever sistemas com tolerância a falhas, que é um tópico bastante abrangente em si. Num exemplo típico, temos três processadores executando uma mesma instrução em lockstep (talvez atrasados um clock ou meio clock entre eles) e verificando se o resultado dos três é igual, por uma votação; se não houver unanimidade, pode-se tentar reexecutar a instrução ou então desconectar o processador discordante, considerando que ele teve algum problema.

Finalmente, o MIMD é como o nosso multicore de cada dia: cada um dos múltiplos processadores executa seu próprio fluxo de instruções com seu próprio fluxo de dados.

6 Multithreading

Multithreading é a capacidade de executar mais de uma thread de instruções, ou seja, dois ou mais fluxos de instruções (dois ou mais programas, simplificadamente) rodando em um mesmo microprocessador.

As vantagens disto ficarão claras mais à frente, mas digamos que, com uma única thread, o processador acaba desperdiçando alguns recursos e tempo pela ociosidade causada por dependências de dados e outros hazards. Com duas threads, esta ociosidade diminui, o que deve aumentar a eficiência energética. Ademais, com a duplicação de alguns blocos estratégicos da CPU, um único processador será visto como sendo *dois* processadores pelo sistema operacional.

Esta é a explicação para a diferença entre *núcleos físicos* (os processadores de fato em hardware) e *núcleos lógicos* (os processadores como percebidos pelo sistema operacional).

6.1 Terminologia e hardware

É importante notar que no estudo de Sistemas Operacionais uma thread é um "subprocesso light," fácil de criar e de trocar informações, e com algum grau de paralelismo. É uma visão de software.

Em Arquitetura de Computadores, olhamos para a mesma definição do ponto de vista do hardware: como projetar um microprocessador capaz de gerenciar a execução de dois programas ao mesmo tempo?

A resposta é mais simples do que parece: para cada thread adicional precisaremos *duplicar todo o estado de um processador,* o que significa adicionar um novo banco de registradores inteiro, um PC, um SP e todos os outros registradores e flip-flops necessários para um programa ser executado (note que os registradores interestágios e os buffers do pipeline não precisam ser duplicados).

Desta forma, se implementarmos um RISC-V com multithreading de hardware para duas threads chamadas de A e B, teríamos registradores x0 a x31 separados para cada thread, digamos registradores Ax0 a Ax31 e Bx0 a Bx31, bem como APC e BPC e assim por diante. A circuitaria combinacional, como ULA's e outras unidades de execução, não precisam ser duplicadas, mas toda a lógica de controle fica um pouco mais complicada.

6.2 Desvantagens

As desvantagens de se usar multithreading em hardware podem ser resumidas no fato de que as threads sendo executadas vão competir pelos recursos que são compartilhados sem serem duplicados. Em especial, estamos falando das unidades de execução do superescalar (como as ULA's e as unidades de leitura e escrita da memória) e da memória cache. As primeiras são limitadas pelo projeto do superescalar; as caches têm o tamanho limitado pela necessidade de baixíssima latência.

Dois programas que rodam no mesmo núcleo irão tomar os recursos disponíveis, beneficiando um programa em detrimento do outro. Por exemplo, se um programa tem uso intensivo de memória, ele provavelmente irá dominar os acessos aos 64 kB de cache L1, fazendo que, com maior frequência, o outro programa incorra em acessos mais custosos na L2, L3 ou RAM principal, o que vai limitar o seu desempenho.

Do ponto de vista de um programa sendo executado, o desempenho será menor do que em um núcleo normal single-threaded. Isto é lógico: se a CPU for exclusiva para um programa, tudo o que for feito é para a execução dele; se ela for dividida entre dois programas, parte das suas capacidades serão direcionadas para o segundo programa. Ambos vão perceber um desempenho reduzido para si em relação a um sistema similar sem multithreading.

Por estas razões, nos PC's com multithreading, este pode ser desabilitado no menu de boot, pois em certos casos de uso o usuário talvez não

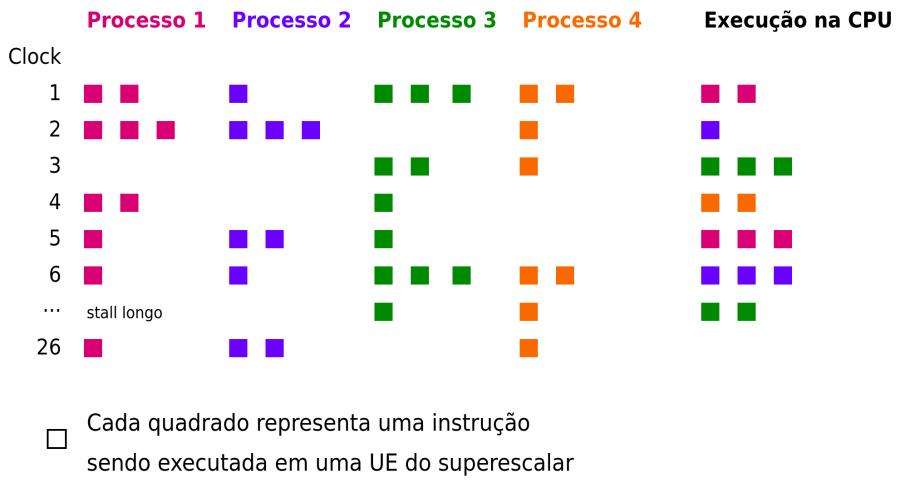


Figura 5: Exemplo de execução de multithreading fine grained.

queira esta perda. Por outro lado, ter mais processadores disponíveis pode fazer diferença quando temos muitos processos a serem executados simultaneamente no sistema.

6.3 Modelos

A decisão fundamental em multithreading é a forma em que iremos escalonar os programas sendo executados no mesmo processador físico.

Multithreading temporal, fine grained

A forma mais óbvia de implementar multithreading é intercalar os processos. Assim, os circuitos da CPU "pertencem" a um dos processos durante certo tempo e no restante do tempo são usados pelo outro ou outros. Resta decidir o algoritmo de intercalação, ou seja, a granularidade destas unidades de tempo.

O método temporal mais simples consiste em dar um ciclo de clock para a execução de cada um dos processos presentes, o chamado *fine grained temporal multithreading* ou MT temporal de granularidade fina, em tradução livre. Podemos ver graficamente um exemplo na figura 5, no qual um processador com MT fine grained intercala 4 processos diferentes, um de cada cor, dando um clock de CPU para cada um deles.

Nesta figura, cada quadrado é uma instrução sendo executada em uma das Unidades de Execução do microprocessador superescalar (UE's, como ULA's de inteiros, ULA's de ponto flutuante, unidades de Load e Store etc). Cada linha é um clock, então os quadrados na mesma linha são instruções executadas simultaneamente.

Além de termos 4 processadores lógicos implementados com o circuito de um único processador físico, uma outra vantagem fica clara aos olhos: pequenos stalls (linhas sem instruções) devidos a, por exemplo, dependências de dados ou atrasos na cache L1, serão completamente eliminados, pois outros processos irão utilizar esse intervalo de clocks que é necessário para que a informação fique disponível. Com menos clocks desperdiçados, provavelmente teremos maior eficiência energética. Os stalls longos terão seu impacto minimizado; por exemplo, um stall originalmente de 20 clocks no processo 1 iria gerar perda de apenas 5 clocks, com os demais 15 clocks sendo usados pelos processos 2, 3 e 4.

Por outro lado, é bastante claro que um programa terá para si apenas 25% do tempo total de CPU; se não tivéssemos multithreading neste processador, um processo teria 100% da CPU para si (embora executássemos apenas um processo por vez ao invés de 4). E ainda vale a observação de que os 4 processos irão competir pela memória, em especial compartilhando os exíguos 64 kB de cache L1, causando atrasos pela falta de capacidade da L1.

Multithreading temporal, coarse grained

Com o *coarse grained temporal multithreading,* iremos chavear o processo sendo executado apenas quando houver um grande stall, normalmente um acesso à memória principal ou uma operação de I/O, ambos com alta latência. Esta estratégia é ilustrada na figura 6. Se no modelo anterior temos granularidade mínima (1 clock), neste aqui a granularidade é bem mais grosseira, ou seja, os tempos alocados aos processos são bem maiores.

É fácil observar que, em contraste ao fine grained, os pequenos stalls não são evitados de forma nenhuma; em compensação, os grandes stalls são todos evitados através do chaveamento para o processo seguinte. À exceção disto, a única especificidade deste modelo é que, em situações atípicas em que os grandes stalls sejam raros (tal como um algoritmo numérico pesado, com muita aritmética em pouca memória, que cabe nas caches menores e quase sem I/O), um dos processos pode ficar com pouco tempo de CPU ou com alta latência para execução das instruções, já que a CPU pode ficar monopolizada pelo outro processo.

Multithreading simultâneo

No multithreading simultâneo vamos executar instruções de programas diferentes no mesmo clock. A ideia é distribuir, para as unidades de execução, instruções que estão prontas para serem executadas, ou seja, instruções com os operandos já disponíveis. Estas instruções, que estão sem dependências de dados que as segurem, são despachadas para execução para unidades disponíveis, como ULA's. A questão é que o processador não vai diferen-

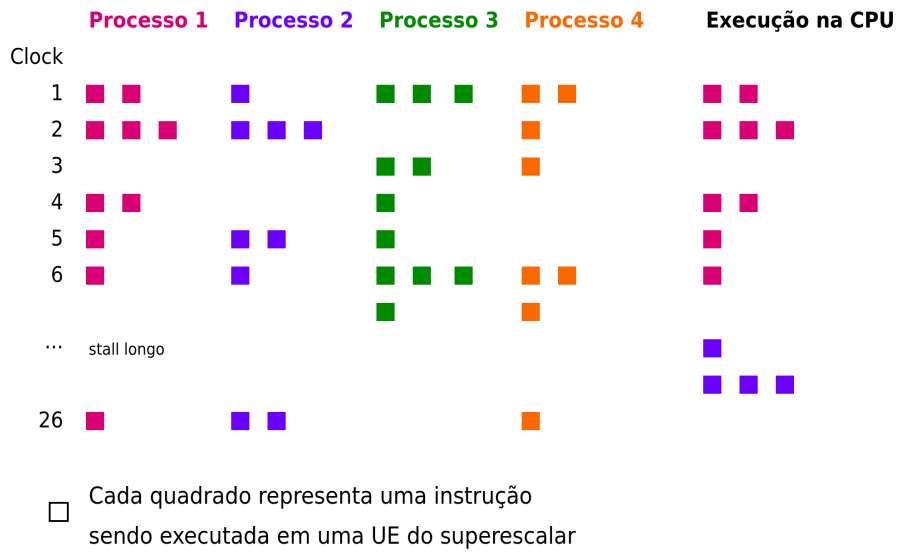


Figura 6: Exemplo de execução de multithreading coarse grained.

ciar qual dos programas é a fonte das instruções – todas as threads podem fornecer fluxos de instruções para ele, indistintamente.

O resultado é ilustrado em um exemplo na figura 7.

Neste modelo, tanto os stalls grandes quanto os pequenos são escondidos e há uma melhor ocupação das unidades de execução – se houver uma unidade ociosa, ela tem outras fontes de instruções para potencialmente preenchê-la. Embora a lógica de despacho dinâmico de instruções não seja muito diferente do superescalar fora de ordem, alguns detalhes de implementação fazem este circuito ficar mais complexo.

A Intel chama de Hyperthreading a sua implementação de multithreading simultâneo, seguindo seu hábito de rebatizar técnicas já existentes.

6.4 Comparação simples

Na prática, para o mercado de consumo doméstico, o que o multithreading faz é criar pontos intermediários no espectro dos equipamentos. Tome por exemplo três laptops com os processadores descritos:

1. PC com 4 núcleos físicos, 4 núcleos lógicos (quadcore sem multithreading);
2. PC com 4 núcleos físicos, 8 núcleos lógicos (quadcore com multithreading);
3. PC com 8 núcleos físicos, 8 núcleos lógicos (octacore sem multithreading).

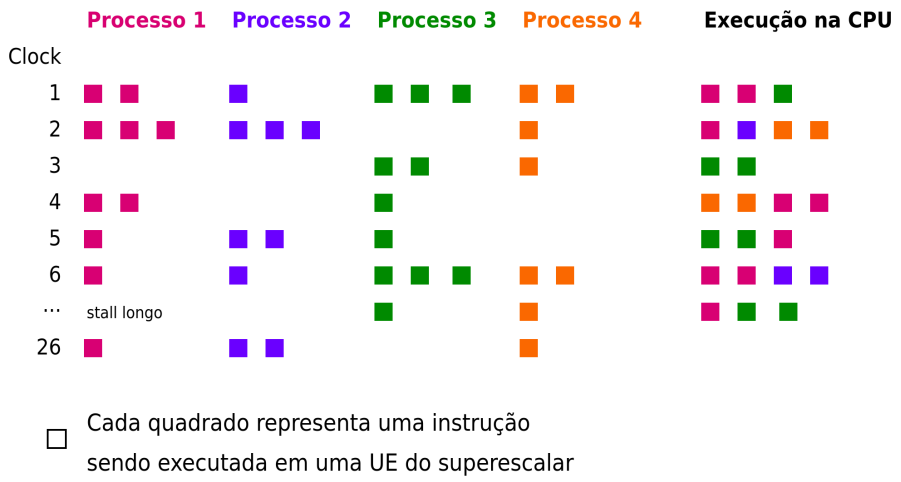


Figura 7: Exemplo de execução de multithreading simultâneo.

Vamos analisar algumas suposições razoáveis sobre o sistema a seguir. Os particulares podem alterar nossas conclusões (variações no clock ou cache, por exemplo), então supomos estas demais variáveis fixas.

O laptop 1 é obviamente o que tem menor desempenho, já que é visto pelo sistema com tendo apenas 4 processadores. Em compensação é o que gasta menos energia em absoluto, já que tem menos circuitaria na implementação.

No PC com multithreading, precisamos de mais circuitos do que o anterior, portanto temos mais gasto energético; em compensação temos 8 processadores disponíveis para executar programas em paralelo, portanto podemos assumir melhor desempenho.

Ainda no laptop 2, podemos supor que a eficiência energética, na média, seja melhor, já que os circuitos têm menos ociosidade; e que o desempenho máximo de pico para uma tarefa específica seja menor do que no PC 1, pois ela está provavelmente dividindo recursos com outra tarefa que executa no mesmo núcleo físico. Note que estas conclusões dependem da carga de trabalho atribuída aos processadores – programas diferentes executando simultaneamente podem ter resultados diferentes.

Finalmente o PC 3 é claramente o que tem melhor desempenho, pois seus 8 núcleos físicos não compartilham nenhum recurso entre si. A contrapartida é que ele também é o que gasta mais energia, provavelmente possuindo o dobro de transistores do laptop 1.

Resumindo: o 1 é o que economiza energia mas tem o pior desempenho; o 3 é o de melhor desempenho mas o que gasta mais energia; e o 2 é um ponto intermediário entre eles.