Fall '20 COSE322-00

System Programming

Practice 05. Network Common Coding Patterns

2020. 11. 5.

Contents

- **Common coding patterns**
  - Memory Allocation in Kernel
  - Memory Caches
  - Caching and Hash Tables
  - Reference Counting and Garbage Collection
  - Function Pointers
  - goto Statements
  - Conditional Directives
  - Compile-Time Optimization for Condition Checks
  - Mutual Exclusions
  - Conversions between Host and Network Order
  - Measuring Times

Memory allocation in Kernel

❖ 커널영역의 메모리 할당을 위해 vmalloc()과 kmalloc() 함수를 사용한다

```
void *kmalloc(size_t size, int flags)
void *vmalloc(unsigned long size)
```

– 차이점:
  - vmalloc(): 단편화 된 페이지들을 모아 연속된 가상 메모리 공간에 매핑하여 할당
  - kmalloc(): 연속된 물리 메모리를 연속된 가상 메모리 공간에 매핑하여 할당

– Flags : Action modifier, Zone modifier, Type
  - Type Flags (일반적인 커널 영역에서의 메모리 할당에는 GFP_KERNEL이 쓰임)

Flag	설명
GFP_ATOMIC	메모리가 있으면 할당, 없으면 NULL을 반환함 (Sleep되면 안 되는 상황)
GFP_KERNEL	Kernel용 메모리 할당이며 sleep될 수 있음
GFP_USER	User영역의 메모리 할당 (커널이나 하드웨어에서 직접 엑세스 할 수 있음)
GFP_DMA	DMA에 쓰기 위한 연속적인 물리 메모리 할당

Slab Caches

- **Memory Caches**
  - 커널 함수는 같은 타입의 메모리를 자주 요청하는 경향이 있다
    - 동일한 구조체의 instance를 자주 할당, 해제하는 경우 해당 구조체를 위한 메모리 영역을 특별히 할당하여 Cache처럼 사용
  - 메모리 캐시 영역 생성/해제
    - Cache 생성, 해제 : `kmem_cache_create, kmem_cache_destroy`
    - Cache에 버퍼 할당, 반환 : `kmem_cache_alloc, kmem_cache_free`
  - `/proc/slab`
  - 대표적인 `cache`
    - `kmalloc-8, 16, 32, 64, 96, 128, 192, 256, 512, …, 8k`
    - `thread_info`

List and Hash Tables

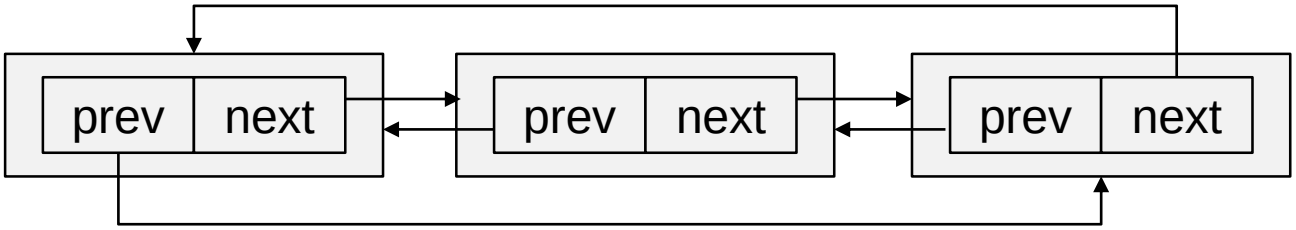
net/ipv4/tcp_ipv4.c

- List와 Hash Table은 커널 전체에서 빈번히 사용되는 자료구조
- List

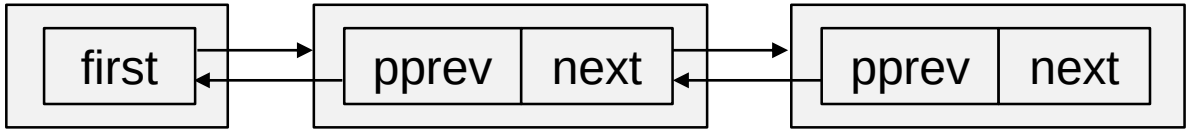
```
185 struct list_head {
186         struct list_head *next, *prev;
187 };
188
189 struct hlist_head {
190         struct hlist_node *first;
191 };
192
193 struct hlist_node {
194         struct hlist_node *next, **pprev;
195 };
```

List and Hash Tables

❖ list_head로 구성한 list (Doubly linked list)



❖ hlist_head와 hlist_node로 구성한 list (해시 테이블에 많이 쓰임)



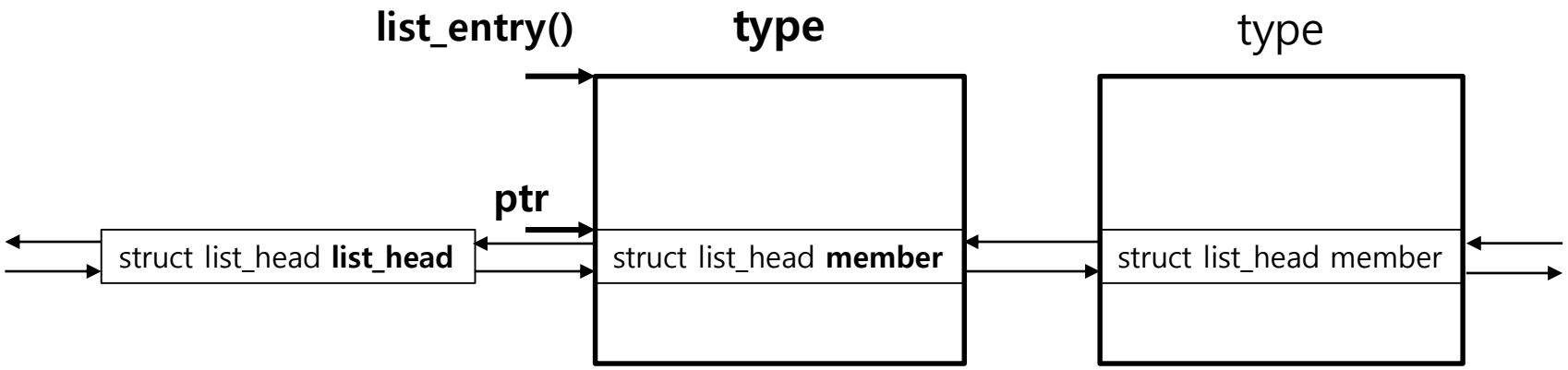
List and Hash Tables

❖ List 자료구조와 관련된 매크로

include/linux/list.h

```
351 #define list_entry(ptr, type, member) \
352         container_of(ptr, type, member)
353

408 #define list_for_each(pos, head) \
409         for (pos = (head)->next; pos != (head); pos = pos->next)
410
```



※ **head** = &list_head

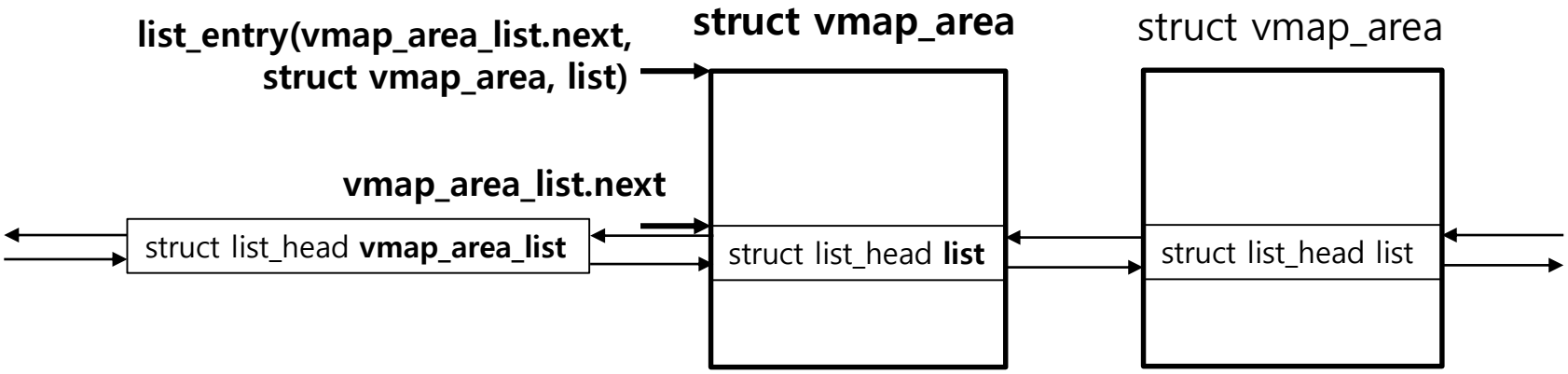
List and Hash Tables

❖ 활용 예시

include/linux/vmalloc.h

```
struct vmap_area {
    unsigned long va_start;
    unsigned long va_end;
    …
    struct list_head list;
    …
};

struct list_head *vmap_area_list;
```



List and Hash Tables

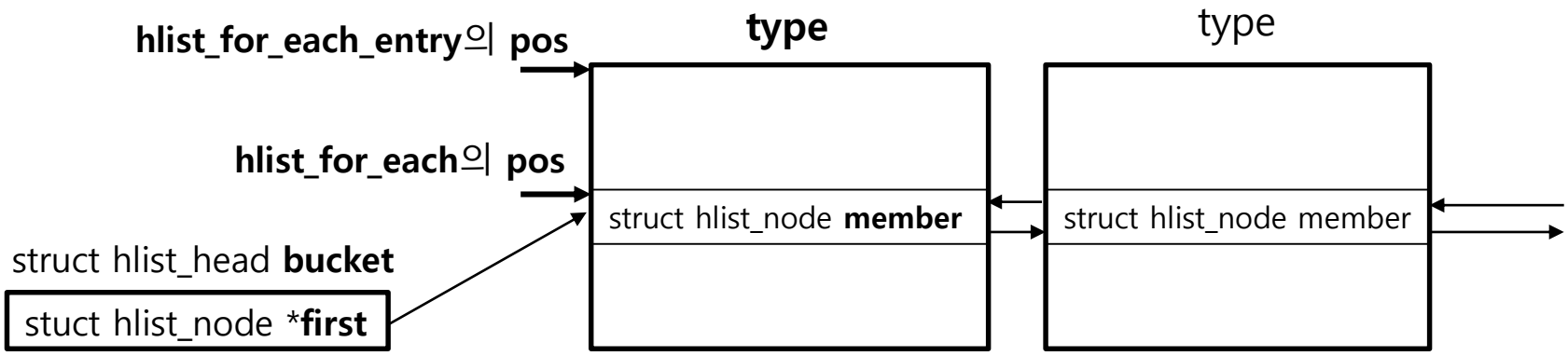
❖ HList 자료구조와 관련된 매크로

include/linux/list.h

```
688 #define hlist_entry(ptr, type, member) container_of(ptr,type,member)

690 #define hlist_for_each(pos, head) \
691         for (pos = (head)->first; pos ; pos = pos->next)

708 #define hlist_for_each_entry(pos, head, member)                          \
709     for (pos = hlist_entry_safe((head)->first, typeof(*(pos)), member);\
710          pos;                                                          \
711          pos = hlist_entry_safe((pos)->member.next, typeof(*(pos)), member))
```



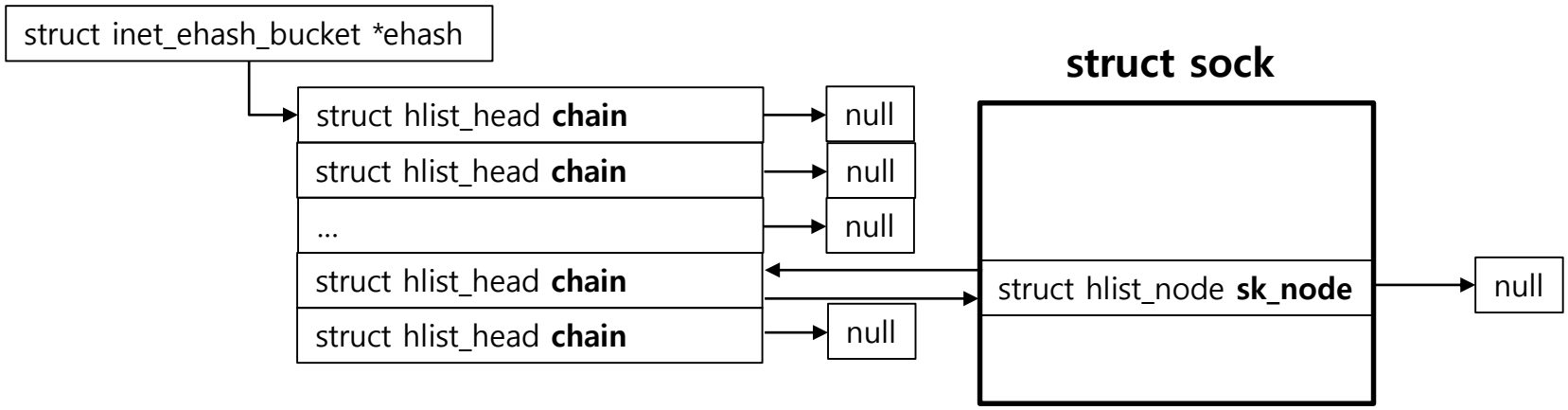
※ **head** = &bucket

List and Hash Tables

❖ 활용 예시

include/net/sock.h
include/net/inet_hashtables.h

```
struct inet_hashinfo {
        struct inet_ehash_bucket        *ehash;
    ...
};
struct inet_ehash_bucket {
        struct hlist_nulls_head chain;
};
struct sock {
        struct hlist_nulls_node sk_node
    ...
}
```



Reference Counts and Garbage Collection

- **특정 구조체를 Free하는 시점**
  - Reference하는 존재가 있다면 Free할 수 없음
  - Garbage collection을 효과적으로 수행하기 위해 대부분의 자료구조는 자체적으로 reference count 변수를 가지고 있음
    - `xxx_hold`
    - `xxx_release`

- **Garbage Collection**
  - Reference Count 조작 혹은 Timer softIRQ 등을 이용해서 각 data마다 자체적으로 GC를 구현

Function Pointers

❖ Function Pointer

```
struct sock {
    …
    void (*sk_state_change)(struct sock *sk);
    void (*sk_data_ready)(struct sock *sk,int bytes);
    …
}
```

- 장점
  - 다양한 기준이나 Object의 역할에 따라 함수호출을 다르게 할 수 있음
- 단점
  - Code readability 저하

goto Statements

❖ goto Statement의 사용

net/ipv4/tcp_input.c

```
int tcp_conn_request(…) {
……
done:
        reqsk_put(req);
        return 0;
drop_and_release:
        dst_release(dst);
drop_and_free:
        reqsk_free(req);
drop:
        tcp_listendrop(sk);
        return 0;
}
```

- free / release나 error 처리 시 용이
- 동일한 코드 반복을 줄일 수 있음. (특정한 상황에서 코드 가독성 상승)

Conditional Directives (#ifdef and family)

❖ 사용하는 이유

- 커널 코드를 설정/디바이스 등에 따라 최적화하기 위함
- 해당 부분은 컴파일 시 처리됨

전처리문	설명
#ifdef #elif, #else #endif	식별자가 정의된 경우 if 1과 동일하고 정의되지 않은 경우 if 0과 동일함
#ifndef #elif, #else #endif	식별자가 정의되지 않은 경우 if 0, 정의된 경우 if 1과 동일함

Conditional Directives (#ifdef and family)

❖ 예시 (1) – 자료구조에서 특정 변수를 포함 또는 제외

```
struct sk_buff {
    …
#ifdef CONFIG_NETFILTER_DEBUG
    unsigned int nf_debug;
#endif
    …
}
```

– Netfilter Debugging tool을 사용하는 경우 sk_buff 구조체에 nf_debug라는 변수를 필요로 함

– 컴파일 시 해당 feature가 사용되는 경우 #ifdef ~ #endif 영역을 함께 컴파일함

Conditional Directives (#ifdef and family)

❖ 예시 (2) – 함수에서 코드의 일부를 포함 또는 제외

```
int ip_route_input(…)
{   …
    if(rth->fl.fl4_dst == daadr &&
       rth->fl.fl4_src == saddr &&
       rth->fl.iif == iif &&
       rth->fl.oif == 0 &&
#ifndef CONFIG_IP_ROUTE_FWMARK
       rth->fl.fl4_fwmark == skb->nfmark &&
#endif
       rth->fl.fl4_tos == tos){
            …
       }
    …
}
```

– “IP: use netfilter MARK value as routing key”를 지원하는 시스템에서 컴파일 할 때 체크하는 코드

Conditional Directives (#ifdef and family)

❖ 예시 (3) – 조건에 따라 변수의 prototype을 다르게 선언할 때

```
#ifdef CONFIG_IP_MULTIPLE_TABLES
struct fib_table * fib_hash_init(int id)
#else
struct fib_table * __init fib_hash_init (int id)
{
    …
}
```

- 커널이 Policy Routing을 지원하지 않으면 __init tag를 prototype에 붙여 선언
  - `__init` 매크로
    - 해당 함수나 변수가 운영체제의 초기화 과정에만 사용된다는 것을 의미함 (부팅 후 제거됨 == 부팅 이후 호출할 수 없음)

Conditional Directives (#ifdef and family)

❖ 예시 (4) – 함수를 다르게 구현할 때

```
#ifndef CONFIG_IP_MULTIPLE_TABLES
...
static inline struct fib_table *fib_get_table(int id)
{
    if (id != RT_TABLE_LOCAL)
            return ip_fib_main_table;
    return ip_fib_local_table
}
...
#else
...
static inline struct fib_table *fib_get_table(int id)
{
    if (id == 0)
            id = RT_TABLE_MAIN;
    return fib_tables[id];
}
```

Compile-Time Optimization for Condition Checks

- **Variable과 External value를 비교하는 경우**
  - 결과를 예측할 수 있는 경우가 매우 많음
    - likely : true를 리턴할 것 같은 비교에 대한 wrapping macro
    - unlikely : false를 리턴할 것 같은 비교에 대한 wrapping macro
  - 컴파일러가 **최적화**를 할 수 있는 여지를 제공함
    - 자주 호출되지 않는 코드를 함수 뒷부분에 배치하여 memory cache나 branch prediction cache에 유리한 영향을 줄 수 있음.

```
err = do_something(x,y,z);
if (err)
        handle_error(err);
```

```
err = do_something(x,y,z);
if (unlikely(err))
        handle_error(err);
```

Mutual Exclusions

- **Semaphore / Mutex**
  - 리소스를 사용할 수 없으면 Sleep 가능
  - 대기 순서를 발행하여, 기아 상태 해결
  - Semaphore는 동시에 여러 개의 리소스를 사용할 수 있을 때 사용
  - Mutex는 한 개의 리소스만 사용할 수 있을 때 사용
  - **Interrupt Context에서 사용 불가**

- **Spin Lock**
  - 해당 자원을 단 하나의 쓰레드만 Hold 가능
  - 대기하는 쓰레드의 looping waste 발생
  - 멀티프로세서 또는 Hold하는 시간이 짧다고 예상되는 경우에 사용
  - **CS(Critical Section) 내에서 sleep이 되면 안 됨**

Mutual Exclusions

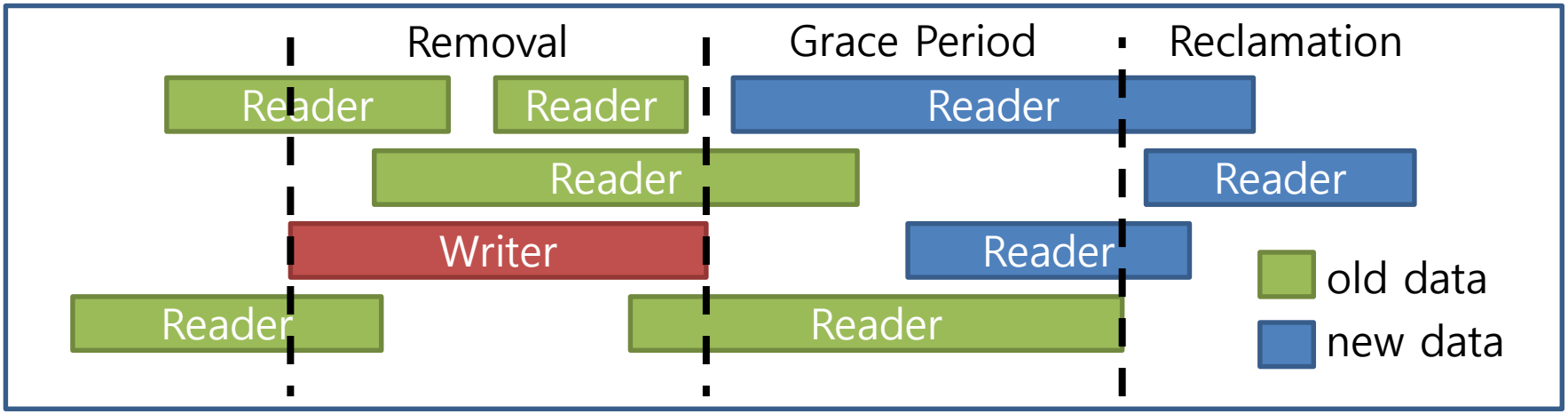
- **Read-Write Spin Lock / Semaphore**
  - 복수의 Readers가 자원을 hold할 수 있음
  - Writer는 단 하나만 접근 가능(물론 Reader도 접근 불가)

- **RCU(Read-Copy-Update)**
  - 커널 잠금을 제거하면서 동기화를 유지하려는 시도임
  - 매우 빈번한 Read와 간혹 발생하는 Write가 존재할 때 사용
  - 2개 이상의 Writer가 접근 하는 것은 동기화 보장을 하지 않음 (다른 lock을 추가로 사용)
  - 조건에 따라 Preemption이 가능한 동기화 기법 (설정에 따라 read-side CS에서 Sleep도 가능하다)

Mutual Exclusions

❖ RCU(Read-Copy-Update) 동작 방식

- Removal 구간:
  - Reader가 CS 영역의 데이터를 접근하는 도중 Writer가 접근해서 해당 데이터를 수정하면, 해당 data를 읽고 복사한 후 복사한 데이터를 수정 (Read-Copy-Update)
- Grace Period 구간:
  - Writer의 사용이 끝난 이후 접근하는 Reader는 Update 완료된 new data에 접근 (old data와 new data가 공존)
- Reclamation 구간:
  - old data에 접근하는 Reader가 모두 종료되면 old data를 폐기



Conversions between Host and Network Order

- **메모리에 자료가 저장되는 방식: Little Endian, Big Endian**
  - 1바이트 이상의 자료가 메모리에 저장되는 방식 (프로세서에 따라 결정됨)
  - Little Endian : LSB를 메모리의 낮은 주소에 위치시킴
  - Big Endian : LSB를 메모리의 높은 주소에 위치시킴

- **네트워크 규칙 : Big Endian Model**
  - 서로 주고받는 프로세서가 어떤 방식을 사용하는 지 알 수 없음
  - 1바이트 이상의 자료를 읽거나 저장, 비교할 때 메모리 저장방식을 고려하기 위한 매크로가 존재함

Little Endian	Big Endian
Intel x86	IBM
AMD	SPARC
DEC	Motorola

Macro	Meaning (short; 2bytes, long; 4bytres)
htons	Host-to-network byte order (short)
htonl	Host-to-network byte order (long)
ntohs	Network-to-host byte order (short)
ntohl	Network-to-host byte order (long)

Measuring Times

- **커널영역에서의 시간 측정은 ticks를 이용한다**
  - tick : 연이은 두 타이머 interrupt의 expire 간격
  - 타이머는 정기적으로 일초에 Hz번의 expire를 수행함
    - (예) Linux on i386 : 1초에 1000번 expiring → 1 tick = 1ms
  - 타이머가 expire할 때마다 커널영역의 전역변수인 jiffies가 1씩 증가함
  - (예시) do_something을 1 tick 이내로만 수행하고자 하는 코드

```
unsigned long start_time = jiffies;
int job_done = 0;
do {
    do_something(&job_done);
    If (job_done)
        return;
while (jiffies - start_time < 1);
```