Operating System for Multiprocessor System-on-Chip

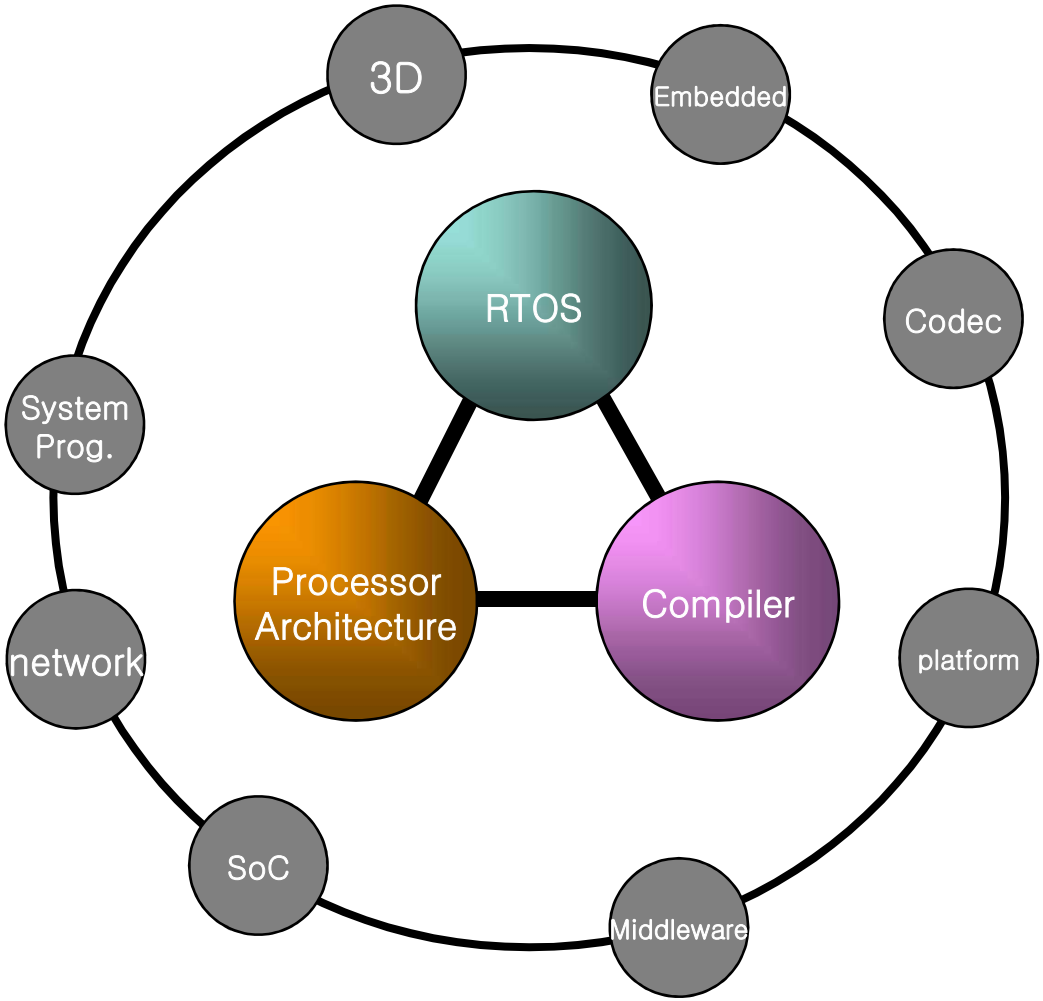
백창우

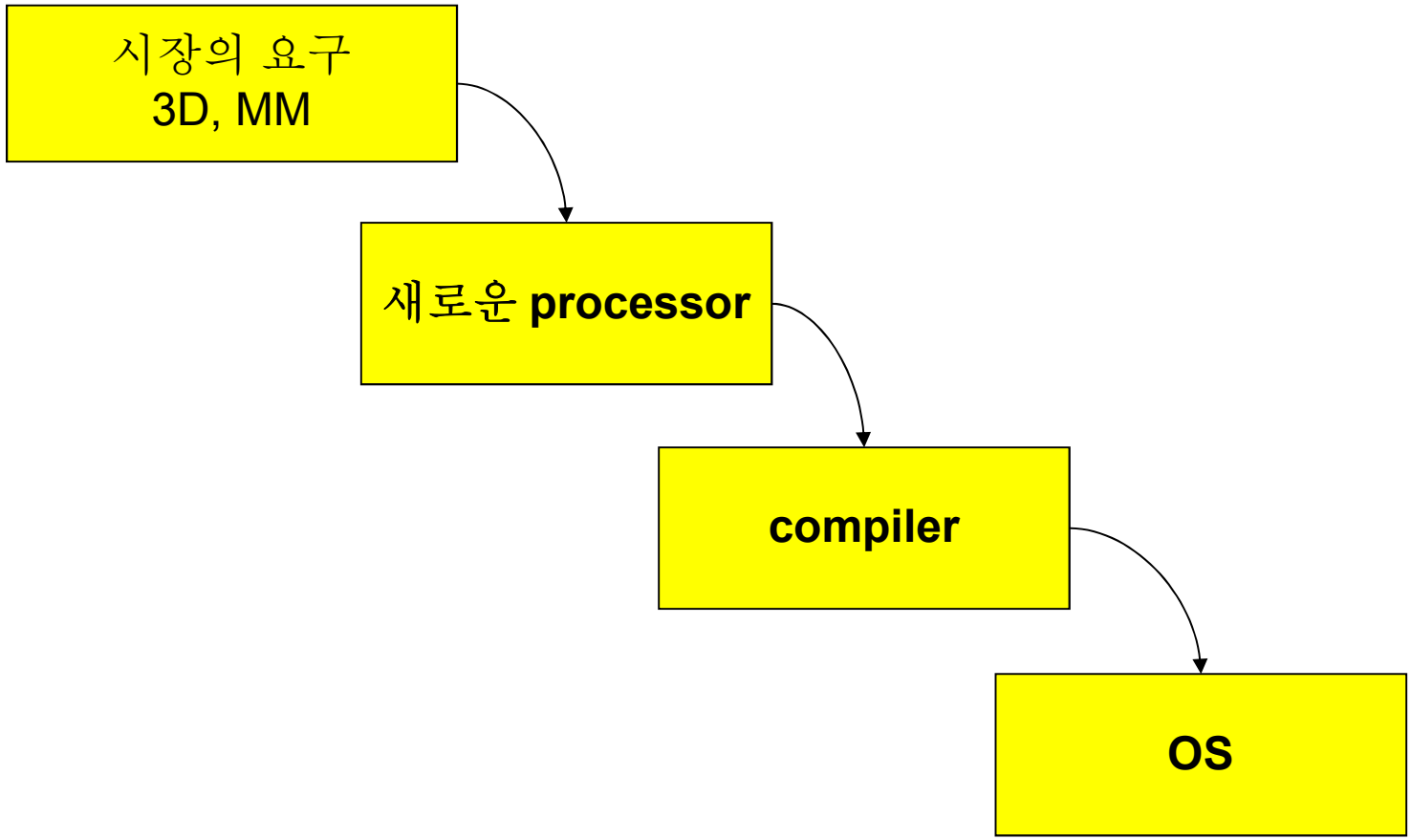
(bckddn@gmail.com)

2007. 05. 08

주요내용 및 방향

- 주요내용
  - Processor Architecture
  - SMP OS


3D
Embedded
RTOS
Codec
System
Prog.
Processor
Architecture
Compiler
network
platform
SoC
Middleware


시장의 요구
3D, MM
새로운 processor
compiler
OS

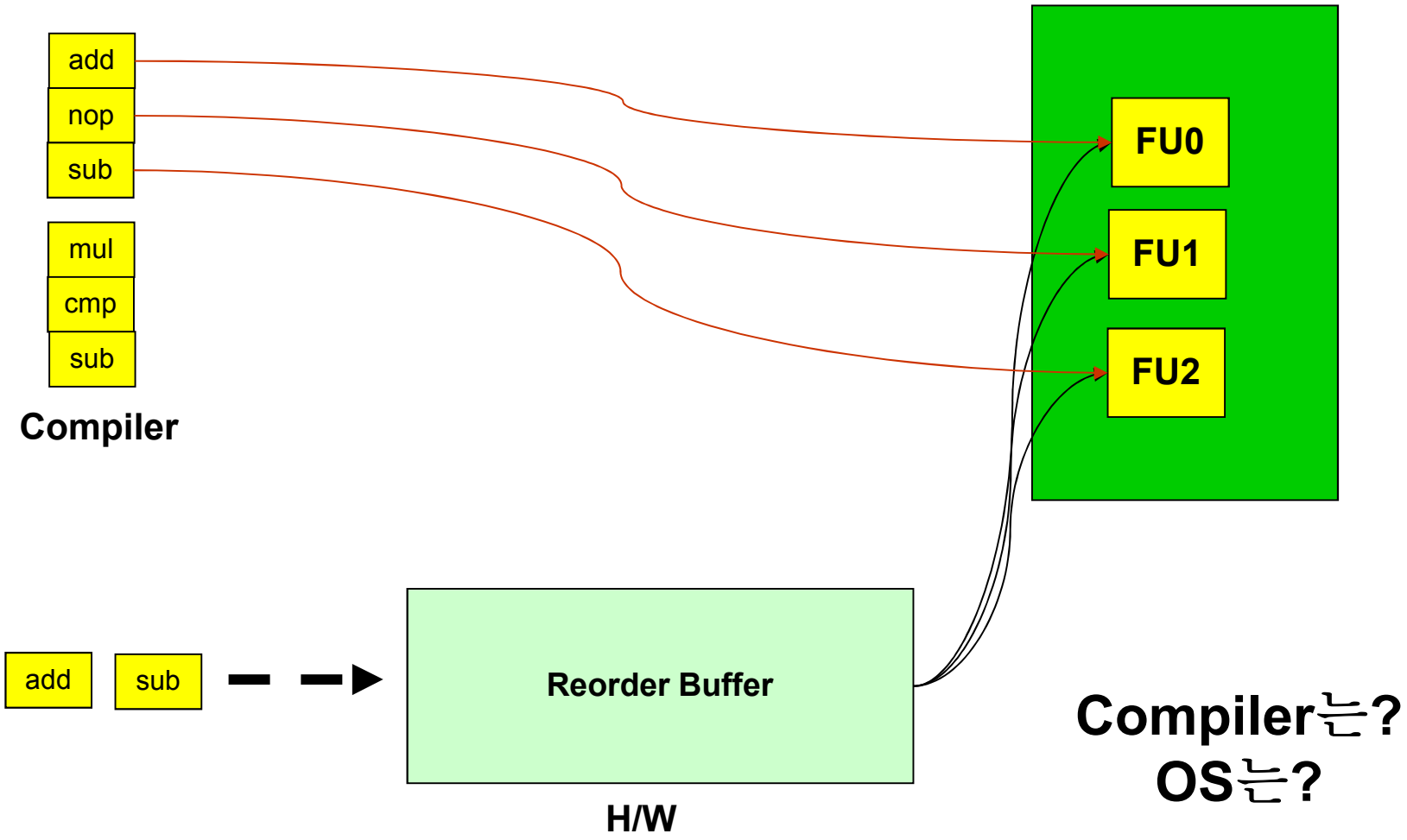
Processor Architecture

- 분류
  - CISC
  - RISC
  - EISC
  - VLIW(EPIC)
  - VLES
  - Superscalar
  - Reconfigurable Processor
  - Multi-core
  - Many-core
  - SMP/AMP
  - Homogenous/Heterogeneous
  - …

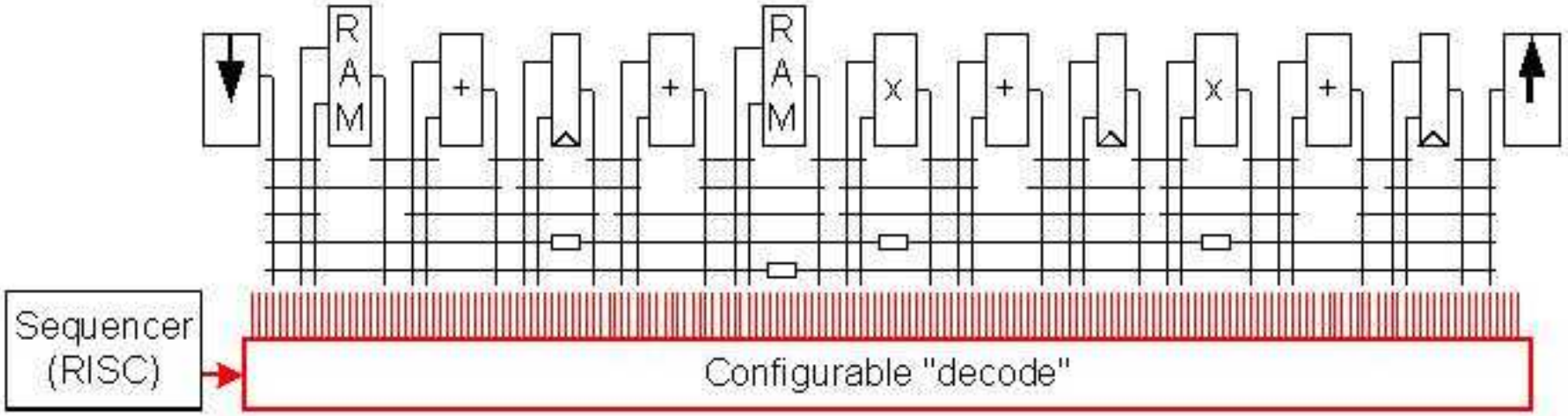
CISC/RISC/EISC

	CISC	RISC	EISC
주요 특성	- 복합 명령어 구조 - 하드웨어 복잡	- **32**비트 고정된 명령어 구조 - 하드웨어 간단	- **16**비트 고정 명령어구조 - 확장 명령어 개발 - 하드웨어 간단
명령어 구조	복 잡	단 순	단 순
상대적 프로그램 크기	**110-130**	**160-180**	**100**(우수)
성 능	**High**	**High**	**High**
약 점	- 복잡한 하드웨어 - **64**비트 이상의 **high-end MCU** 개발이 어려움	- **16**비트/**64bitMCU** 개발이 어려움 - 프로그램 사이즈가 크다	

VLIW / VLES / Superscalar

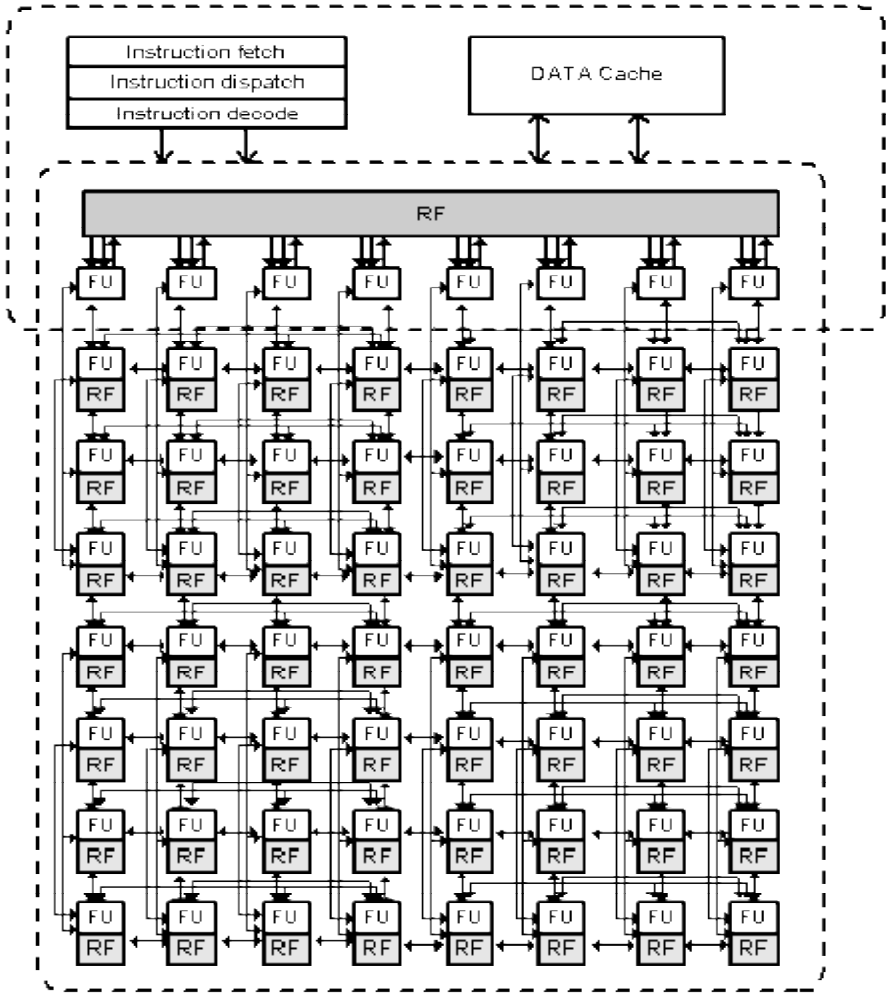


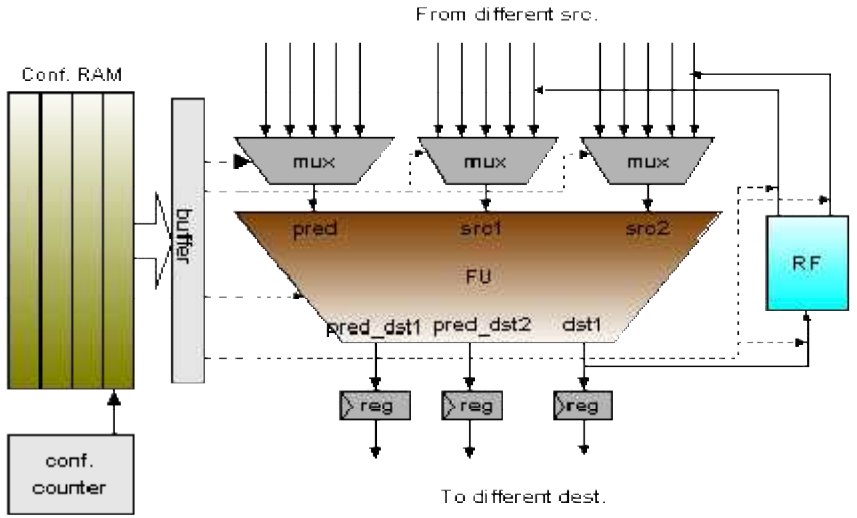
Reconfigurable Processor



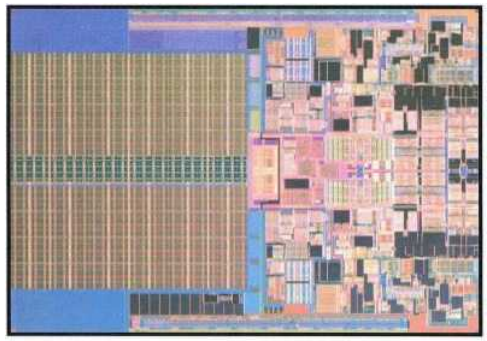
Compiler는?
OS는?

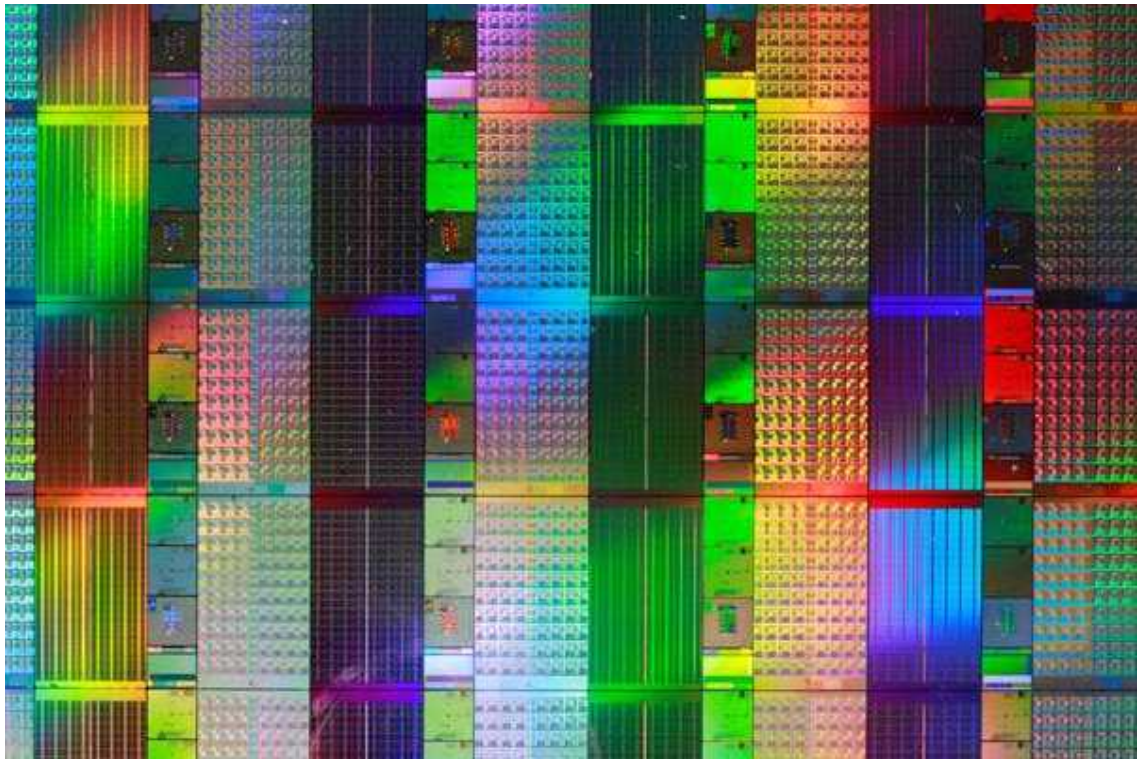
Reconfigurable Processor



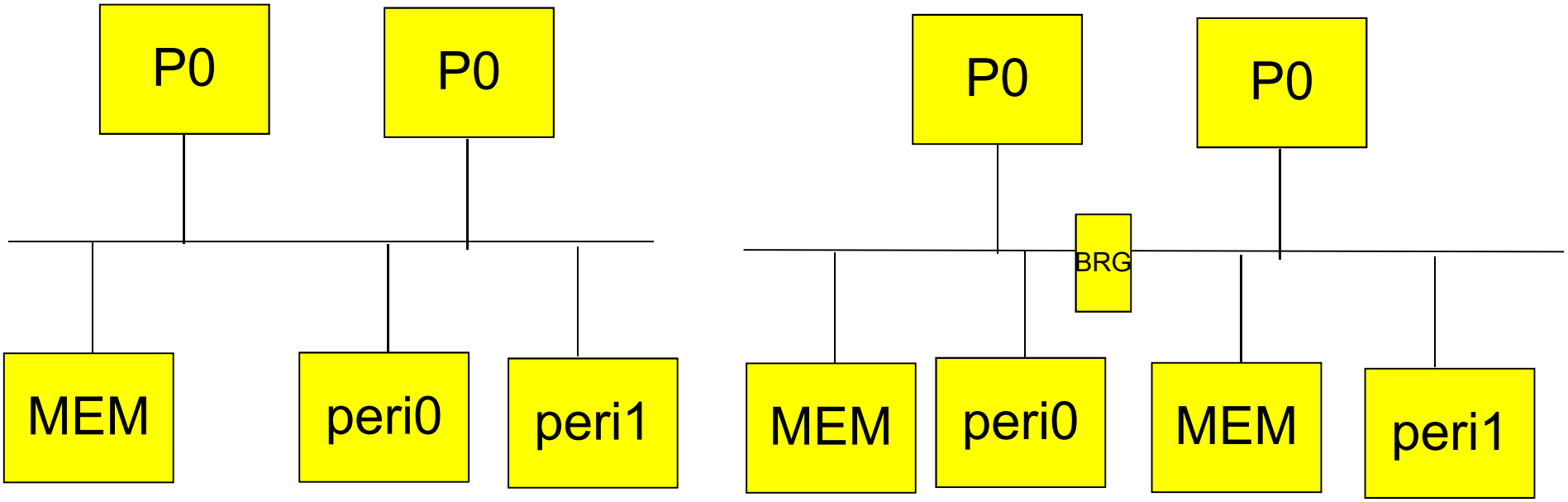


Multi-core / Many-core





SMP / AMP



Homogenous / Heterogeneous

Parallelism

- 구분
  - ILP (instruction level parallelism)
    - 한번에 여러 개의 instruction을 수행
  - LLP (loop level parallelism)
    - Loop을 중첩해서 수행
  - TLP (thread level parallelism)
    - Multi-thread로 수행
  - DLP (data level parallelism)
    - 여러 data를 한번에 수행

1	3
3	2

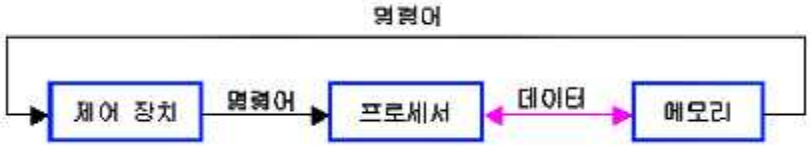
X

2	2
1	5

=

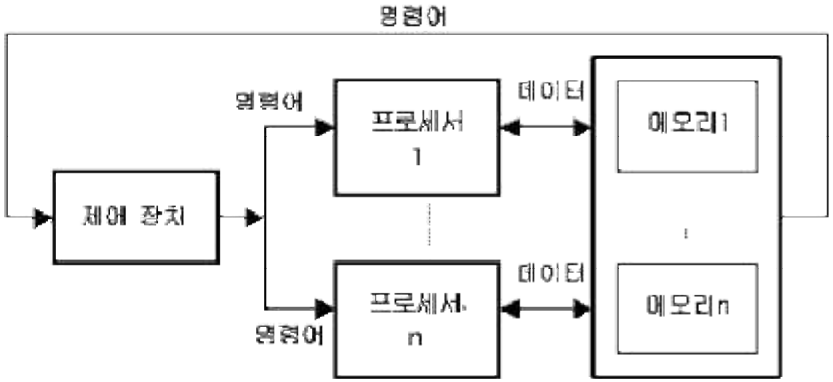
2	6
3	10

Flynn의 분류



SISD(Single Instruction stream Single Data stream)

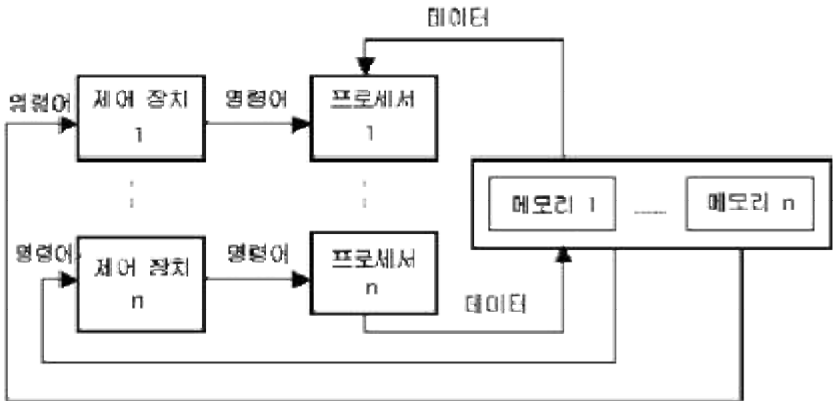
SISD(Single Instruction stream Single Data stream)
- 제어 장치와 프로세서를 각각 하나씩 갖는 구조
- 한 번에 한 개씩의 명령어와 데이터를 처리하는 단일 프로세서 시스템
- 명령어가 순서대로 실행되지만 실행 과정은 여러 개의 단계들로 나누어 중첩시켜 실행 속도를 높이도록 파이프라이닝
(pipelining)으로 되어 있는 것이 보통



SIMD(Single Instruction stream Multiple Data stream)

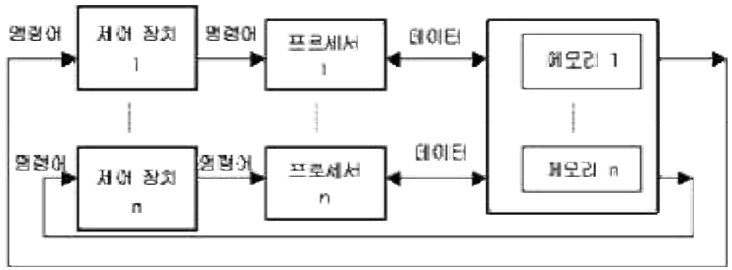
SIMD(Single Instruction stream Multiple Data stream)
- 배열 프로세서와 파이프라인이 이 분류에 속함
- 여러 개의 프로세서들로 구성되고, 프로세서들의 동작은 모두 하나의 제어 장치에 의해 제어
- 모든 프로세서들은 제어 장치로부터 동일한 명령어를 받지만 명령어 실행 과정에서 서로 다른 데이터들을 사용
- 모든 프로세서들이 기억 장치를 공유하는 경우도 있고, 각 프로세서가 기억 장치 모듈을 따로 가지는 분산 기억 장치
구조도 있다.

Flynn의 분류



MISD(Multiple Instruction stream Single Data stream)

MISD(Multiple Instruction stream Single Data stream)
- 여러 개의 제어 장치와 프로세서를 갖는 구조
- 각 프로세서들은 서로 다른 명령어들을 실행하지만 처리하는 데이터는 하나의 스트림
- 하나의 데이터에 대해 여러 명령어를 수행하는 구조
- 프로세서들이 파이프라인으로 연결되어서 한 프로세서가 처리한 결과를 다음 프로세서로 보내는 방식
- 복잡한 데이터 처리 과정을 갖는 특수한 경우에만 사용되고 일반 용도의 컴퓨터 구조로는 사용되지 않음



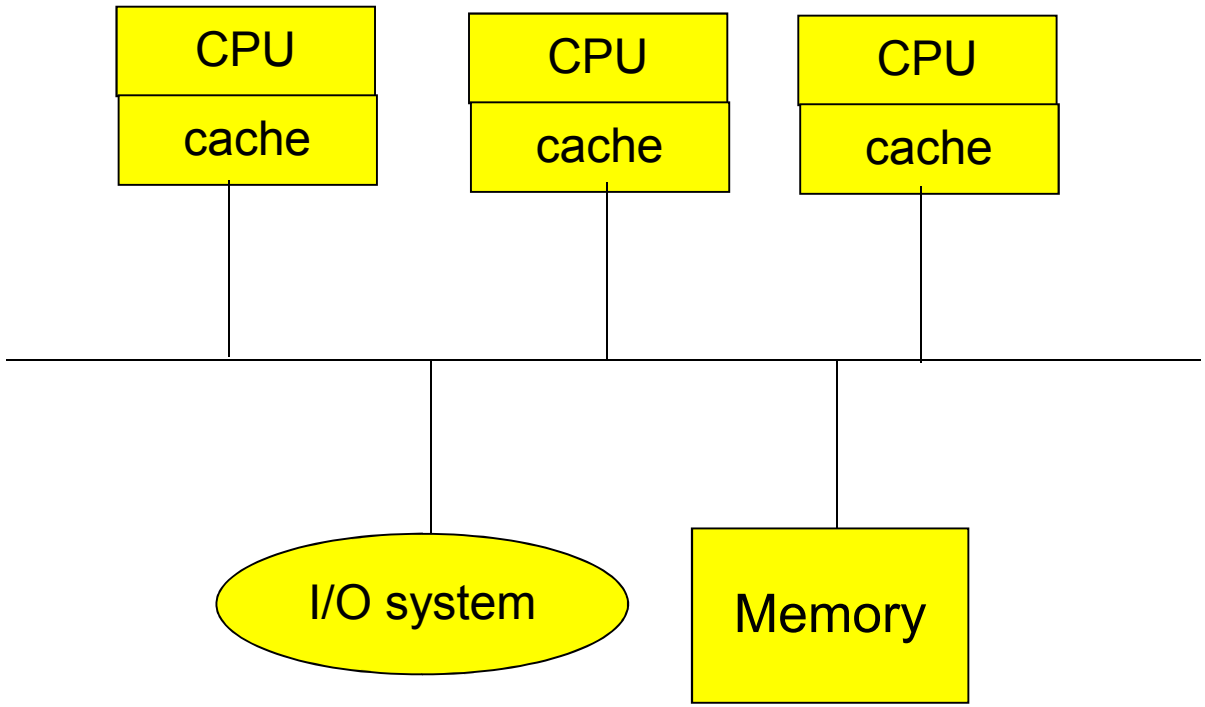
MIMD(Multiple Instruction stream Multiple Data stream)

MIMD(Multiple Instruction stream Multiple Data stream)
- 대부분의 다중 프로세서 시스템과 다중 컴퓨터 시스템이 이 분류에 속함
- 여러 개의 프로세서들이 서로 다른 명령어와 데이터를 처리
- 밀결합 시스템(tightly coupled system) 프로세서들 간의 상호 작용 정도에 따라 그 정도가 높은 구조 밀결합 시스템의
  전형적인 구조는 모든 프로세서가 기억장치를 공유하는 공유기억장치(shared memory) 구조
- 소결합 시스템(loosely coupled system)프로세서들 간의 상호 작용 정도가 낮은 구조 각 프로세서가 자신의 지역 메모
  리(local memory)를 가진 독립적인 컴퓨터 모듈로 구성

SMP (Symmetric Multi-Processor)

- **MIMD Tightly Coupled Shared Memory System**
- **Global physical memory access**
  - 모든 프로세서는 대칭적으로 메모리 접근 가능
- **Each processor has it's own Cache (1 or more level)**
- **Physically shared main memory**
- **Message passing requires a thin software layer**
  - Shared memory, Virtual Interrupt
- 일반적으로 **Shared Bus**로 연결
  - Bus design에 큰 영향을 받음, low cost interconnect
- 일반적으로 **CPU**는 **homogenous**하여 동일 **OS**로 동작
- **Cache Coherency Issue**
- **Memory Consistency**

SMP (Symmetric Multi-Processor)



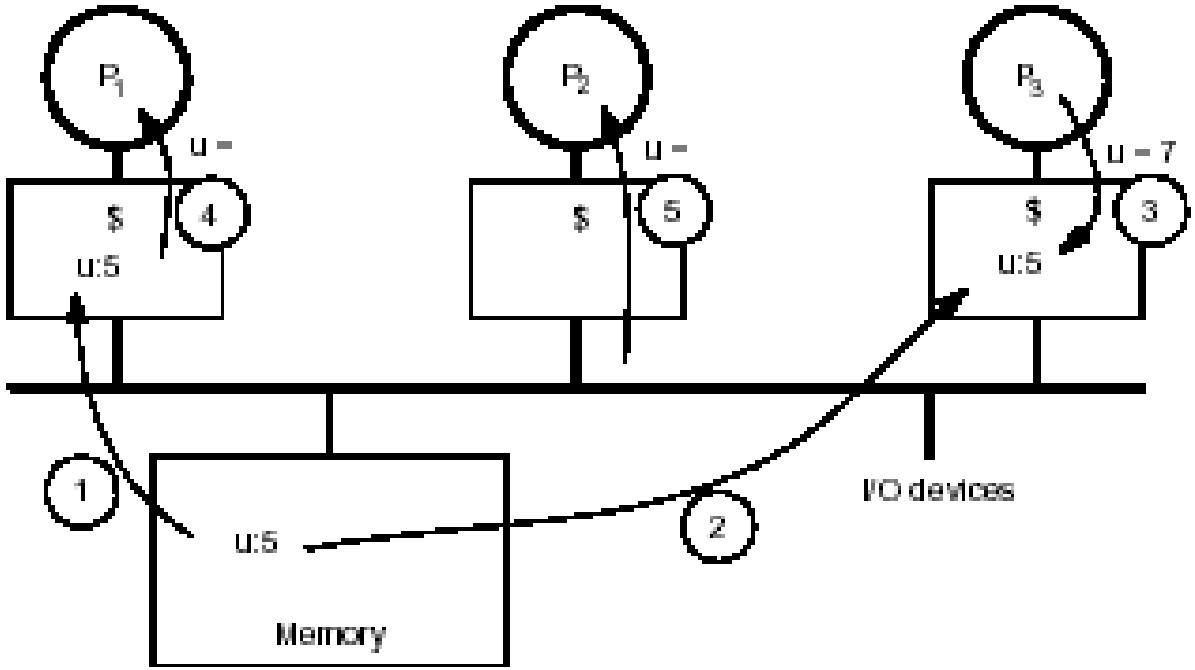
Multi-core vs SMP?

- SMP has cache coherency
- Some Multi-core dose not have it always

SMP Issue

- Cache Coherency Protocol
- Memory Consistency Model
- Synchronization

Cache Coherent Problem



Solution to the Cache Coherence Problem

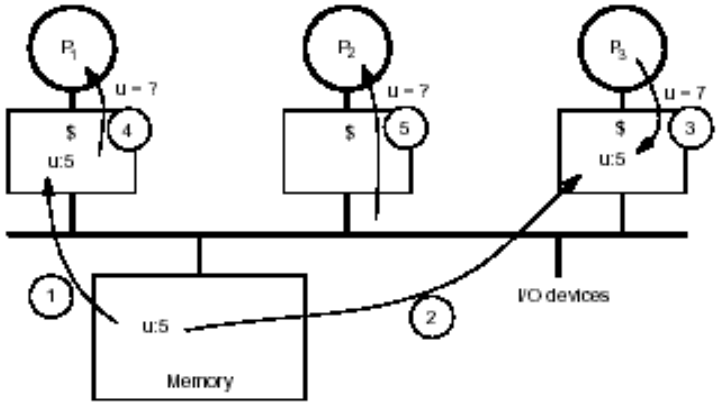
- **Software** 적인 방법
  - Shared writable data are non-cacheable
  - Writable data exists in one cache : Centralized global table
- **Hardware** 적인 방법
  - Shared caches vs Snoopy schemes vs. Directory schemes
  - Monitor possible write operation : Snoopy cache controller

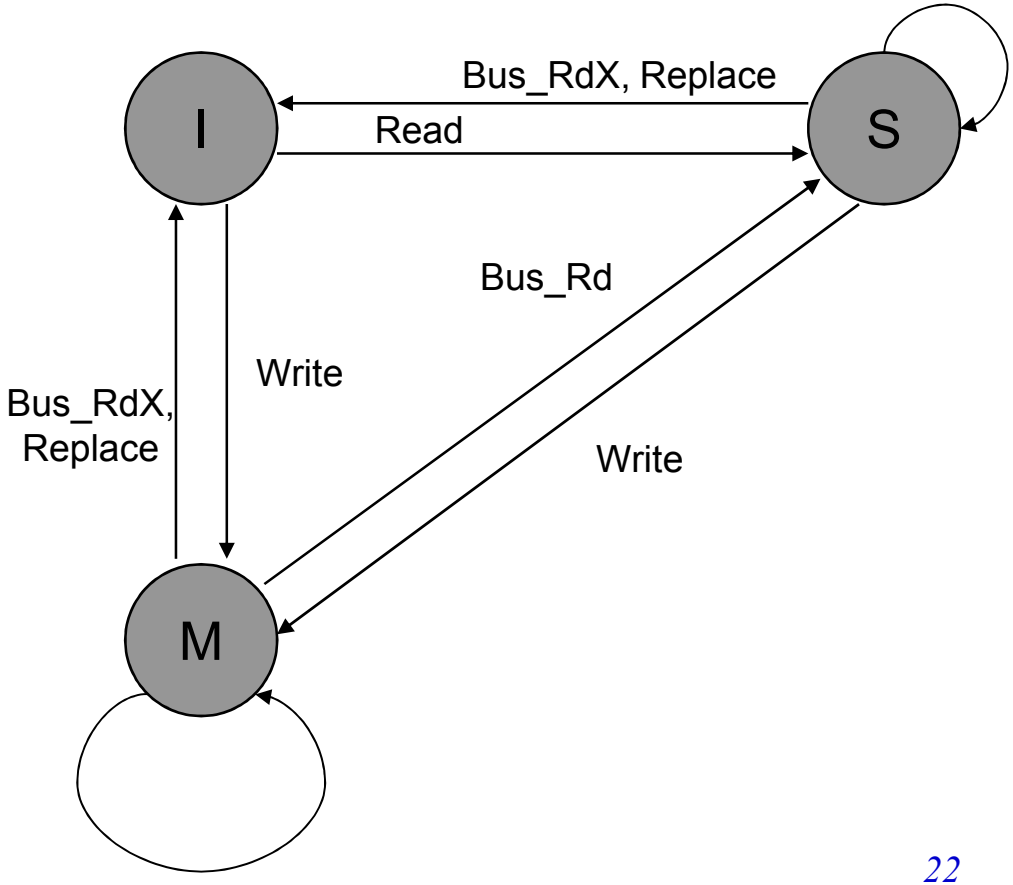
Bus Based Snooping Protocol

- MSI Protocol
- MESI Protocol
- MOESI Protocol

MSI Protocol

- States
  - **Invalid / Not Present**
  - **Shared (readable)**
  - **Modified (read/write)**





Memory Consistency Model

Initially X = 2

P1	P2
…..	…..
r0=Read(X)	r1=Read(x)
r0=r0+1	r1=r1+1
Write(r0,X)	Write(r1,X)
…..	……

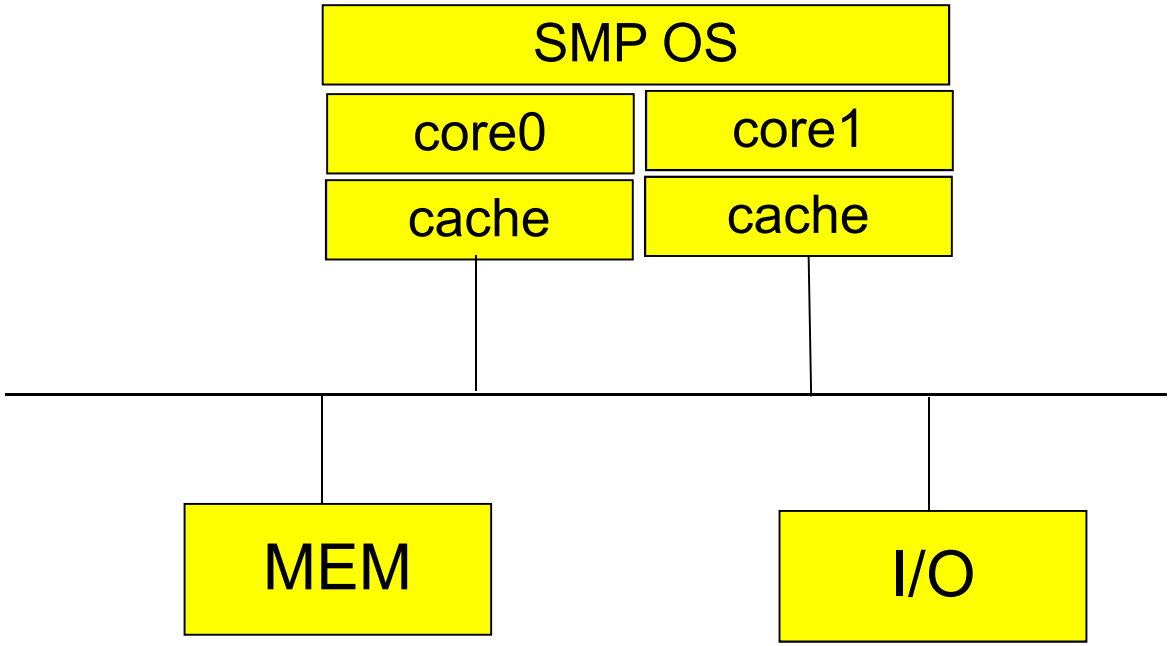
Possible execution sequences:

P1:r0=Read(X)
P2:r1=Read(X)
P1:r0=r0+1
P1:Write(r0,X)
P2:r1=r1+1
P2:Write(r1,X)
x=3

P2:r1=Read(X)
P2:r1=r1+1
P2:Write(r1,X)
P1:r0=Read(X)
P1:r0=r0+1
P1:Write(r0,X)
x=4

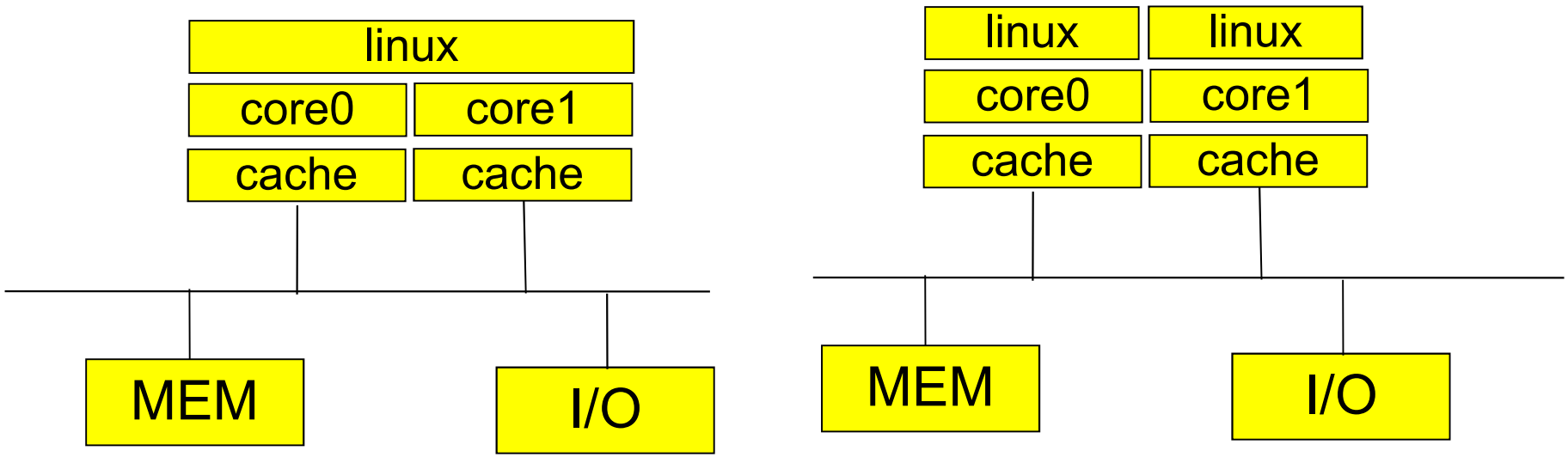
SMP OS

- **OS on Homogenous Symmetric Multi-Processor**
- 하나의 **Single Instance OS**가 모든 **core**에서 동작
- **OS**의 **code/data**를 모든 **core**가 공유
  - Core별로 관리하는 data가 존재



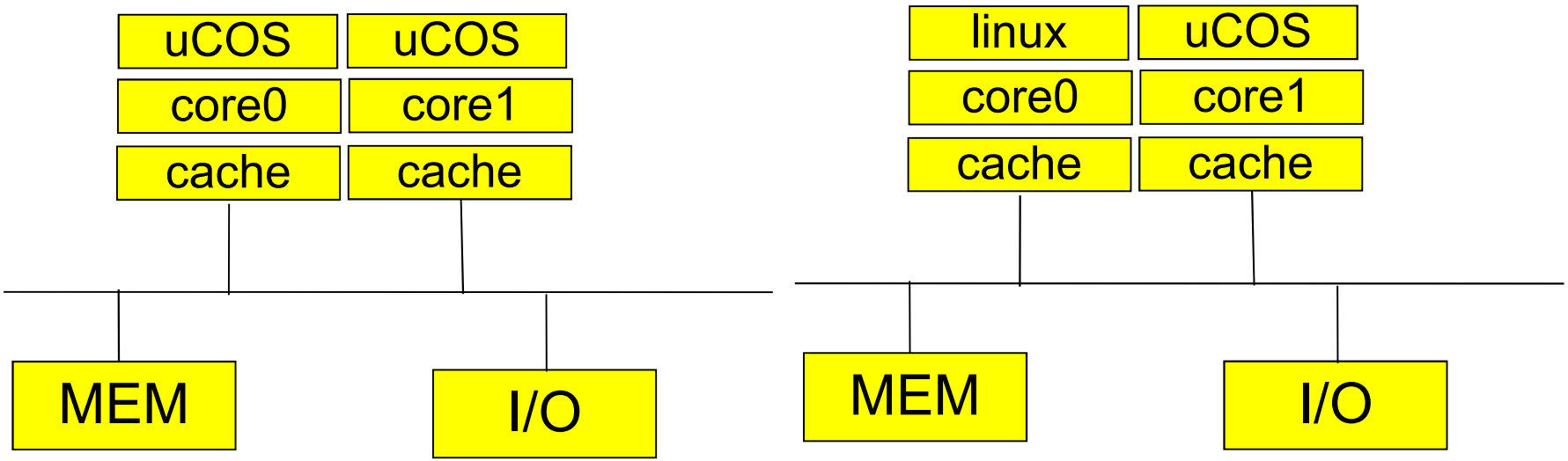
SMP OS vs AMP OS

- **SMP OS**
  - SMP system에 하나의 OS Instance만 존재
    - OS 구현이 힘듦, Application 재사용, 한번 설계 시 확장 용의, System Utilization이 높음
- **AMP OS**
  - SMP system에 여러 OS Instance 존재
    - OS 재 사용, 전담 Task로 성능 향상, Application은 통신 복잡, re-design overhead



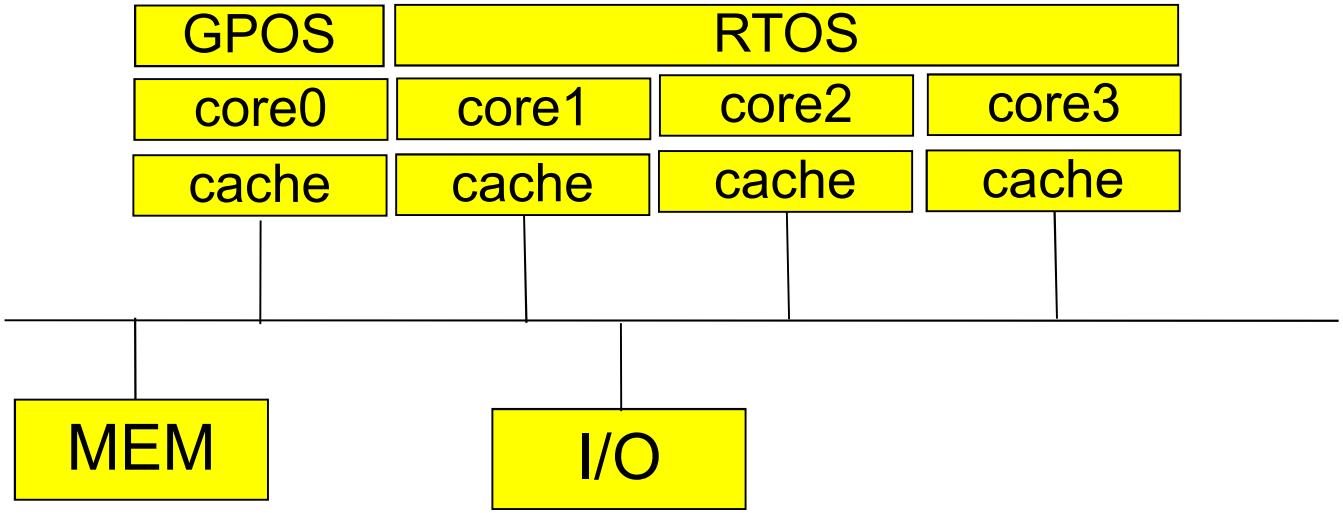
Homogeneous AMP OS vs Heterogeneous AMP OS

- **Homogeneous AMP OS**
  - 같은 종류의 OS가 여러 Instance 존재
- **Heterogeneous AMP OS**
  - 다른 종류의 OS가 여러 Instance 존재
    - 다양한 Middle ware / Application 동작 가능



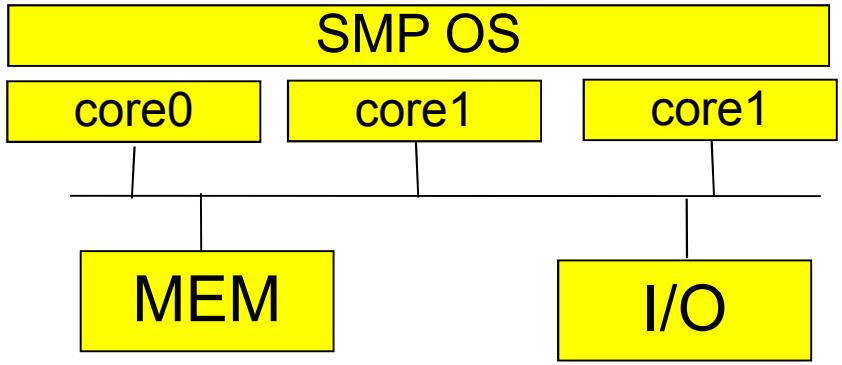
Hybrid System Configuration

- 높은 응답성을 위해서 **Task** 전담 **OS**를 분리
- 여러 **Middleware** 활용을 위해 여러 **OS**를 이용
- **SMP OS**와 **AMP OS** 특징을 살림

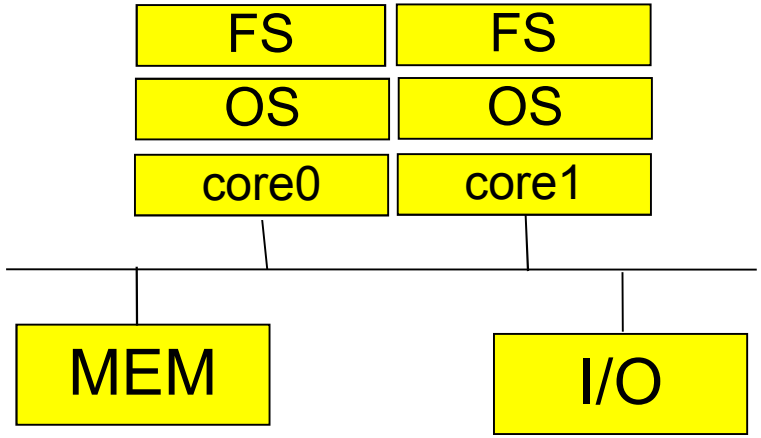


MP OS 구성 분류

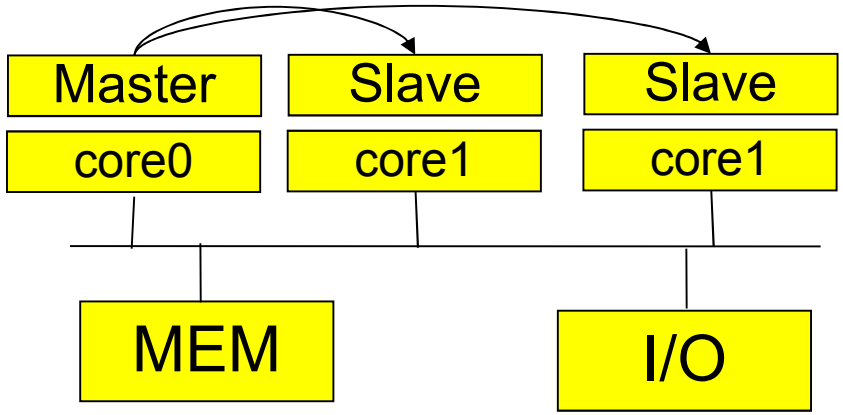
- **Separate Supervisor (AMP OS)**
  - Core 마다 각자 OS가 올라감
  - 좋은 fault tolerance, 나쁜 concurrency
  - 각자 I/O Device나 file system을 전담
- **Master / Slave (AMP OS)**
  - Master는 상태 감시 및 job 할당
  - Slave는 스케줄 가능한 resource pool
  - Master가 bottleneck, 나쁜 fault tolerance
- **Symmetric (= SMP OS)**
  - 모든 processor가 동등, 하나의 OS Instance



Symmetric



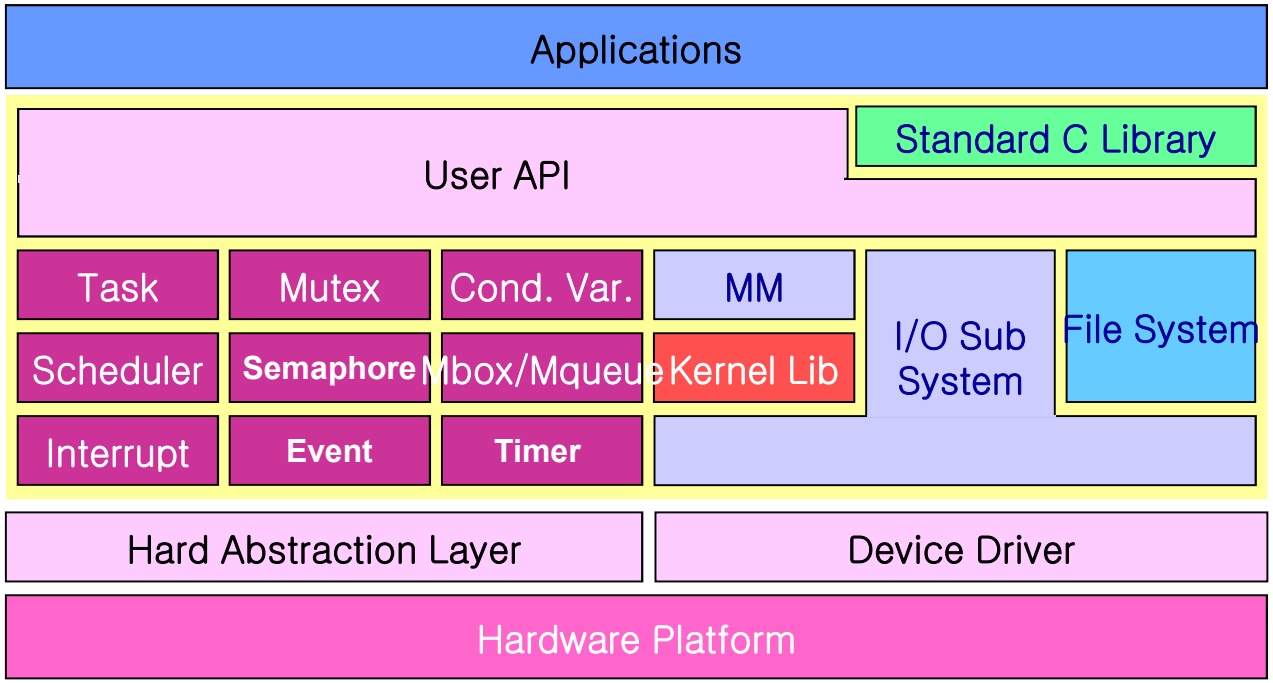
Separate Supervisor



Master / Slave

Single core OS vs SMP OS

- **SMP OS**는 **Single core OS**와 기능적 측면에서 크게 틀리지 않음
- **multi-core**라는 이유 구현 난위도가 다소 높음



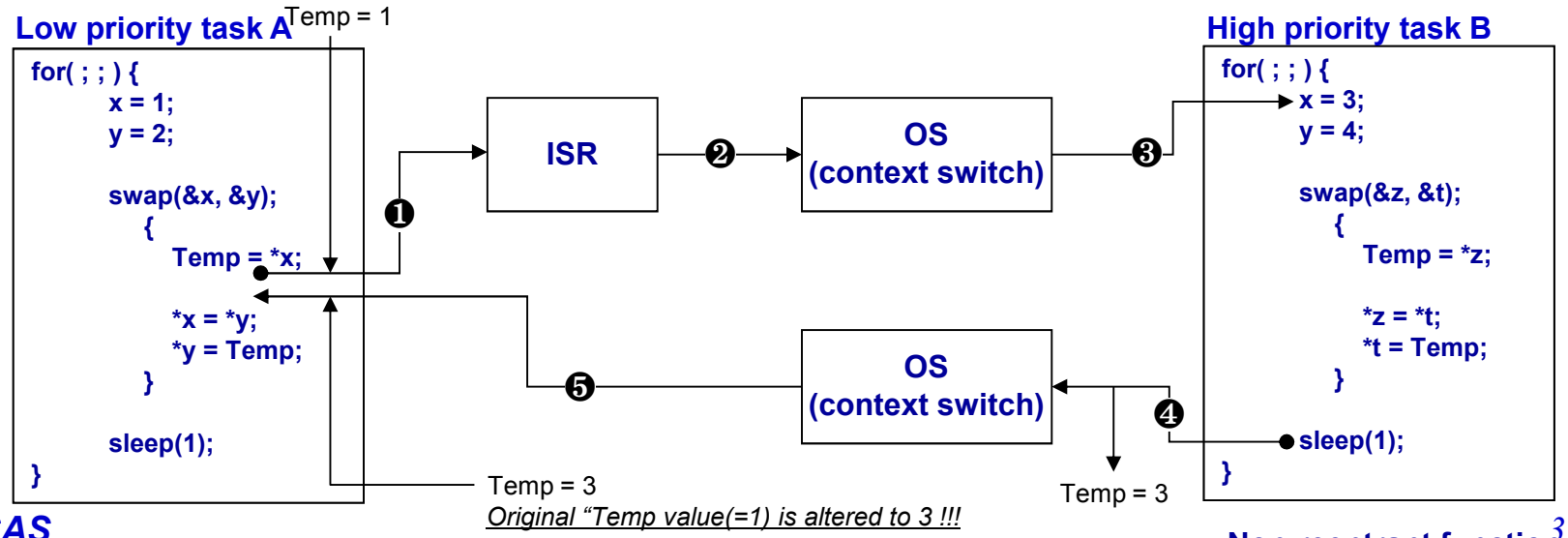
SMP OS Functions

- **Preemptive Kernel (concurrency)**
- **Scheduling**
  - Real-time scheduling on Multi-processor
    - pFair, PD+, PD^2
  - Cache affinity scheduling
    - FP (Fixed Processor), LP (Last Processor), MI (Minimum Intervening), LMI (Limited Minimum Intervening)
- **Locking** 매커니즘
  - Spin Lock
  - Lock Granularity
    - Coarse grained lock
    - Fine grained lock
- **Processor local data storage**
- **Inter Processor Interrupt**
- **Load Balancing**

Preemptive Kernel

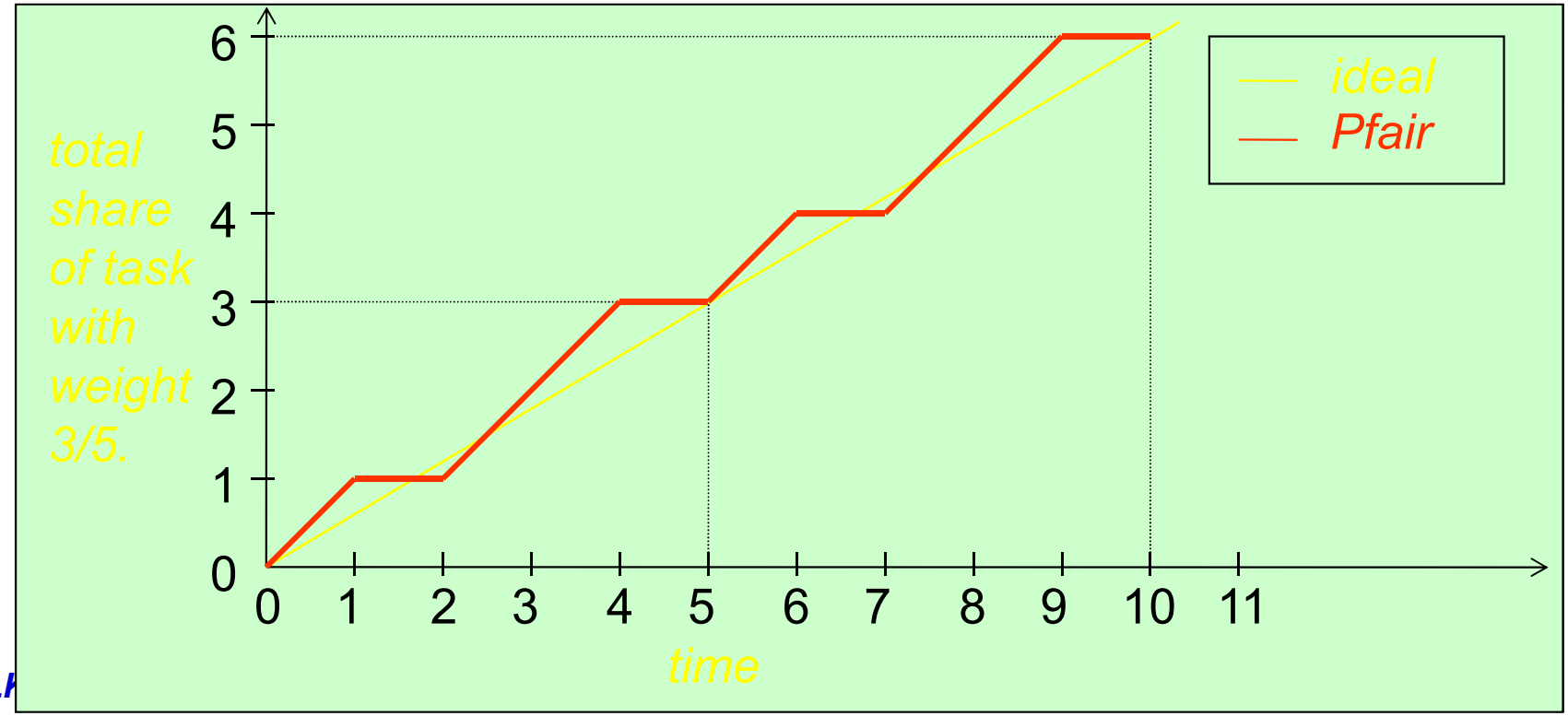
- **SMP OS가 되기 위해서는 preemptive kernel이여야 함**
  - 여러 core가 동시 kernel에 진입할 수 있어야 하기 때문
- **Preemptive kernel은 reentrancy가 전제 임**

```
int Temp;
void swap(int *x, int *y)
{
        Temp        = *x;
        *x          = *y;
        *y          = Temp;
}
```



Real-time Scheduling

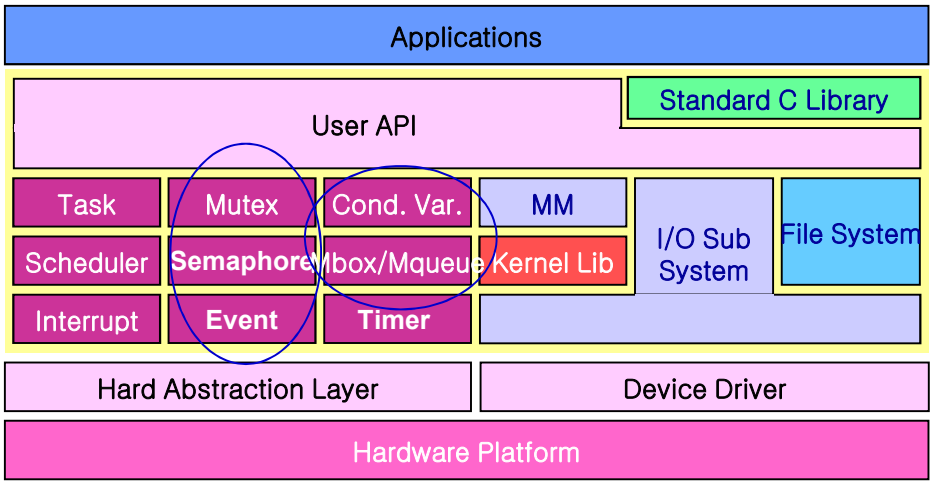
- **Periodic Task system**에 적합 **(static scheduling** 알고리즘 임**)**
- **Idle**한 **weight**에 대한 **utilization**과 **utilization lag(**차이**)**를 최소화하기 위해 **task**를 **sub task**로 나누고 **(Tie-Breaking)** 각 **sub task**를 단위로 **scheduling** 수행
- **Optimal pFair : PF, PD, PD^2**



Locking 매커니즘

❑ Lock Primitive

- ❖ Single processor에서는 interrupt disable로 critical section을 보장하는 것이 가능했으나 SMP에서는 가능하지 않음
- ❖ SMP에서는 spin lock을 사용하여야 함
  - ➢ Local interrupt를 disable하는 것만으로는 critical section을 보장할 수 없음
  - ➢ Spin lock은 H/W에서 지원하는 atomic bus locked operation 또는 test-and-set 명령을 사용하여 구현
    - ✓ *X86:XCHG or lock prefix, ARM: ldrex/strex*
  - ➢ Uni-processor에서 구현한 synchronization primitive 중에 interrupt disable을 사용하여 구현한 것들은 모두 spin lock으로 변경해야 함.



Spin Lock

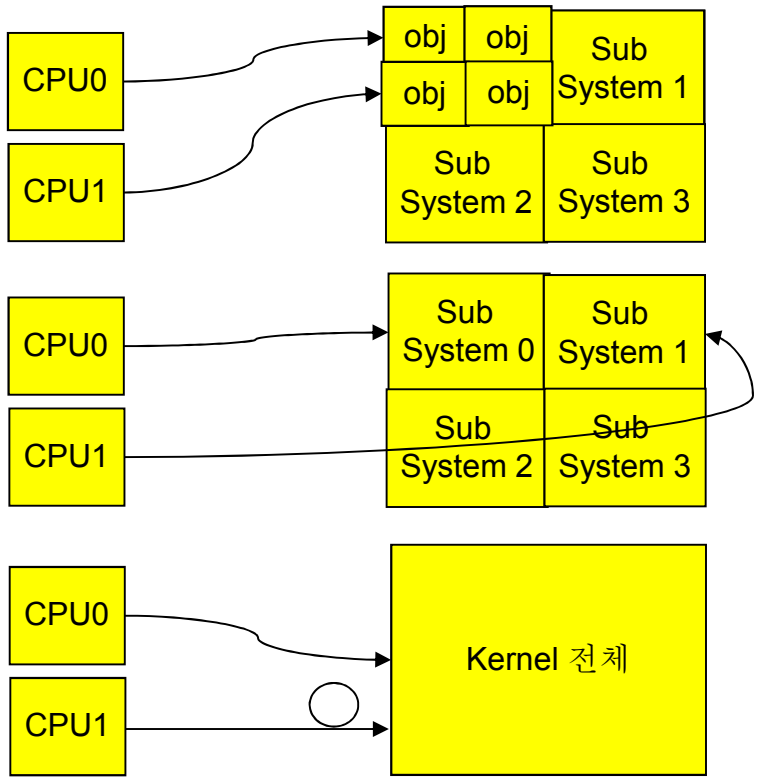
```
27 #define __down_op(ptr,fail)                     \
28         ({                                  \
29         __asm__ __volatile__(                   \
30         "@ down_op\n"                           \
31 "1:     ldrex   lr, [%0]\n"                     \
32 "       sub     lr, lr, %1\n"                   \
33 "       strex   ip, lr, [%0]\n"                 \
34 "       teq     ip, #0\n"                       \
35 "       bne     1b\n"                           \
36 "       teq     lr, #0\n"                       \
37 "       movmi   ip, %0\n"                       \
38 "       blmi    " #fail                         \
39         :                                   \
40         : "r" (ptr), "I" (1)                    \
41         : "ip", "lr", "cc");                    \
42         smp_mb();                               \
43         })
```

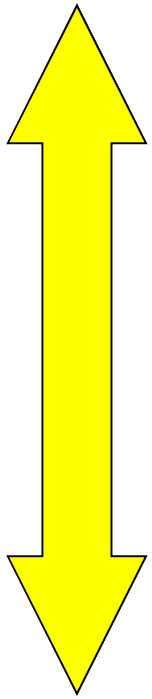
Spin Lock

```
67 #define __up_op(ptr,wake)                       \
68         ({                                      \
69         smp_mb();                               \
70         __asm__ __volatile__(                   \
71         "@ up_op\n"                             \
72 "1:     ldrex   lr, [%0]\n"                     \
73 "       add     lr, lr, %1\n"                   \
74 "       strex   ip, lr, [%0]\n"                 \
75 "       teq     ip, #0\n"                       \
76 "       bne     1b\n"                           \
77 "       cmp     lr, #0\n"                       \
78 "       movle   ip, %0\n"                       \
79 "       blle    " #wake                         \
80         :                                       \
81         : "r" (ptr), "I" (1)                    \
82         : "ip", "lr", "cc");                    \
83         })
```

Lock Granularity

- **Lock Granularity**가 세밀할수록 **SMP Scalability**를 높일 수 있음
- 하지만 너무 **Lock Granularity**를 높이면 **lock** 자체의 **overhead**가 발생
- **SMP Scalability: CPU** 수를 늘리면 늘릴수록 그만큼 성능도 선형증가 되어야 함. **(but amdal's low**에 의해 한계는 발생)





Lock Granularity가 높음
동시에 수행되는 구간 많음
Lock overhead 많음
Lock contention 복잡

Processor local data storage

- 하나의 **core**만 유지하면 되는 전역 자료들에 대해서 **spin lock** 없이 접근할 수 있게 각 **core**에 각각 할당된 공간 **(ex : per_cpu)**
- 예를 들면 **runqueue, irq nest, current task**등이 들어갈 수 있음

```
#define DEFINE_PER_CPU(type, name) \
__attribute__((__section__(".data.percpu"))) __typeof__(type) per_cpu__##name
```

Inter Processor Interrupt

- **Shared memory**를 사용하지 않고 빠르게 **processor**간 통신이 필요한 경우 사용
  - Task terminate from other core.
  - Rescheduling from other core.
  - Etc…
- **SMP H/W**에서 **Virtual Interrupt** 또는 **IPI**로 지원하고 있음

Load Balancing

- **Idle processor 없이 system utilization을 높임**
- **Load balance policy**
  - Balancing Point
  - Migration policy
- **Load Balance 고려사항**
  - Fairness
    - 모든 task는 모든 core에서 골고루 수행되어야 함
  - Efficiency
    - Idle 상태인 core가 없이 모든 core가 task를 수행할 수 있어야 함
  - Scalability
    - Core 개수에 따라 성능도 증가
    - Core 개수에 따른 overhead 최소화
  - Affinity
    - Cache affinity가 적용되어야 함
    - NUMA라면 memory affinity도 고려되어야 함