

Tembel Penguenlere Çekirdek Programlama

Faik Uygur
faik@pardus.org.tr

Mayıs 13, 2006

Çekirdek Programlama

- Ignore everyone who tells you kernel hacking is hard, special or different. It is however not magic, nor written in a secret language that only deep initiates with beards can read.
-
-

-- Alan Cox

Çekirdek Geliştiricileri

- Con Kolivas

- Anestezi uzmanı bir doktor.
- 1999 yılında 2.4.18 performans yamaları ile uğraşüyor.
- Benchmark uygulamalarını deneyerek sonuçları çekirdek listesine gönderiyor.
- contest ve interbench test araçlarını yazıyor.
- scheduler üzerinde çalışıyor.

Çekirdek Geliştiricileri

- Rusty Russell

- 1997 Usenix seminerinde Dave Miller'ın Linux'un Sparc'a portu konuşmasını dinliyor.
- Paket filtreleme kodunda eksik gördüğü kısımları yazıyor.
- ipfwadm -> ipchains – maintainer oluyor.
- ip_conntrack modülü ile uğraşırken module kodu ve userspace kısmı ile uğraşmaya başlıyor.
- module-init-tools maintainer'ı

Çekirdek Geliştiricileri

- Robert Love
 - Montavista'nın geliştirdiği preemptive patch'inin maintainer'ı oldu.
 - Linux 2.6 kitabı
- Marcello Tosatti
 - 18 yaşında 2.4 kernel'inin maintainer'ı oldu.
- Dave Jones
 - 2.4 -> 2.5 sync çalışması
 - Fedora çekirdek sorumlusu

Ortak Noktalar

- ekirdek listesi
- ekirdek ve eřitli yama testleri
- ekirdek hataları
- Benchmark alıřmaları
- Diđer geliřtiricilerin iřlerini kolaylařtıracak aralar.

Kod Okuma

- Sizden daha iyi programcıların kodlarını görebilirsiniz.
- Eksiklerinizin farkına varıp kendinizi geliştirebilirsiniz.
- Yazdığınız kodun kalitesi artmaya başlar.
- Karşılaştığınız problemlerin %90'ı başka bir proje'de çözülmüştür.
- Büyük projeler üzerinde çalışırken başka bir yere bakmaya ihtiyaç bile duymayabilirsiniz.

Kod Okuma

- Aşağıdan-Yukarı

- Kodun ne yaptığını anlamak daha zordur.
- Nereden başlayacağını bulmak büyük programlar için daha zordur.
- Detaylı bilgi sahibi olmayı sağlar.

- Yukarıdan-Aşağı

- Büyük resim hakkında bilgi sahibi olmak işinizi kolaylaştırır.
- Proje yapısı hakkında genel bir fikir sahibi olunur.
- Proje'de ilgili veriler tespit edilebilir.

Bilgisayar Mimarisi

- İşlemci Mimarisi
- Hafıza Yönetim Birimi (MMU)
- Cache Yapıları
- IO Mimarisi

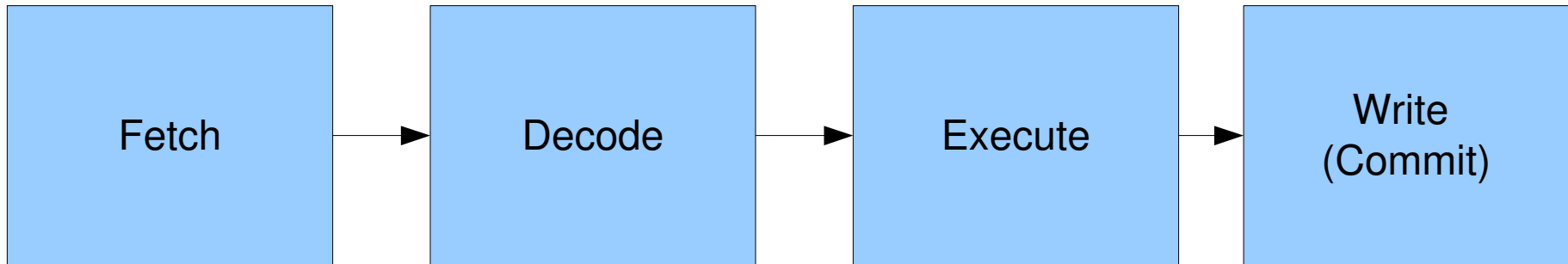
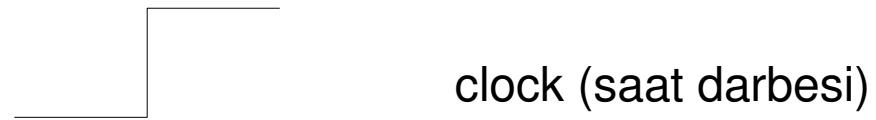
İşlemci Mimarisi

İşlemci Mimarisi

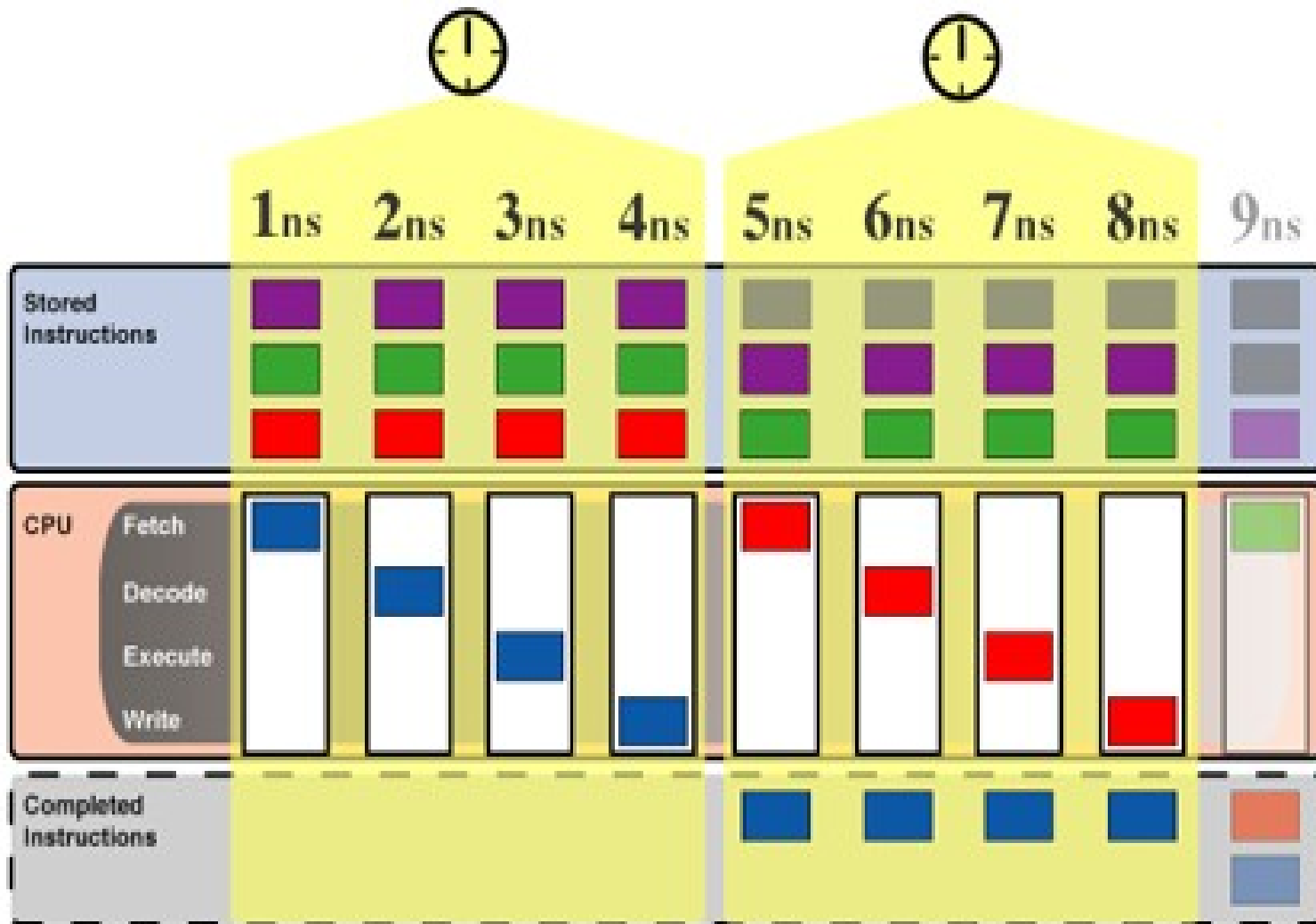
- İşlemci İç Yapısı
- Pipelining
- Superscalar Mimari
- Out-of-Order Execution
- Branch Prediction

İşlemci Yapısı

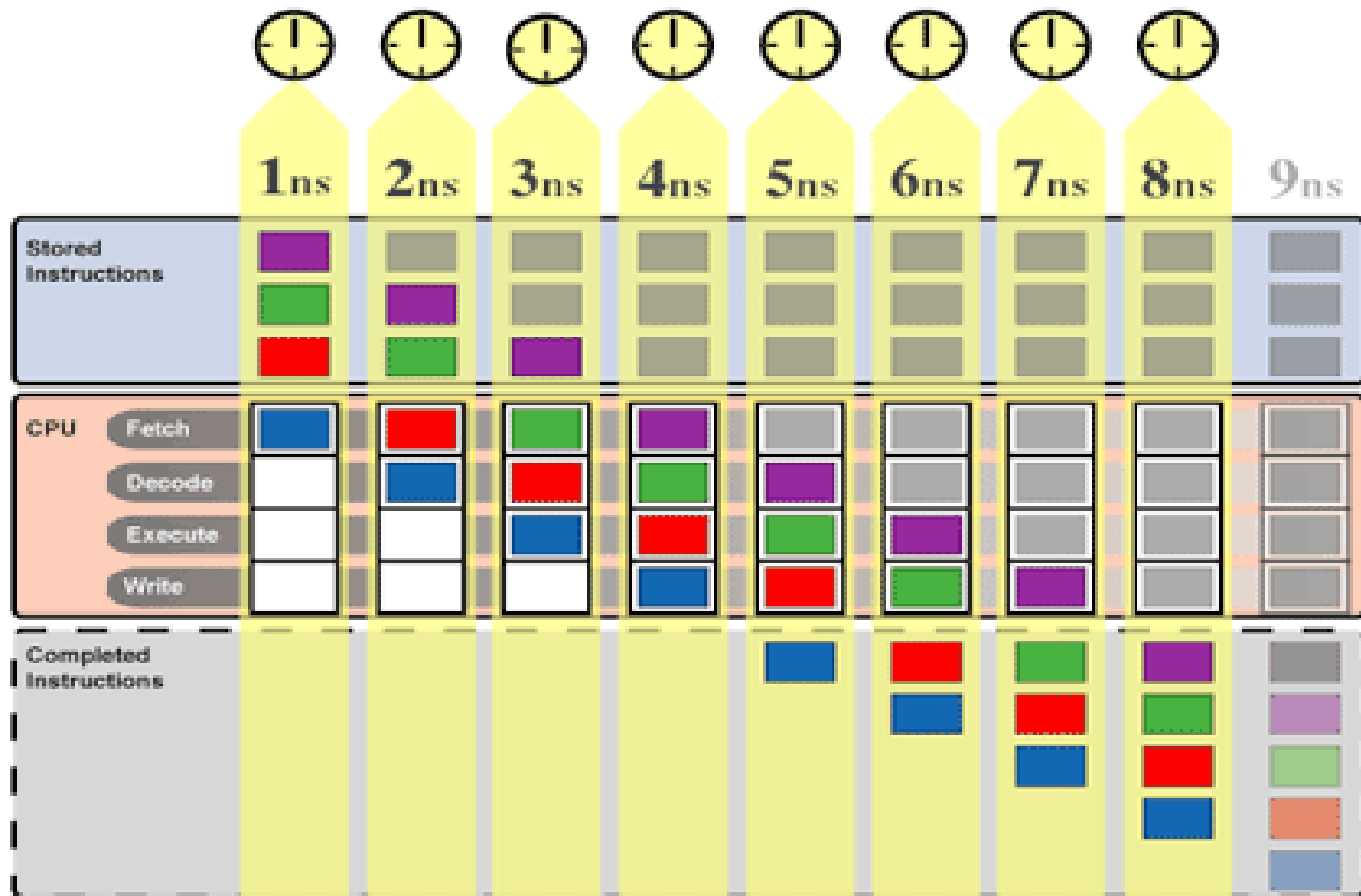
- Instruction Fetch (Al)
- Decode (Çöz)
- Execute (İşle)
- Write (Yaz)



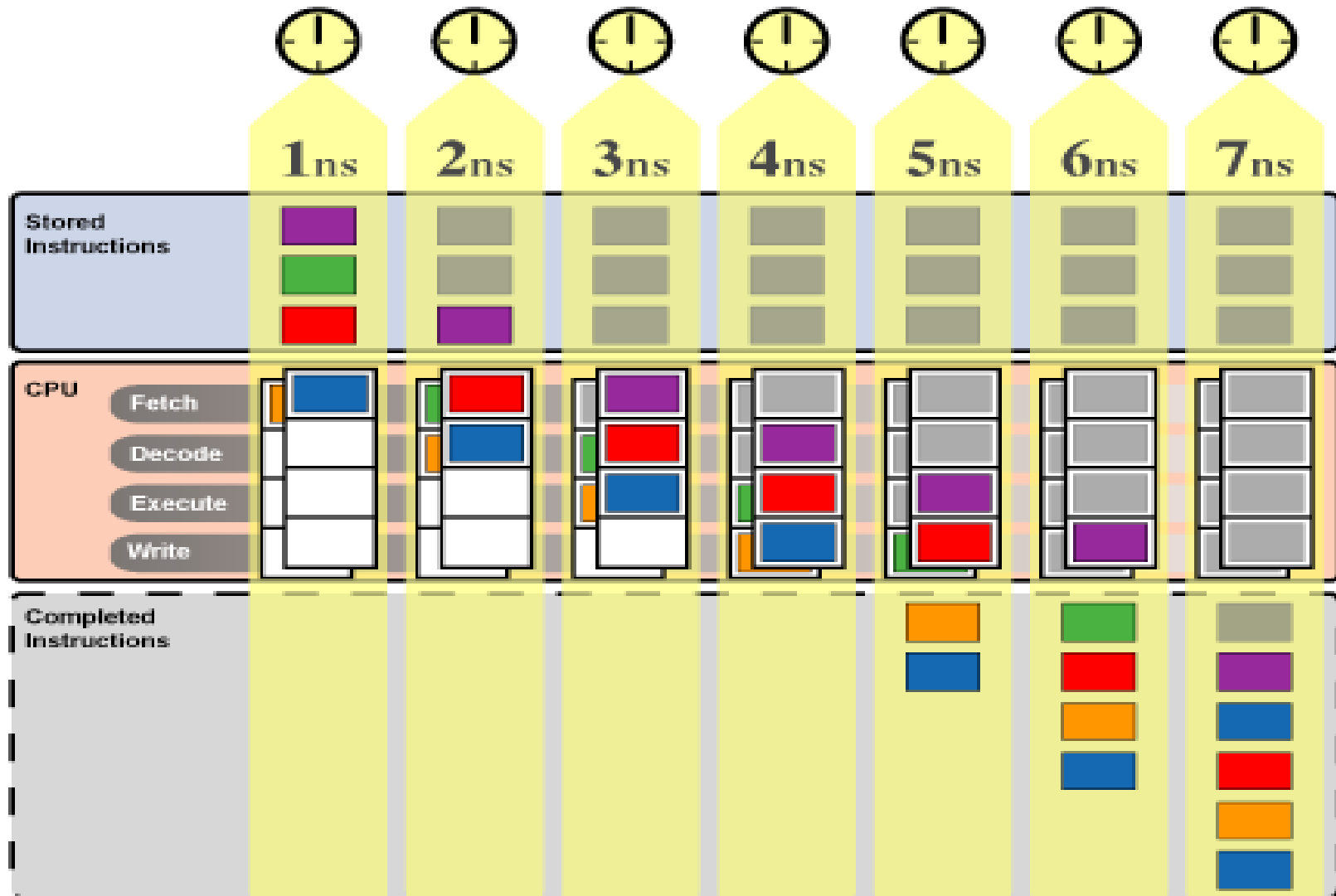
Pipelining



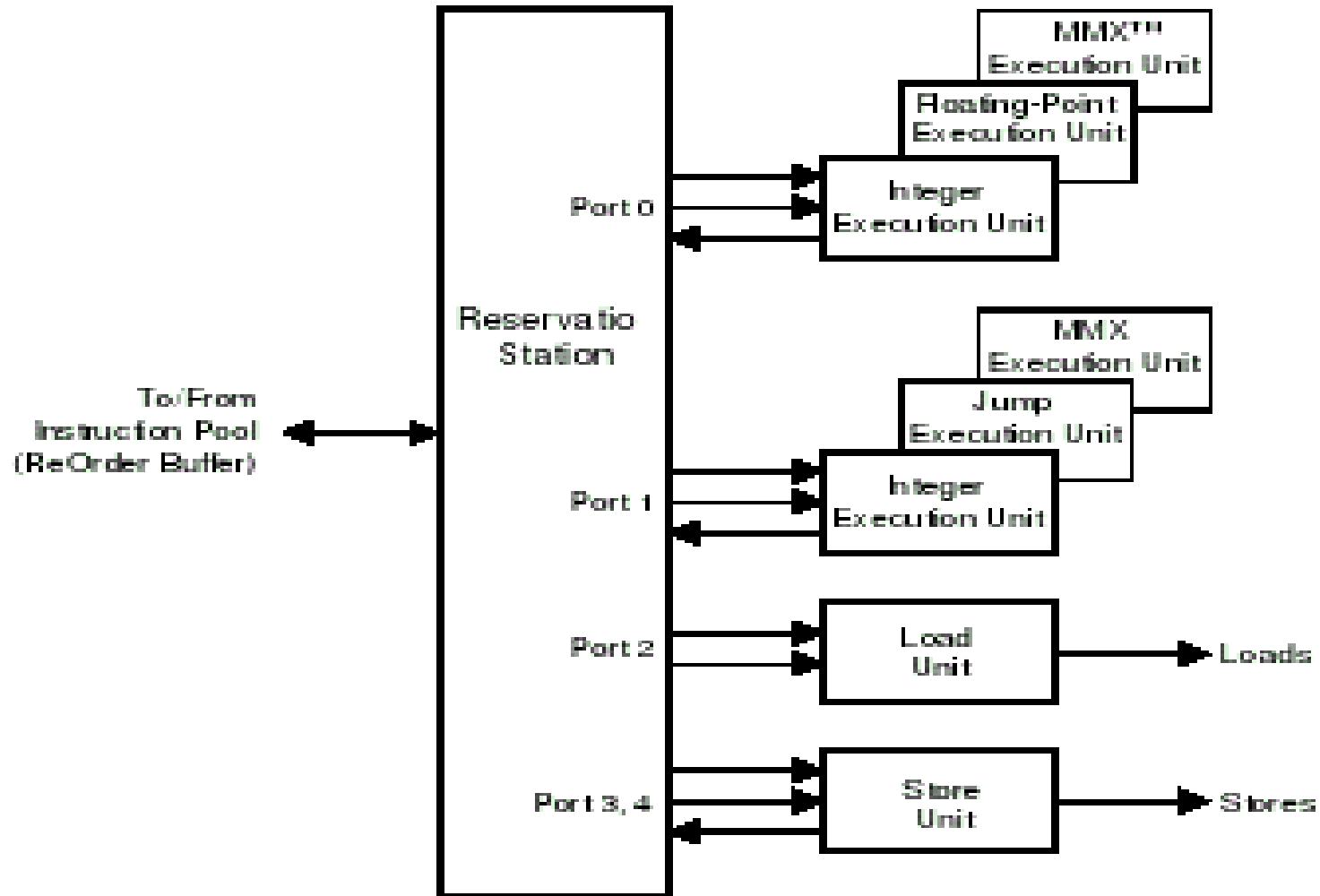
Pipelining



Superscalar Mimari



İşlemci Yapısı



Out-of-Order Execution

- Başka bir instruction sonucu bekleniyor olabilir.
- Hafızadan veri bekleniyor olabilir.
- Execution unit o an meşgul olabilir.

Out-of-Order Execution

- Reservation station'a her cycle'da 3 instruction gelebilir.
- Reorder buffer her cycle'da 5 instruction sonlandırabilir.

Branch Prediction

- P4 işlemcilerde hatalı tahmin instruction L1 cache'de ise 19 clock cycle
- Instruction hafızada ise 30 clock cycle zaman kaybına sebep oluyor.
- Pipeline uzadıkça ceza artar.

Branch Prediction

- Static Execution
 - Geri gösteren branchlar (döngüler)
 - İleriyi gösteren branchlar (durumlar)
- Speculative Execution
 - Branch History Table (BHT)
 - Branch Target Buffer (BTB)

Gcc Branch optimizasyonu

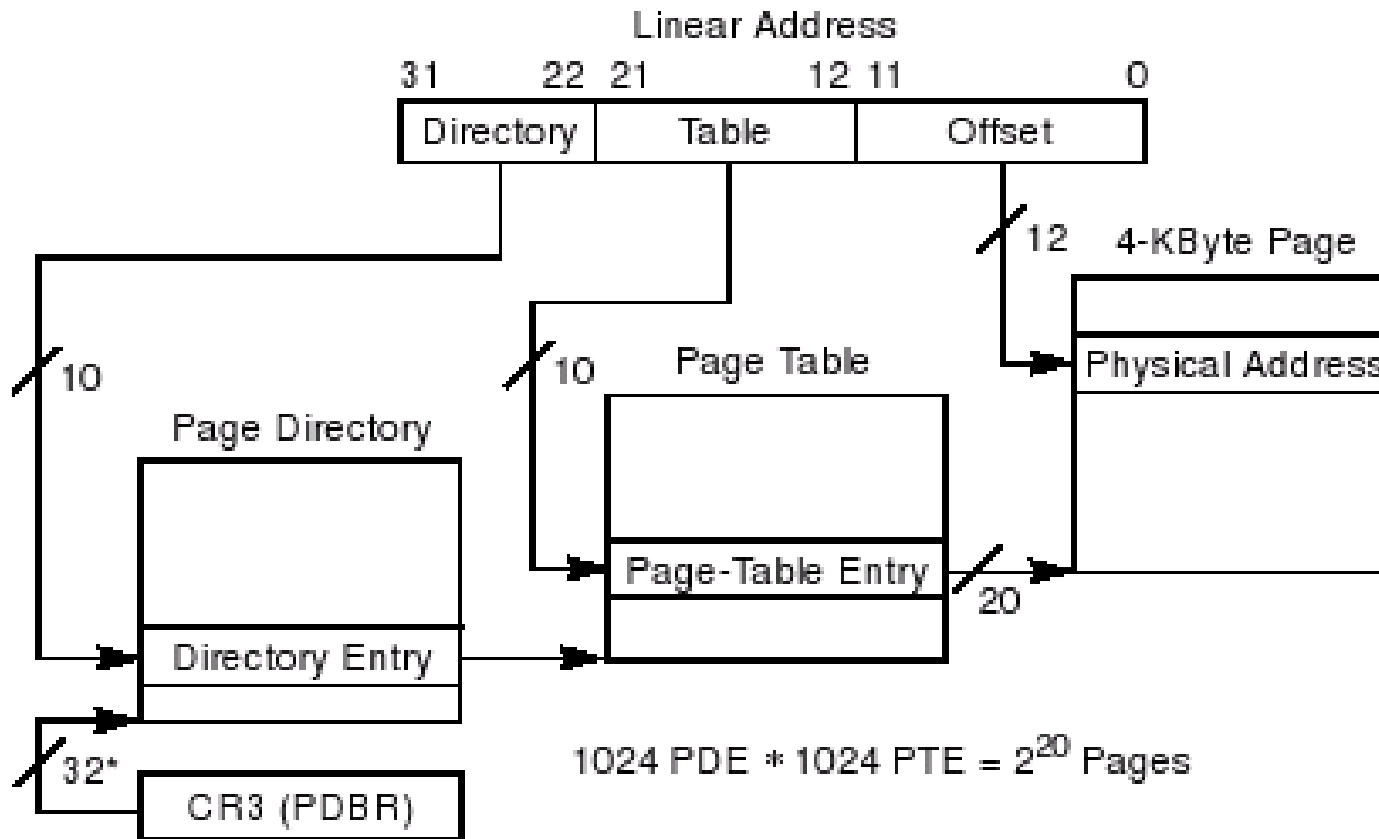
- `linux/compiler.h`
 - `likely()`
 - `unlikely()`

MMU (Memory Management Unit)

MMU (Memory Management Unit)

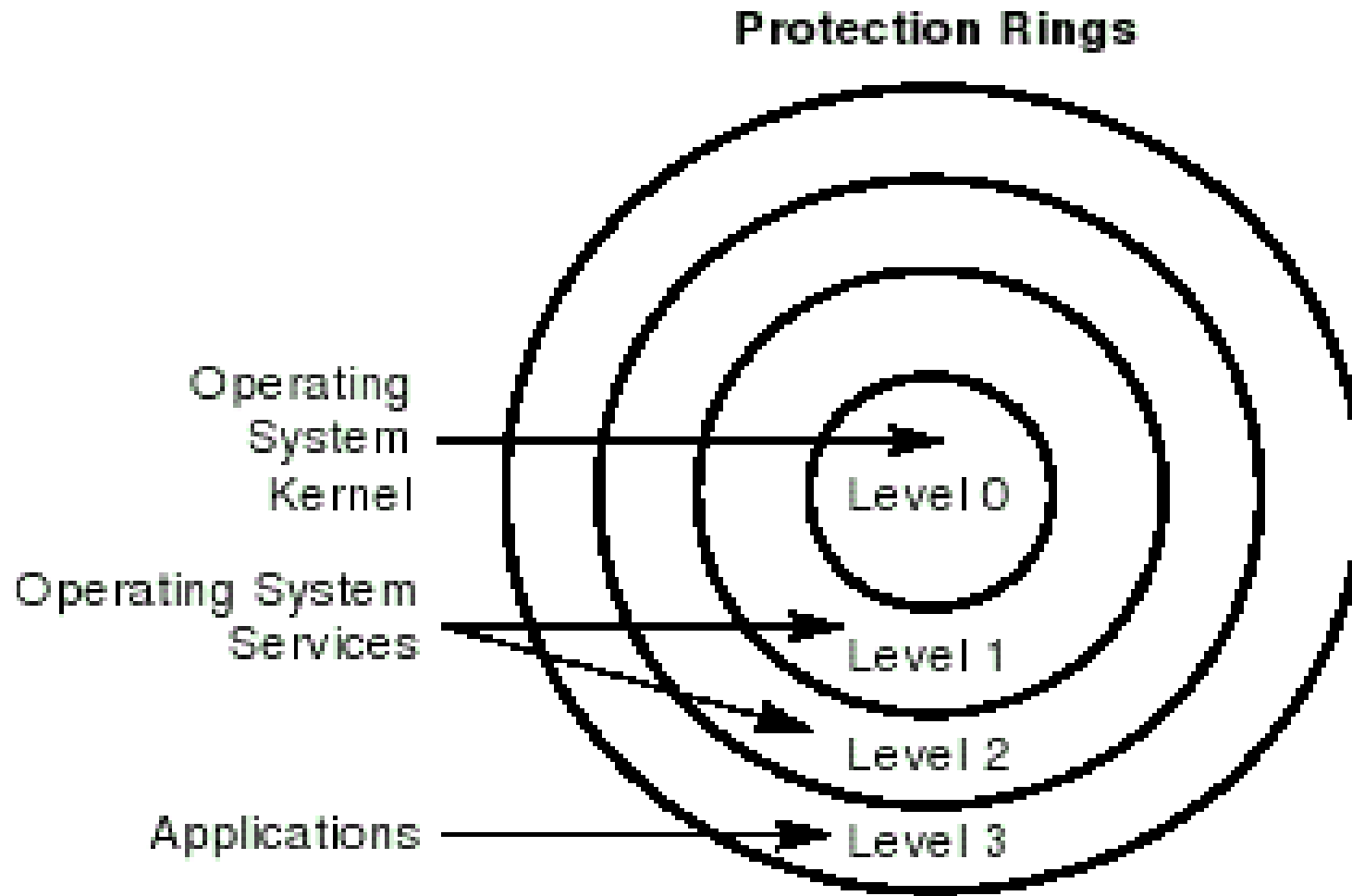
- CR3 yazmacı – PD fiziksel adres göstericisi
- 4 GB erişim için 1 milyar gösterge lazım.
- Page Directory
 - 4KB 1 adet – 1024 girdi --- 4 KB
- Page Table
 - $1024 * 4KB$ --- 40 KB
- Page Frame
 - $1024 * 1024 * 4KB$ ** 4MB
- Ulaşılan Alan
 - 4 GB alanın tamamına ulaşım için ~4MB yeterli oluyor

MMU (Memory Management Unit)

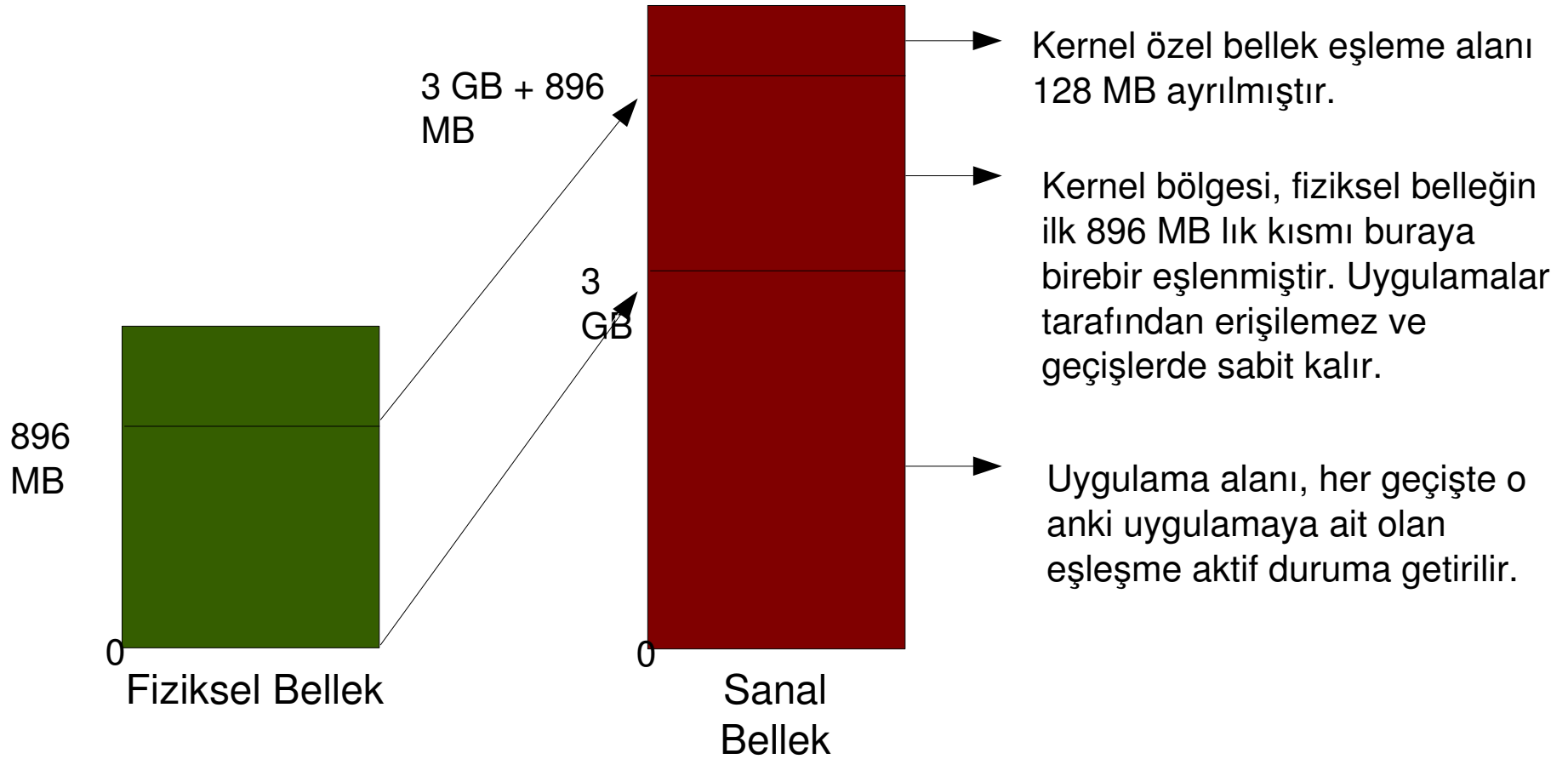


*32 bits aligned onto a 4-KByte boundary.

MMU (Memory Management Unit)



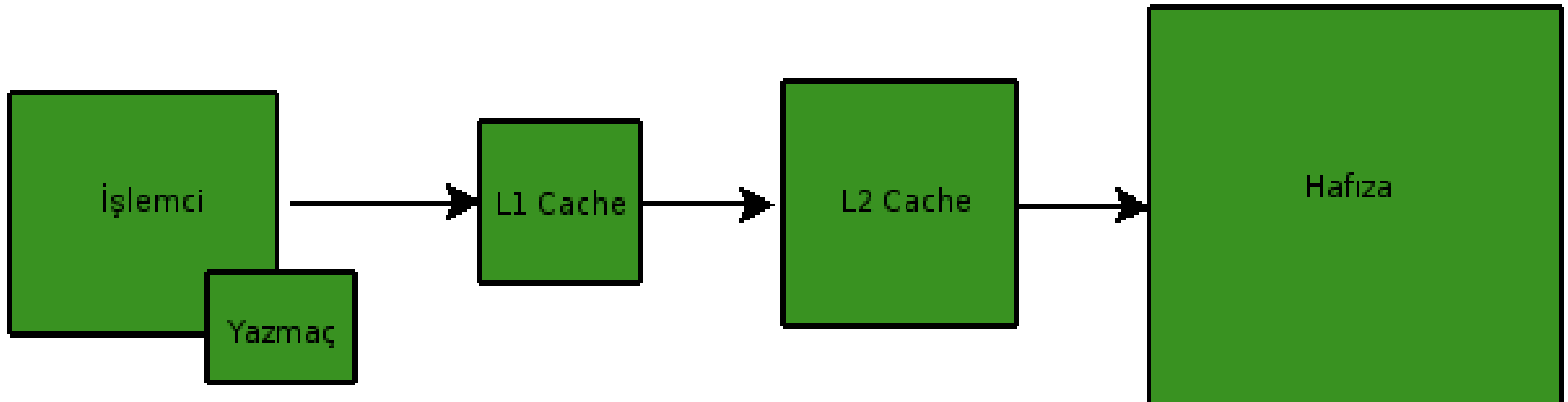
MMU (Memory Management Unit)



Cache Yapıları

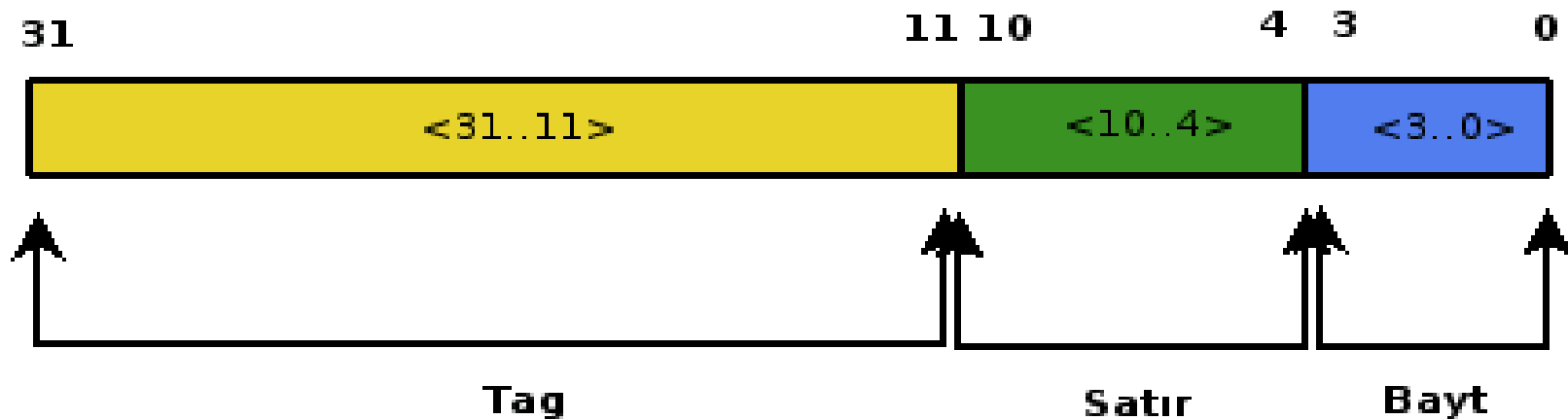
Cache Yapıları

- Hafıza erişim hızındaki yavaşlık sebebiyle cache yapıları ortaya oluşturulmuştur.
- Yazmaç – 0 clock cycle
- L1 Cache – 1-2 clock cycle
- L2 Cache – 5 clock cycle
- Hafıza – 100 – 1000 clock cycle



Cache Yapıları

- Cache satırı nedir?
 - (2^7 – 128 satır)
 - (2^4 – 16 byte)



Cache Yapıları

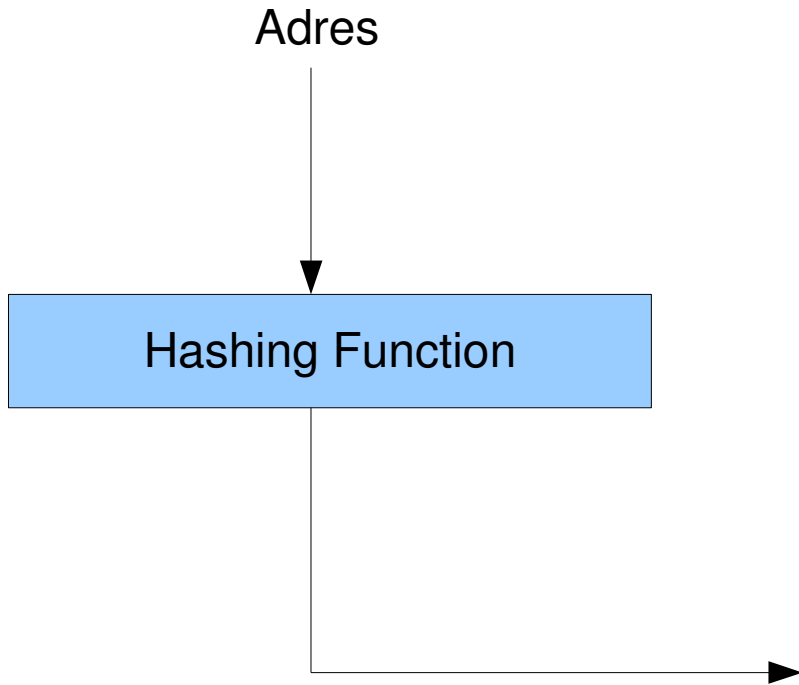
- Cache replacement policy
 - LRU
 - Rasgele
- Yazma politikası
 - write-back (kirli olarak işaretle)
 - write-through
- Cache flush
- Uncached kısımlar
- SMP sistemlerde cache coherency

Cache Yapıları

- Direk Eşleştiren (Direct Mapped)
- 2-Satır Eşli (Two-way set associative)
- n-Satır Eşli (n-way set associative)
- Tam Eşli (Full associative)

Cache Yapıları

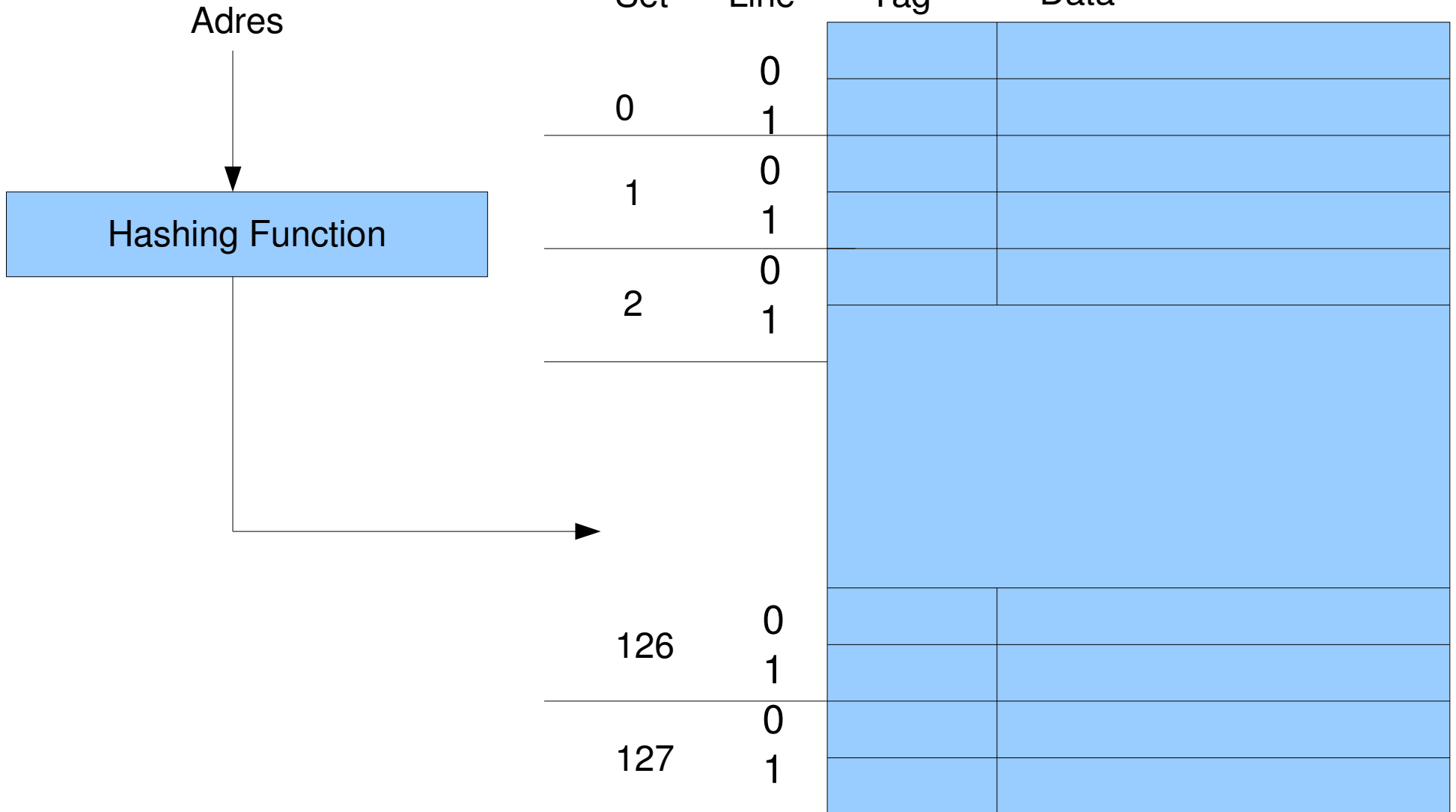
Direk Eşleştirilen



Line	Tag	Data
1		
2		
3		
4		
5		
...		
124		
125		
126		
127		

Cache Yapıları

2-satır eşli

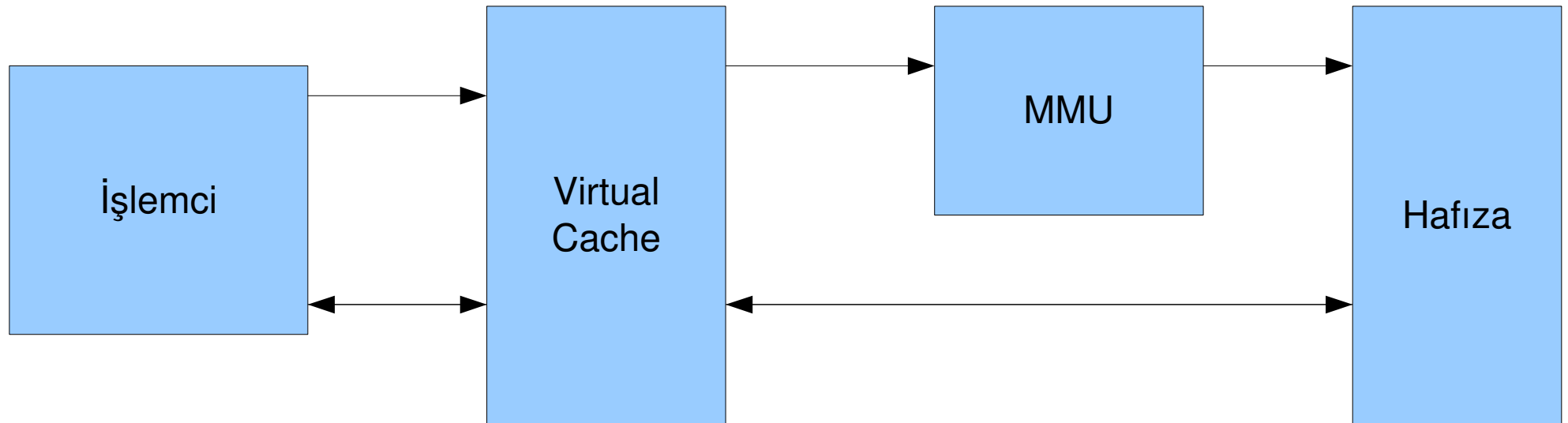


Cache Yapıları

- Tam Eşli (Full associative)
 - TLB (Translation Lookaside Buffer)

Cache Yapıları

- Sanal Cache (Virtual Cache)



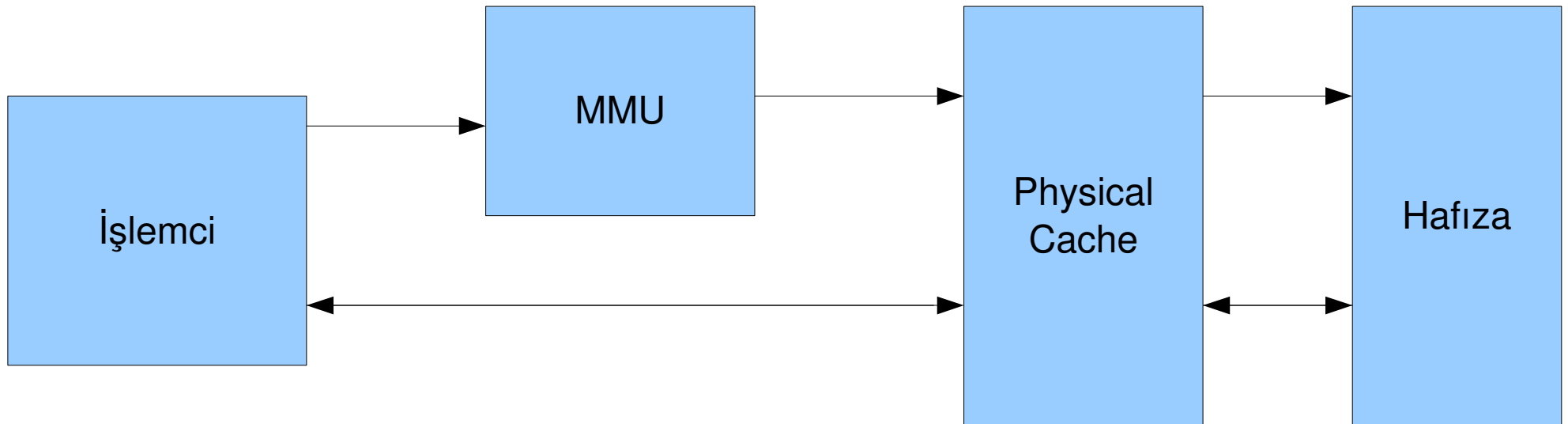
Cache Yapıları

- Sanal Cache (Virtual Cache)

- Ambiguite problemi – Aynı anda çalışan süreçlerin sanal bellek bölgelerindeki adresler aynı anda cachelenebilir.
 - Context switch sonrası problemi.
- Alias problemi – Birden fazla sanal bellek adresi aynı fiziksel adrese işaret edebilir. Paylaşılan bir fiziksel bölge olabilir.
 - İşletim sistemi bu durumu nasıl çözmeli?
 - Cachlenmemiş olarak map edebilir.
 - Birden fazla map'e izin vermeyebilir.
 - Sadece bir alias'a izin verilebilir. Diğer alias erişiminde page fault verdirebilir. Eski olan flush edilip, invalid yapılır, yenisi valid yapılır.

Cache Yapıları

- Fiziksel Cache (Physical Cache)

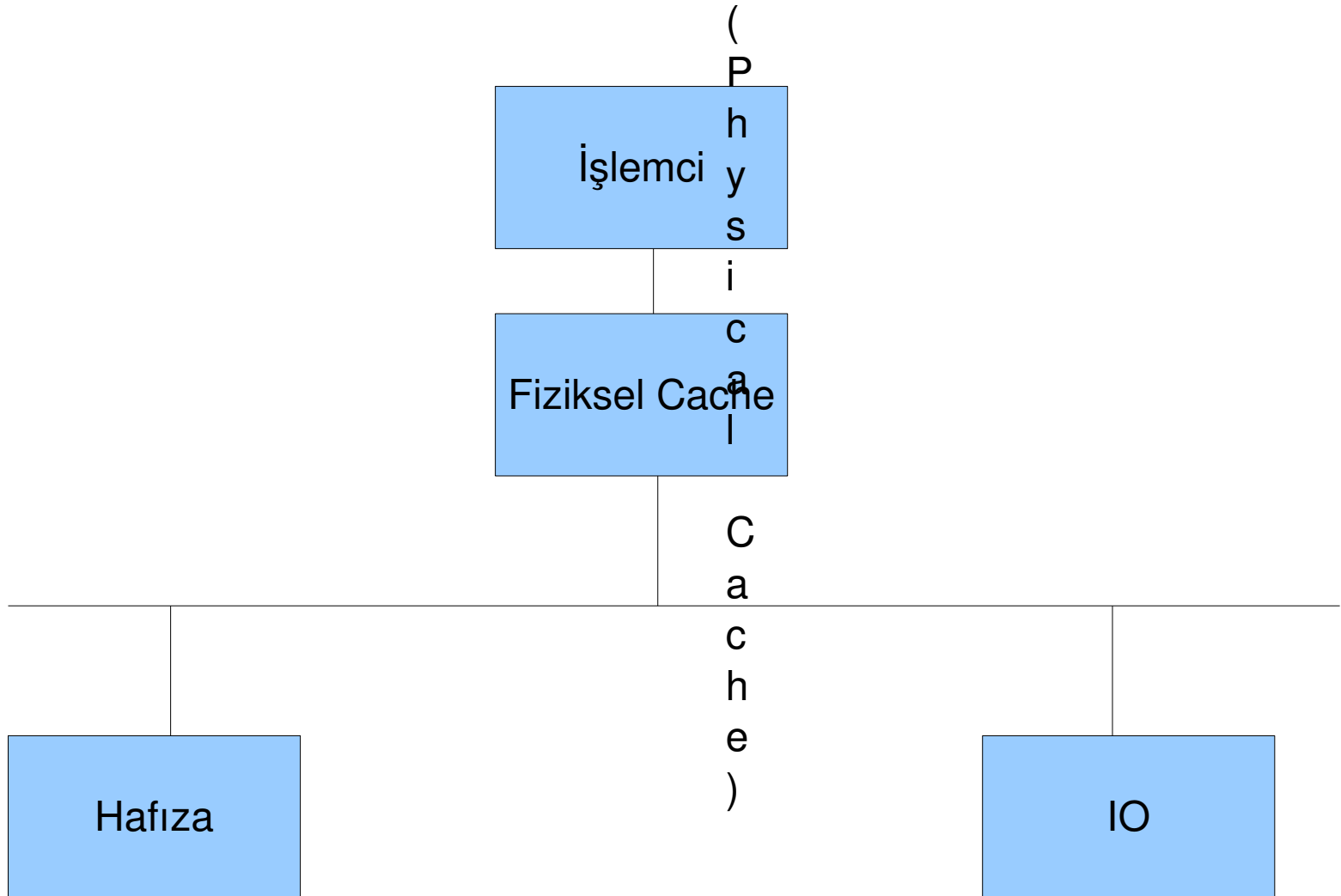


Cache Yapıları

- Fiziksel Cache (Physical Cache)
 - Alias ve Ambiguity problemi bu yapılarda yoktur. Ancak adres dönüşümünü beklemek zorunlu.
 - Virtual cacheler gibi dma işlemleri ile uyumluluğu sağlamak için flush edilmezler.
 - Bus izleme tekniği kullanırlar. (cache snooping)

Cache Yapıları

- Fiziksel Cache (Physical Cache)



Cache Yapıları

- Key tagli sanal cache (process id)
 - context switchlerde flush edilmemesi bir artıdır.
- Fiziksel adres tagli sanal cache
 - Cache arama adres çözümlenmesine bağlı.

Cache Yapıları

- Cache yapılarının farklılıkları işletim sisteminde çeşitli işler için farklı ihtiyaçlar gerektirir.
- Bu ihtiyaçların birleşimini içeren çözümler farklı platformlarda aynı kodun çalışmasını sağlar
 - context switch
 - fork
 - exec
 - exit
 - sbrk/brk
 - shared memory
 - system call

IO

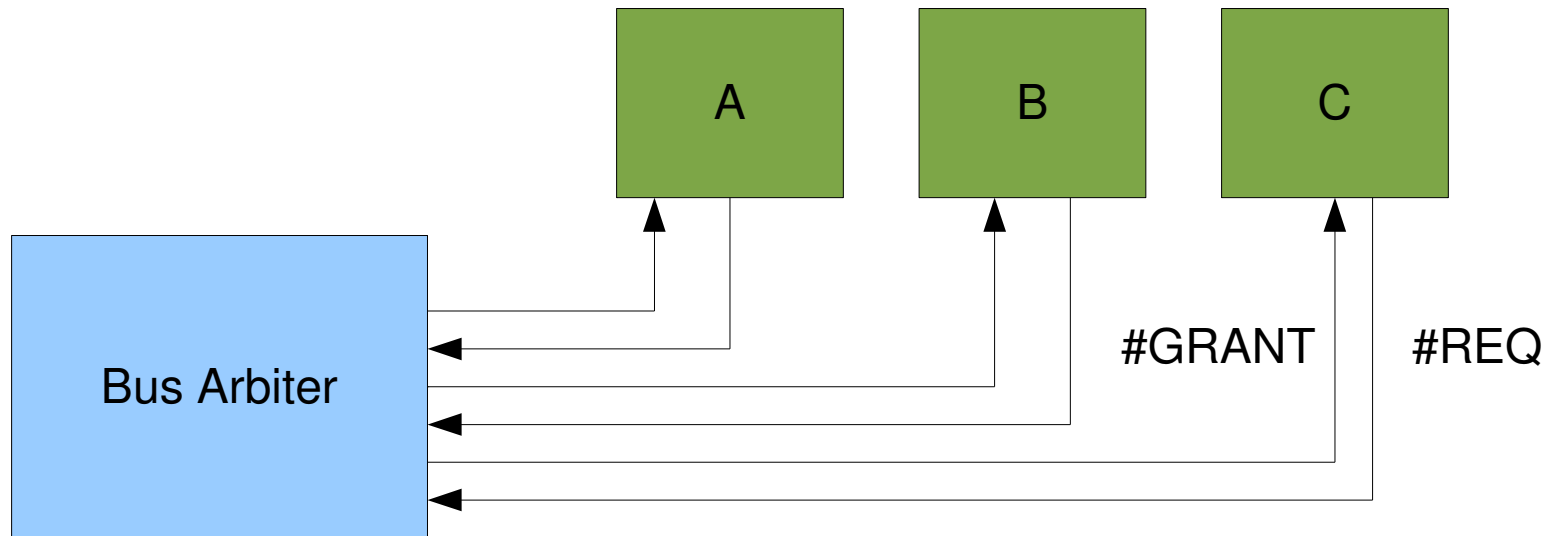
- Bus nedir?
 - Bilgisayar sisteminde farklı alt sistemlerin birbirleriyle iletişimleri bir şekilde sağlanmalı.
 - Bus bu alt sistemlerin birbirlerine ortak bir veri hatı üzerinden bağlandığı sistemdir.
 - Busların dezavantajları bu ortak hat üzerinde haberleşme darboğazları oluşturmalarıdır.
 - Kontrol-Address-Veri hatlarından oluşur.

IO

- Bus bantgeniřliđi
 - Bus'ın maksimum hızını bus'ın uzunluđu ve bađlı olan cihazların sayısı belirler.
- Bus arbitration
 - En yüksek önceliđe sahip olan önce servis edilmeli
 - En düşük öncelikli cihaz da bus'ı kullanılabilmeli.
 - Daisy chain ve Centralized arbitration modelleri.

IO

- Bus mastering nedir?

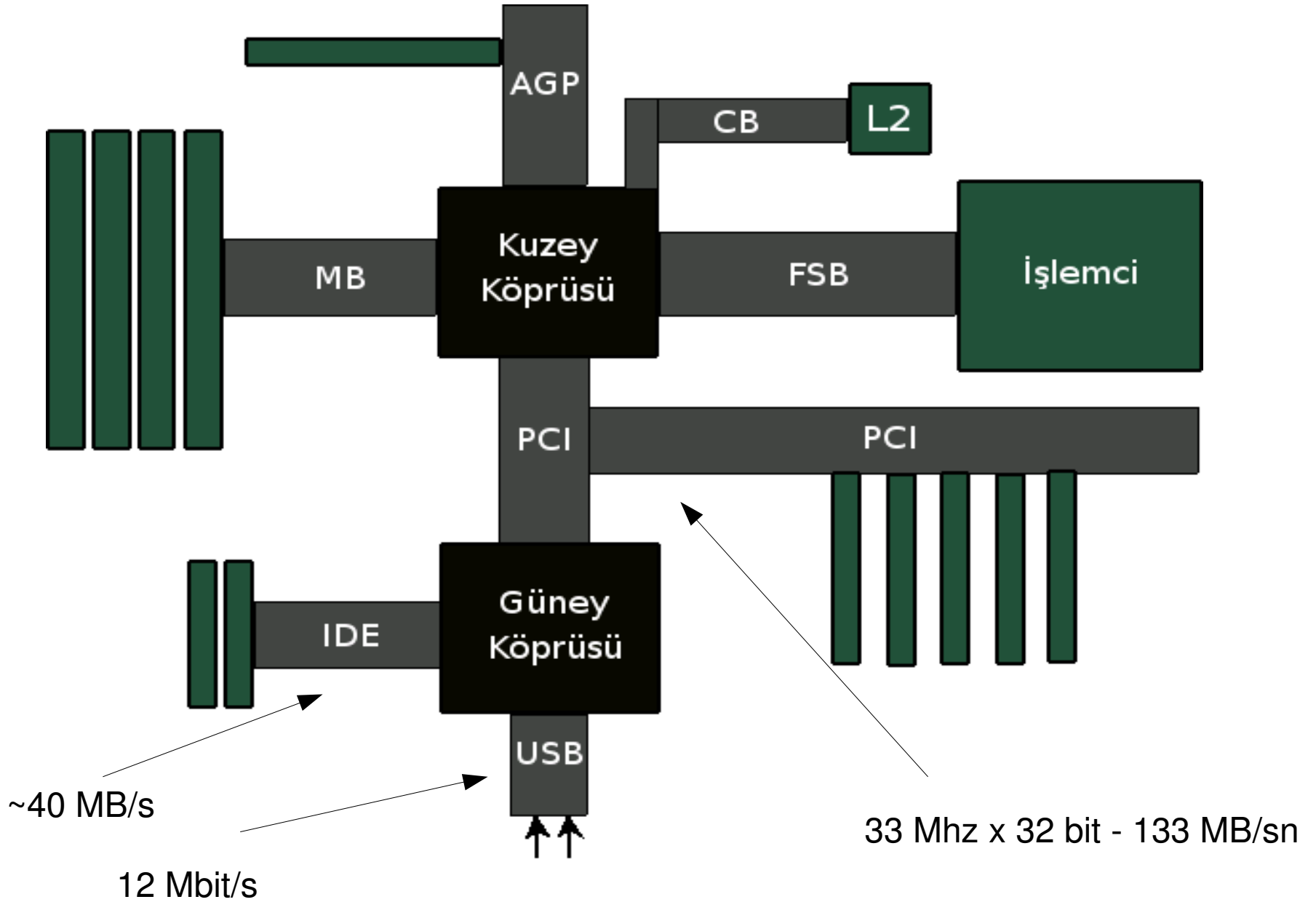


IO

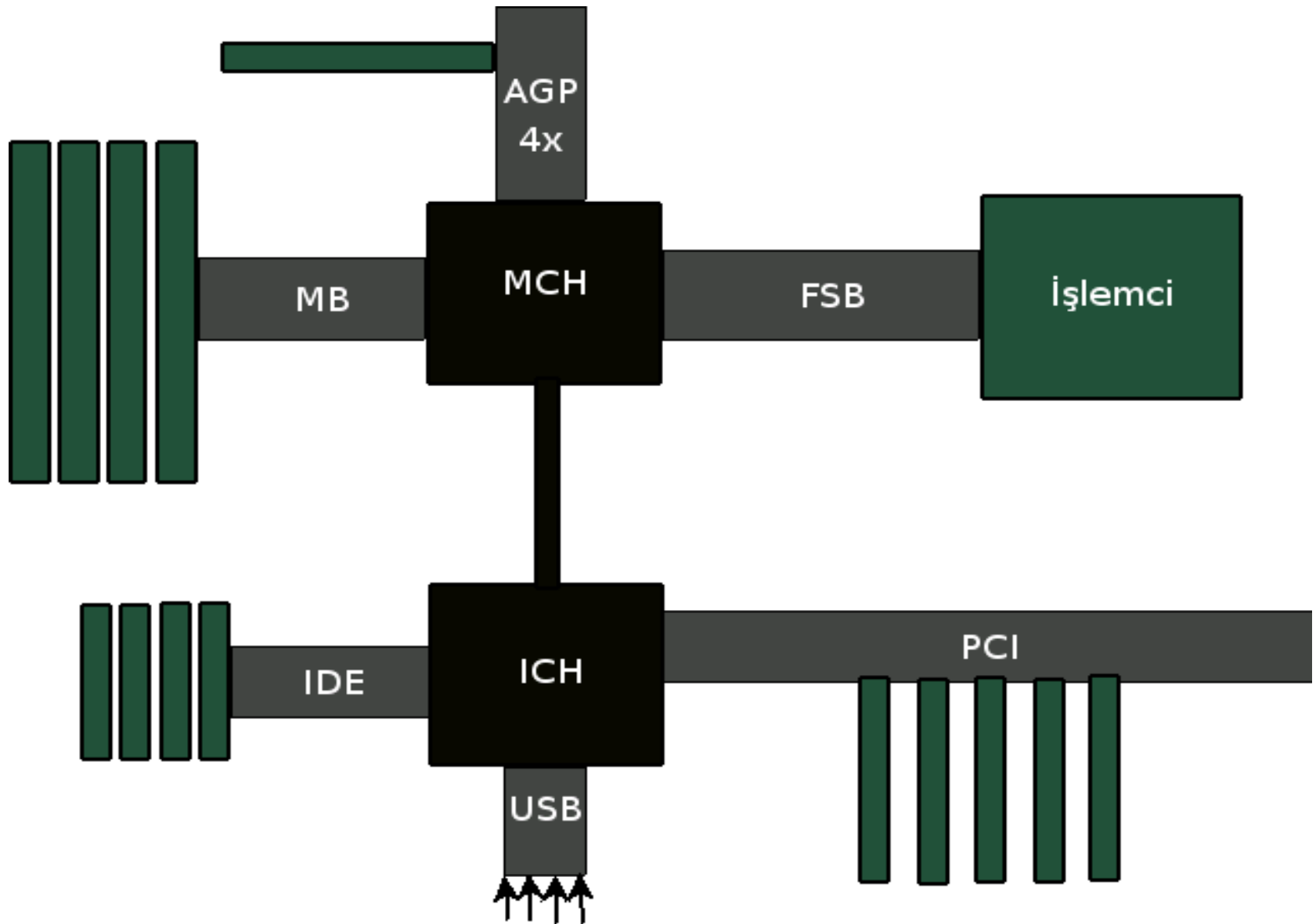
– DMA nedir?

- DMA (Direct Memory Access) cihazların işlemci'yi kullanmadan hafızaya erişebilmelerini sağlar.
- Anakart üzerinde DMA controller bulunmaktadır. (Intel 8237)
- PCI içerisinde var olan bus mastering özelliği PCI cihazlarda bu işi yapar.

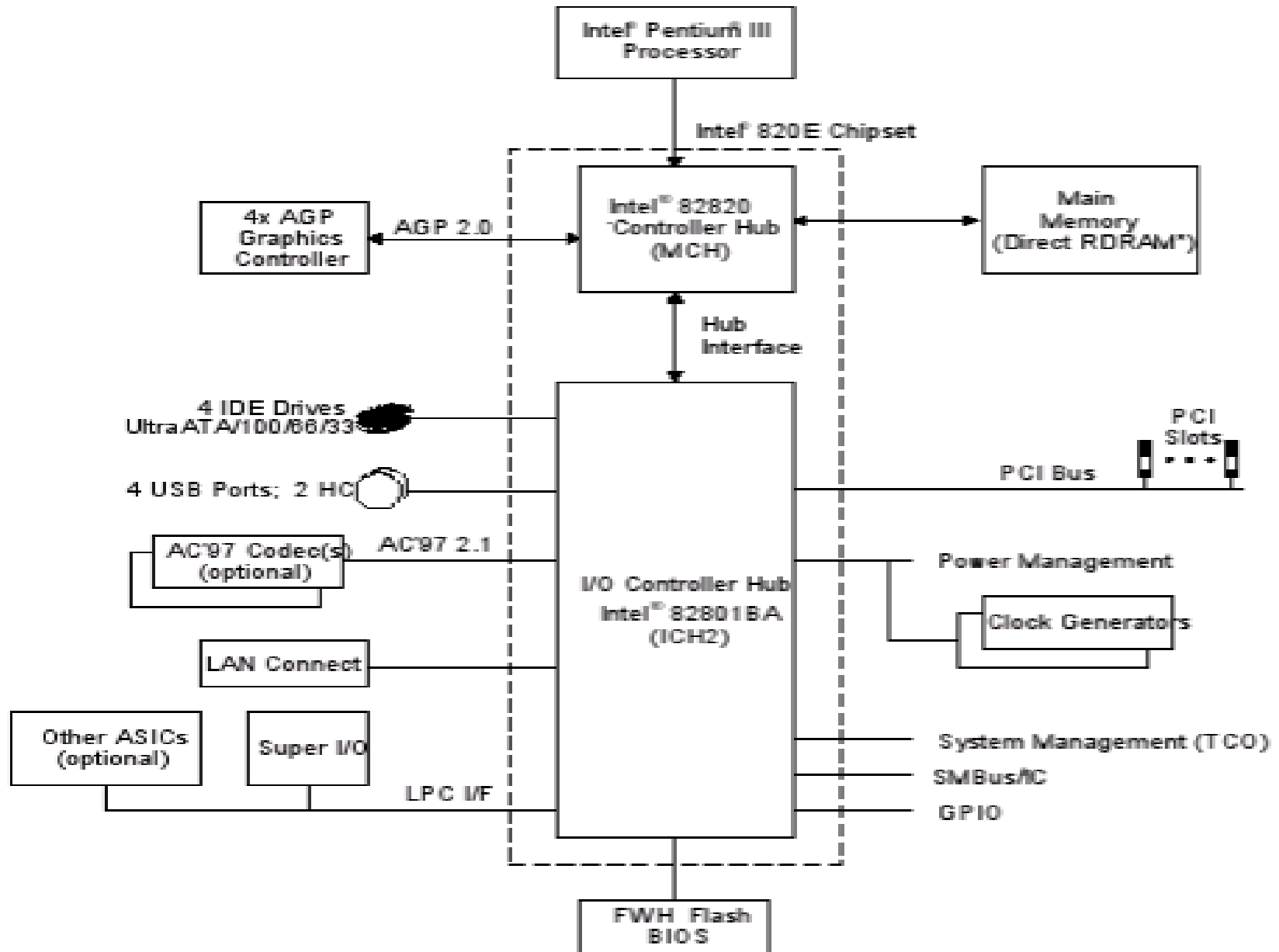
IO



IO



IO



Ordering

Ordering

- Compiler Ordering
- Compiler Barrier
- Memory Ordering
- Memory Barriers

Compiler Ordering

- Compiler optimizasyon amaçlı instructionların yerlerini değiştirebilir, kaldırabilir.
- asm volatile keyword'ü optimizasyon yapmasını engeller. Yine de compiler akışta hiç ulaşılamayacak bir kod varsa burayı kaldırabilir.
- c volatile her zaman bu değişken hafıza'dan alınsın demek ve gereksiz yere optimizasyonu ve performansı engelleyebilir.
- `*(volatile int *)addr = foo;`
- `asm volatile ("eieio" ::);`

Compiler Barrier

- Compiler register'da cachlediği deęeri kullanmaya devam edebilir.
- cachlenmiř tüm registerlar hafızaya.
- herřeyi c volatile yapacađına memory clobber kullan.
- `#define barrier() asm volatile ("": : : "memory");`
 - `#define local_irq_disable() __asm__ __volatile__ ("cli": : : "memory")`
 - `#define local_irq_enable() __asm__ __volatile__ ("sti": : : "memory")`

Compiler Barrier

```
int del_timer(struct timer_list * timer)
{
    int ret = 0;
    if (timer->next) {
        unsigned long flags;
        struct timer_list * next;
        save_flags(flags);
        cli(); <----- Bariyer
        if ((next = timer->next) != NULL) {
            (next->prev = timer->prev)->next = next;
            timer->next = timer->prev = NULL;
            ret = 1;
        }
        restore_flags(flags);
    }
    return ret;
}
```

Memory Ordering

- İşlemci optimizasyon amaçlı hafıza erişim rutinlerini program sırası dışında çalıştırabilir.
- Sistemdeki bir cihazın yazmaçları sistemde hafıza'ya eşleştirilmiş ise rutin çalışma sırası önemli.
- İşlemcinin çalıştırdığı rutinler, sistemdeki diğer işlemciler tarafında aynı sırada görülmeyebilir.

Memory Barriers

- İşlemci optimizasyon amaçlı gerçekleştirdiği çalıştırma sırasını, programın sırasında çalıştırmaya zorlanabilmesi için bir mekanizma sağlamalı.
- Bariyerler yalnızca işlemci-işlemci ve ya işlemci-cihaz arası çalışma durumlarında gerekir.
- yazma bariyeri
- veri bağımlılık bariyeri
- okuma bariyeri
- genel bariyer

Memory Barriers

x86

```
#define mb() alternative("lock; addl $0,0(%%esp)",  
"mfence", X86_FEATURE_XMM2)
```

```
#define rmb() alternative("lock; addl $0,0(%%esp)", "lfence",  
X86_FEATURE_XMM2)
```

```
#define wmb() __asm__ __volatile__ ("" : : : "memory")
```

```
#define smp_mb() mb()
```

```
#define smp_rmb() rmb()
```

```
#define smp_wmb() wmb()
```

Memory Barriers

Powerpc

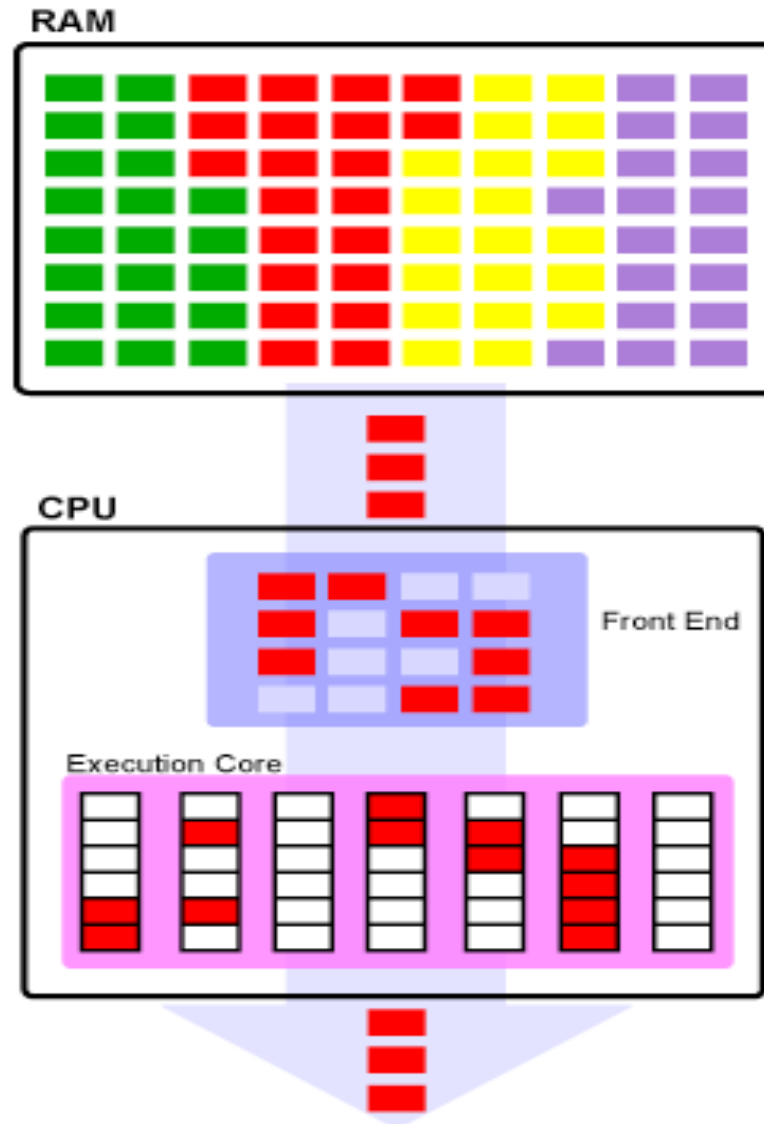
```
#define mb()    __asm__ __volatile__ ("sync" ::: "memory")
#define rmb()  __asm__ __volatile__ ("lwsync" ::: "memory")
#define wmb()  __asm__ __volatile__ ("sync" ::: "memory")

#define smp_mb()    mb()
#define smp_rmb()   rmb()
#define smp_wmb()   __asm__ __volatile__ ("eieio" :::
"memory")
```

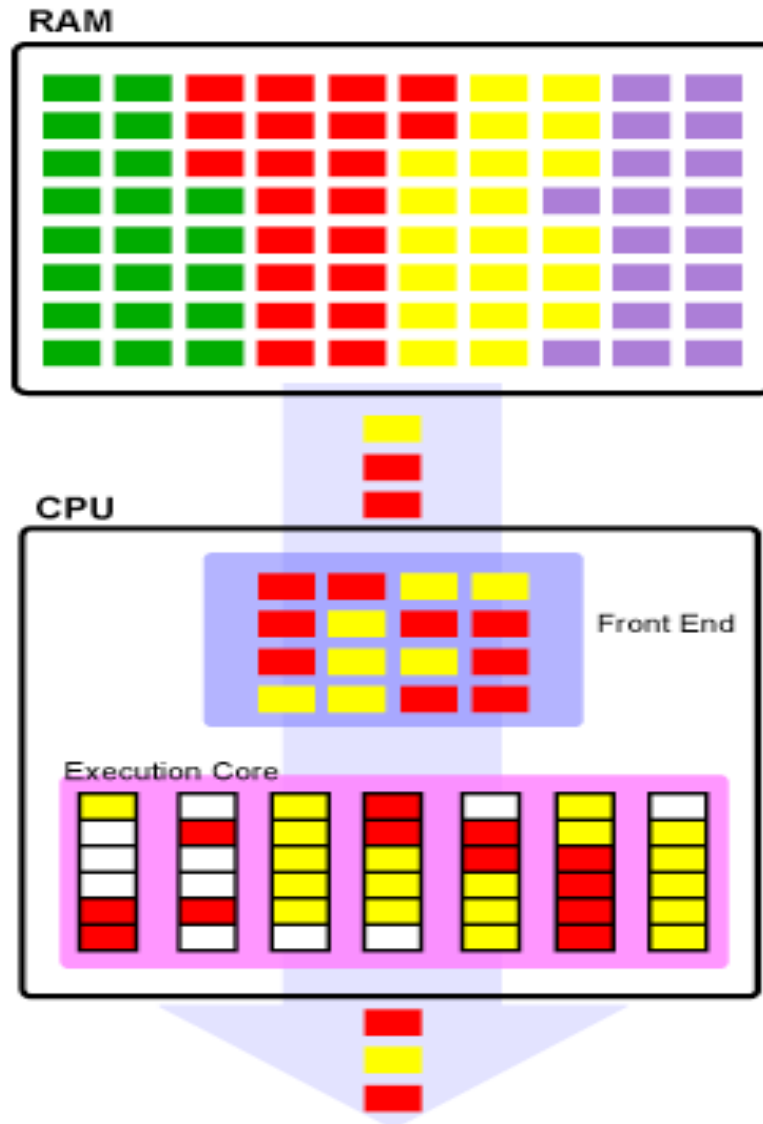
İşlemci sistemleri

- Hyperthreading
- Symetric Multiprocessing (SMP)
- NUMA

Hyperthreading



Hyperthreading



Hyperthreading

- Effects of Hyper-Threading on Linux APIs

– Kernel function	2419s-noht	2419s-ht	Speed-up
– Simple syscall	1.10	1.10	0%
– Simple read	1.49	1.49	0%
– Simple write	1.40	1.40	0%
– Simple stat	5.12	5.14	0%

Hyperthreading

- Effects of Hyper-Threading on AIM9 workload

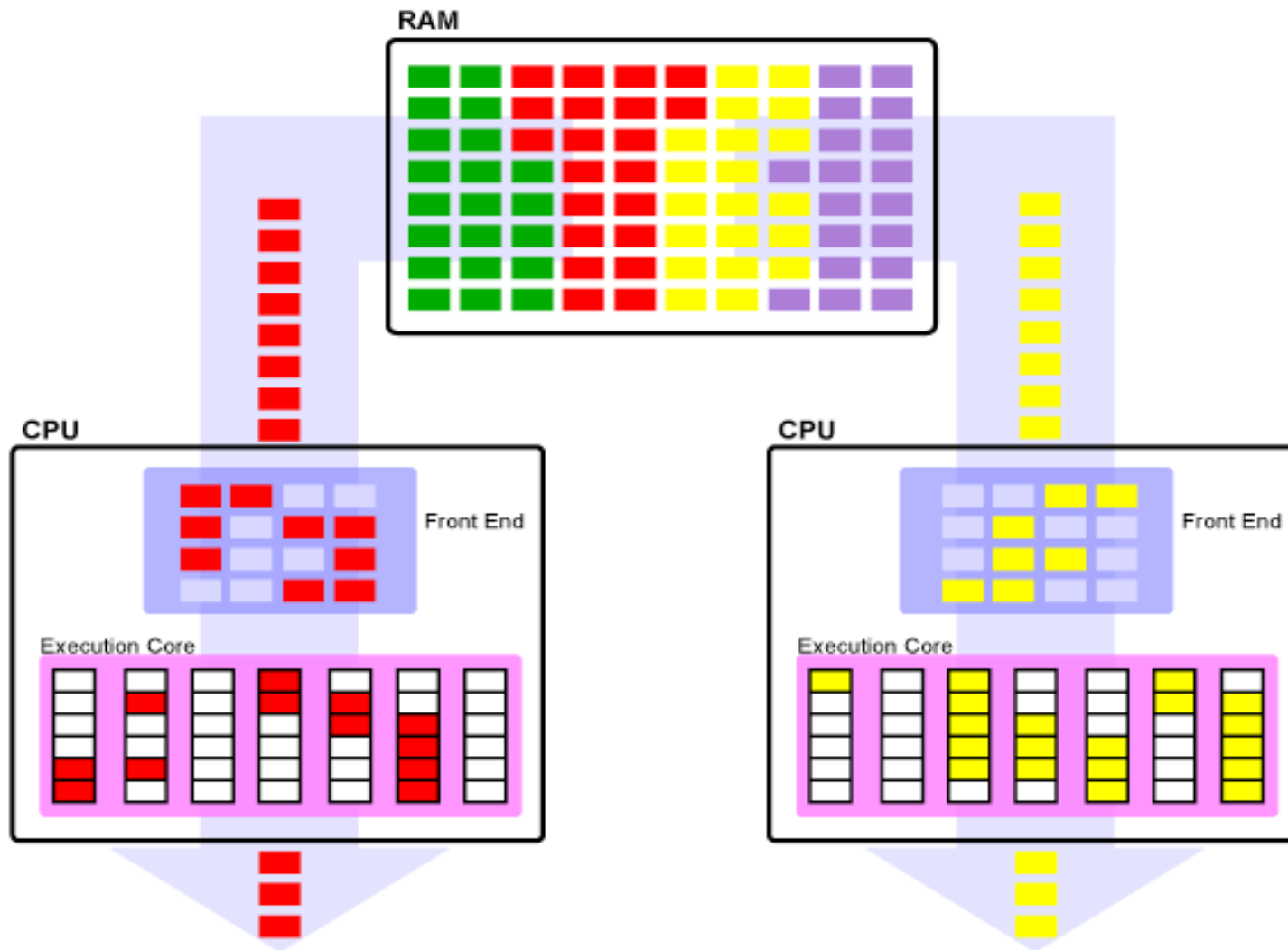
-	2419s-noht	2419s-ht	Speed-up
up			
- add_double	638361	637724	0%
- add_float	638400	637762	0%
- add_long	1479041	1479041	0%
- add_int	1483549	1491017	1%
- add_short	1480800	1478400	0%

Hyperthreading

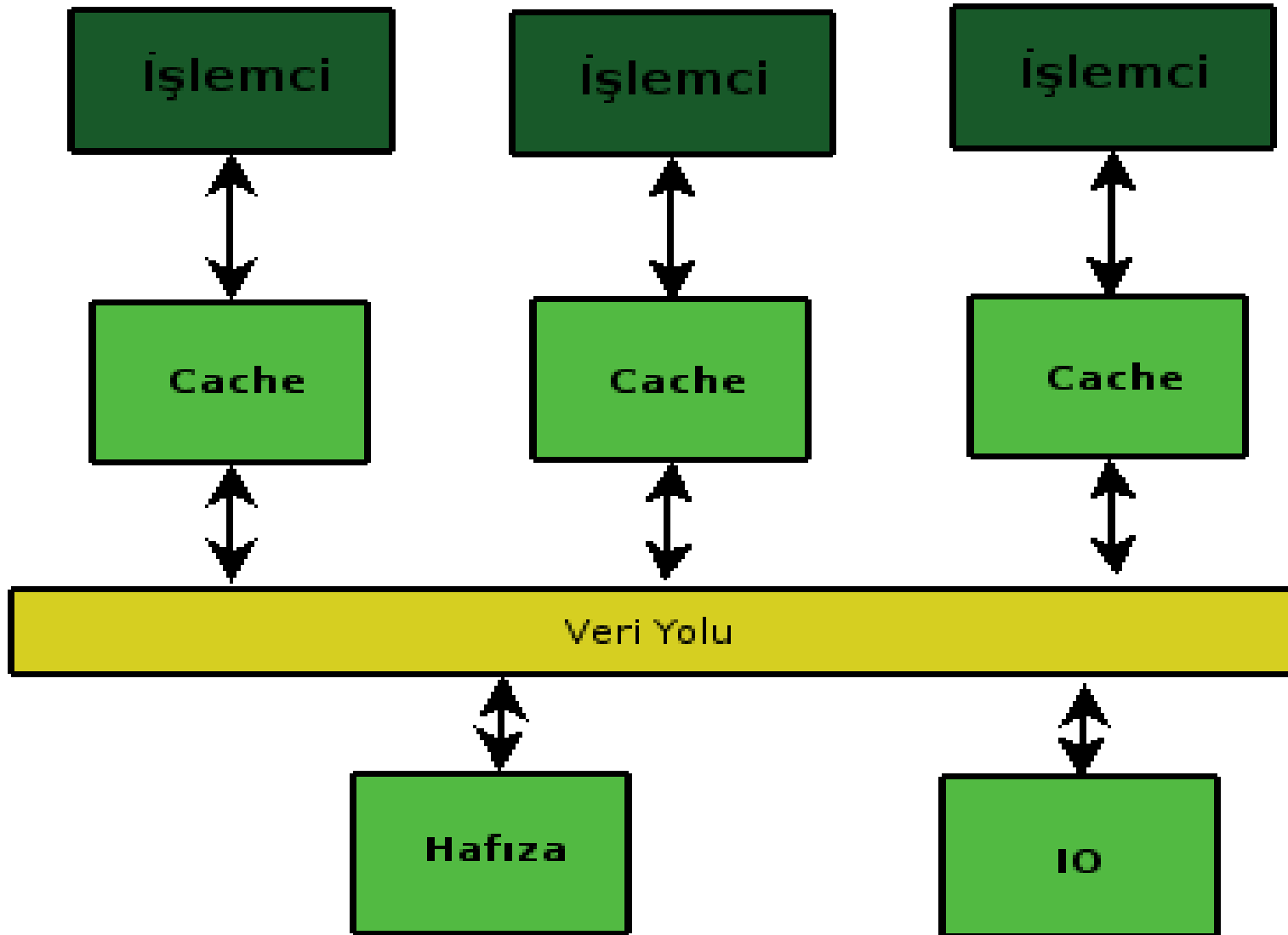
– Chat odası simulasyonu

• Chat rooms	2419s-noht	2419s-ht
Speed-up		
• 20 24%	164,071	202,809
• 30 22%	151,530	184,803
• 40 22%	140,301	171,187
• 50 28%	123,842	158,543

SMP



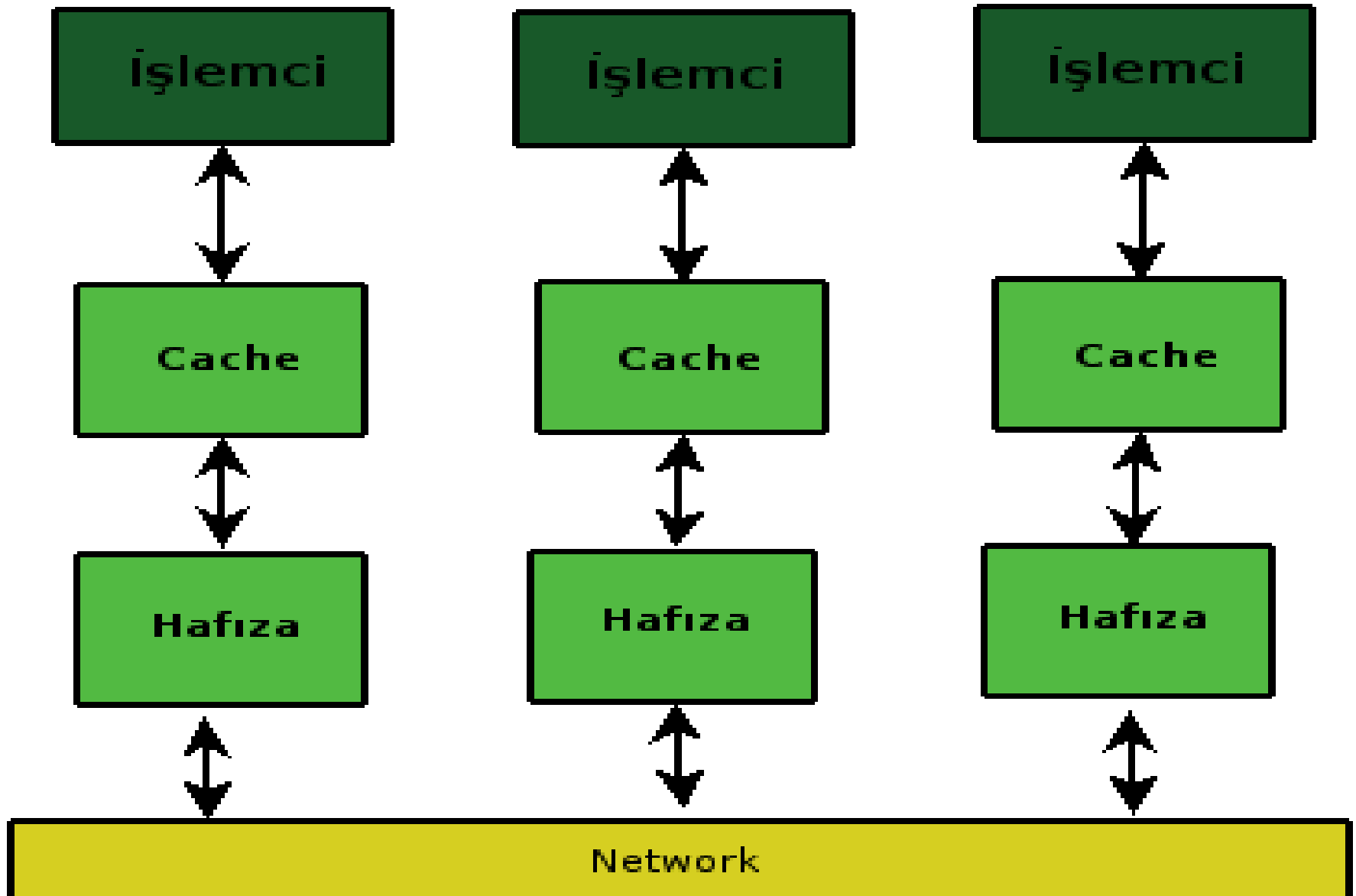
SMP



NUMA

- SMP sistemlerinin ölçeklenebilirlikle alakalı sorunlarını aşmak üzere geliştirilmiş sistemlerdir.
- 8-12 işlemci sonrası bus contention oluşmaya başlar.
- Düğümlerden oluşur: Hafıza aynı bus'dan erişen işlemcilerden oluşur.
- Hafızanın tamamına aynı hızda erişemezsiniz.
- Belli düğümler hafıza erişimlerinde kendilerine ait bus kullanırlar.

NUMA



Çekirdek Kilitleme

- Ölçeklenebilirlik
- Neden Kilitlenmeli
- Kilitleme Yöntemleri
 - Atomic Locks
 - Spinlocks
 - Semaphores
 - RCU
 - Big-Kernel Lock
- Preemption
- Pre-CPU Değişkenler

Ölçeklenebilirlik

- Linux 2.0 ile tek kilit kullanılarak SMP sistemler desteklenmeye başlandı.
- Linux 2.2 ile alt sistemler kendi kilitlerini kullanmaya başladı.
- Linux 2.4 ile kaynak bazlı kilit kullanılmaya başlandı.
- Ölçeklenebilirlik arttı ancak çekirdek programlamak zorlaştı.

Neden Kilitlemeli?

- Interruptlar
- Uyuyan Kodlar
- Preemption
- SMP Sistemler

Atomic Locks

- void atomic_inc(atomic_t *v)
- inc atomic_inc_and_test(atomic_t *v)

```
static __inline__ void atomic_inc(atomic_t *v)
{
    __asm__ __volatile__(
        LOCK_PREFIX "incl %0"
        : "=m" (v->counter)
        : "m" (v->counter));
}
```

Spin Locks

- `void spin_locks(spinlock_t *lock);`
- `void spin_lock_irqsave(spinlock_t *lock,
unsigned long flags);`
- `void spin_lock_irq(spinlock_t *lock);`
- `void spin_lock_bh(spinlock_t *lock);`

Semaphores

- void down(struct semaphore *sem);
- int down_interruptible(struct semaphore *sem);
- int down_trylock(struct *semaphore);
- void up(struct semaphore *sem);

Reader-Writer Locks

- void down_read(struct rw_semaphore *sem);
- int down_read_trylock(struct rw_semaphore *sem);
- void up_read(struct rw_semaphore *sem);
- void down_write(struct rw_semaphore *sem);
- int down_write_trylock(struct rw_semaphore *sem);

RCU

- `rcu_read_lock();`
- `rcu_read_unlock();`
- The data dependency barrier is very important to the RCU system, for example. See `rcu_dereference()` in `include/linux/rcupdate.h`. This permits the current target of an RCU'd pointer to be replaced with a new modified target, without the replacement target appearing to be incompletely initialised.

Big-Kernel Lock

- `lock_kernel();`
- `unlock_kernel();`

Kaynaklar

- Unix Systems for Modern Architectures – Curt Schimmel
- Design of the Unix Operating System – Maurice J. Bach
- Computer Organization and Design – Patterson – Hanessey
- IA-32 Intel Architecture Software Developer's Manual Volume-3
- <http://www.ussg.iu.edu/hypertext/links/kernel/9605.0/0214.html>
(Linus Torvalds'ın memory clobber vaazı)
- <http://www.ussg.iu.edu/hypertext/links/kernel/9507/0008.html>
(David Miller – cache yapılarına göre farklı çalışma durumları)
- <http://arstechnica.com/> - Jon “Hannibal” Stokes
- <http://funix.sf.net>

Sorular

Teşekkürler