14. 고급 주제 및 최신 기술 연동

14.1 비동기 네트워크 라이브러리 (libevent, libuv) 사용법

🎯 학습 목표

- `select`, `poll`, `epoll` 등의 저수준 시스템콜 없이
  고성능 네트워크 I/O 비동기 처리를 구현하고자 할 때,
  C 기반으로 널리 사용하는 라이브러리 2가지는 다음과 같다.

라이브러리	특징
`libevent`	`epoll`, `kqueue` 등을 자동 감지하여 **이벤트 기반 비동기 처리** 제공
`libuv`	Node.js의 런타임 기반이며 **크로스플랫폼 지원**, 타이머/파일/IPC 지원

✅ 1. libevent 기본 사용법

🔹 설치

```
sudo apt install libevent-dev
```

🔹 주요 API 구조

```
struct event_base* base = event_base_new();        // 이벤트 루프
struct event* ev = event_new(base, fd, flags, callback, arg);
event_add(ev, timeout);                             // 이벤트 등록
event_base_dispatch(base);                          // 루프 시작
```

✅ TCP Echo 서버 예제 (libevent)

```
#include <event2/event.h>
#include <event2/bufferevent.h>
#include <event2/listener.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 12345

void echo_read_cb(struct bufferevent* bev, void* ctx) {
    char buffer[1024];
    int n = bufferevent_read(bev, buffer, sizeof(buffer));
    bufferevent_write(bev, buffer, n); // Echo back
}

```

```c
void echo_event_cb(struct bufferevent* bev, short events, void* ctx) {
    if (events & BEV_EVENT_EOF || events & BEV_EVENT_ERROR)
        bufferevent_free(bev);
}

void accept_conn_cb(struct evconnlistener* listener, evutil_socket_t fd,
                    struct sockaddr* addr, int socklen, void* ctx) {
    struct event_base* base = ctx;
    struct bufferevent* bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);
    bufferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb, NULL);
    bufferevent_enable(bev, EV_READ | EV_WRITE);
}

int main() {
    struct event_base* base;
    struct evconnlistener* listener;
    struct sockaddr_in sin;

    base = event_base_new();

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(PORT);

    listener = evconnlistener_new_bind(base, accept_conn_cb, base,
        LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE, -1,
        (struct sockaddr*)&sin, sizeof(sin));

    printf("libevent echo server running on port %d\n", PORT);
    event_base_dispatch(base);
    return 0;
}
```

컴파일

```
gcc -o echo_libevent echo_libevent.c -levent
```

✅ 2. libuv 기본 사용법

설치

```
sudo apt install libuv1-dev
```

핵심 구조

```c
uv_loop_t* loop = uv_default_loop();  // 이벤트 루프
uv_tcp_t server;                      // TCP 서버 객체
uv_tcp_init(loop, &server);
uv_listen((uv_stream_t*)&server, backlog, on_new_connection);
uv_run(loop, UV_RUN_DEFAULT);
```

✅ TCP Echo 서버 예제 (libuv)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <uv.h>

#define PORT 12345

uv_loop_t* loop;

void alloc_buffer(uv_handle_t* handle, size_t suggested_size, uv_buf_t* buf) {
    buf->base = malloc(suggested_size);
    buf->len = suggested_size;
}

void echo_write(uv_write_t* req, int status) {
    free(req->data);
    free(req);
}

void echo_read(uv_stream_t* client, ssize_t nread, const uv_buf_t* buf) {
    if (nread > 0) {
        uv_write_t* req = malloc(sizeof(uv_write_t));
        uv_buf_t wrbuf = uv_buf_init(buf->base, nread);
        req->data = buf->base;
        uv_write(req, client, &wrbuf, 1, echo_write);
        return;
    }
    if (nread < 0) uv_close((uv_handle_t*)client, NULL);
    free(buf->base);
}

void on_new_connection(uv_stream_t* server, int status) {
    uv_tcp_t* client = malloc(sizeof(uv_tcp_t));
    uv_tcp_init(loop, client);
    uv_accept(server, (uv_stream_t*)client);
    uv_read_start((uv_stream_t*)client, alloc_buffer, echo_read);
}

int main() {
    loop = uv_default_loop();
    uv_tcp_t server;
    uv_tcp_init(loop, &server);

    struct sockaddr_in addr;
    uv_ip4_addr("0.0.0.0", PORT, &addr);
    uv_tcp_bind(&server, (const struct sockaddr*)&addr, 0);

    uv_listen((uv_stream_t*)&server, 128, on_new_connection);
    printf("libuv echo server running on port %d\n", PORT);
    uv_run(loop, UV_RUN_DEFAULT);
```

```
    return 0;
}
```

🔹 컴파일

```
gcc -o echo_libuv echo_libuv.c -luv
```

✅ 3. libevent vs libuv 비교

항목	libevent	libuv
대상	주로 네트워크 서버	네트워크 + 타이머/파일/IPC 등 범용
추상화 수준	중간 수준 (buffer, listener)	더 높은 수준 (핸들 단위 추상화)
성능	고성능 (`epoll`, `kqueue` 기반)	고성능 + 더 많은 기능 포함
코드 난이도	비교적 직관적	콜백 체인 많아 약간 복잡
Node.js 기반	❌	✅

✅ 4. 실전 활용 팁

- `select` / `epoll`로 직접 구현할 시간과 비용을 아끼고 싶다면 적극 추천
- 서버 성능 테스트, 게임 서버, 이벤트 중심 API 서버에서 많이 사용됨
- 멀티 플랫폼 구현 시 `libuv` 추천
- 단순 TCP 이벤트 핸들링이면 `libevent`가 더 간단

14.2 C와 MQTT 연동 (mosquitto)

🎯 학습 목표

MQTT(Message Queuing Telemetry Transport)는 **경량 메시지 브로커 프로토콜**로,
IoT, 센서, 실시간 제어 등 **리소스가 제한된 환경**에서 이상적인 메시징 시스템이다.

이 절에서는 **mosquitto C 라이브러리**를 이용해 MQTT 클라이언트를 구현하는 방법을 학습한다.

✅ 1. 기본 개념 정리

항목	설명
Broker	메시지를 중계하는 서버 (예: Mosquitto)
Publisher	특정 **topic**에 메시지를 전송하는 클라이언트
Subscriber	특정 **topic**을 구독하고 메시지를 수신하는 클라이언트

항목	설명
Topic	메시지를 분류하는 경로 (예: `/sensor/temp`)
QoS (0/1/2)	전송 보장 수준 (최대 1회, 최소 1회, 정확히 1회)

✅ 2. 개발 환경 구성

🔹 Mosquitto 브로커 설치

```
sudo apt install mosquitto mosquitto-clients
sudo systemctl start mosquitto
```

🔹 C 라이브러리 설치

```
sudo apt install libmosquitto-dev
```

✅ 3. C Publisher 예제 (`mqtt_pub.c`)

```
#include <mosquitto.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main() {
    mosquitto_lib_init();

    struct mosquitto* mosq = mosquitto_new("pub-client", true, NULL);
    if (!mosq) {
        perror("mosquitto_new");
        return 1;
    }

    mosquitto_connect(mosq, "localhost", 1883, 60);

    for (int i = 0; i < 5; i++) {
        char payload[64];
        sprintf(payload, "Hello MQTT %d", i);
        mosquitto_publish(mosq, NULL, "test/topic", strlen(payload), payload, 0,
false);
        printf("Published: %s\n", payload);
        sleep(1);
    }

    mosquitto_disconnect(mosq);
    mosquitto_destroy(mosq);
    mosquitto_lib_cleanup();
```

```
    return 0;
}
```

✅ 4. C Subscriber 예제 (`mqtt_sub.c`)

```c
#include <mosquitto.h>
#include <stdio.h>
#include <stdlib.h>

void on_message(struct mosquitto* mosq, void* userdata,
                const struct mosquitto_message* message) {
    printf("Received on %s: %s\n", message->topic, (char*)message->payload);
}

int main() {
    mosquitto_lib_init();

    struct mosquitto* mosq = mosquitto_new("sub-client", true, NULL);
    if (!mosq) {
        perror("mosquitto_new");
        return 1;
    }

    mosquitto_message_callback_set(mosq, on_message);

    mosquitto_connect(mosq, "localhost", 1883, 60);
    mosquitto_subscribe(mosq, NULL, "test/topic", 0);

    mosquitto_loop_start(mosq);
    printf("Subscribed to 'test/topic'\n");
    getchar();  // 엔터 치면 종료
    mosquitto_loop_stop(mosq, true);

    mosquitto_disconnect(mosq);
    mosquitto_destroy(mosq);
    mosquitto_lib_cleanup();

    return 0;
}
```

✅ 5. 컴파일 및 실행

```
gcc -o mqtt_pub mqtt_pub.c -lmosquitto
gcc -o mqtt_sub mqtt_sub.c -lmosquitto
```

실행 순서

```
./mqtt_sub       # 수신 대기
./mqtt_pub       # 메시지 전송
```

✅ 6. 테스트 확인

```
# Mosquitto CLI로도 확인 가능
mosquitto_sub -t test/topic
mosquitto_pub -t test/topic -m "From terminal"
```

✅ 7. 확장 포인트

- QoS 수준 조정 (`mosquitto_publish`의 qos 파라미터)
- 인증 설정 (`mosquitto_username_pw_set`)
- TLS 보안 적용
- 여러 토픽 동시 구독 (`mosquitto_subscribe_multiple`)
- `mosquitto_loop_forever()` → `mosquitto_loop_start()`로 비동기화 가능

14.3 멀티코어 네트워크 처리 모델

🎯 학습 목표

현대의 멀티코어 CPU 환경에서 **단일 스레드 기반 네트워크 서버**는 성능 한계에 도달한다.
이 단원에서는 **여러 코어를 효율적으로 활용하는 네트워크 처리 모델들**을 비교하고,
C 언어로 구현 가능한 구조 설계까지 설명한다.

✅ 1. 왜 멀티코어 모델이 필요한가?

- **단일 스레드 처리 방식**은 병목 발생 (`select`, `epoll` 만 사용하는 경우)
- 서버 부하 증가 시 **CPU 1개만 100%**, 나머지는 Idle
- 멀티 코어 활용이 가능한 구조로 변경하면 **성능이 선형 증가**

✅ 2. 대표적인 멀티코어 처리 모델

모델	설명	장점	단점
`fork` 모델	코어 수만큼 프로세스 생성	커널 스케줄링 최적화	메모리 복사 비용
`pthread` 모델	스레드 풀에서 처리 분산	자원 공유 용이	동기화 복잡

모델	설명	장점	단점
`prefork` 모델	프로세스 미리 생성 후 socket 공유	안정적 + 빠른 응답	socket race 방지 필요
SO_REUSEPORT	커널 레벨에서 socket을 분산 수신	코드 복잡도 낮음	커널 3.9+ 이상
Event-loop + Thread pool	이벤트 수신은 메인스레드, 처리는 워커스레드	구조적 확장성 우수	설계 복잡도 ↑

✅ 3. 핵심 기법: `SO_REUSEPORT`

🔹 설명

- 여러 프로세스/스레드가 **같은 포트를 listen()** 하도록 허용
- 커널이 자동으로 **connection을 분산 처리**

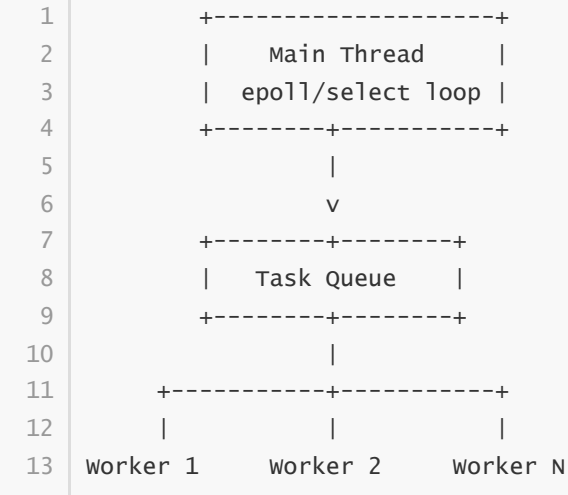
🔹 코드 예시 (다중 프로세스 수신)

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
int opt = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt));
```

🔹 프로세스 예시

```
for (int i = 0; i < num_cores; i++) {
    if (fork() == 0) {
        listen_and_accept(); // 각 프로세스가 동일 포트 listen
        exit(0);
    }
}
```

✅ 4. 구조도 예시: Event-loop + Thread Pool



```
(thread)     (thread)     (thread)
```

✅ 5. C 기반 에코 서버 멀티코어 확장 예시

`fork + SO_REUSEPORT` 기반

```c
int cores = sysconf(_SC_NPROCESSORS_ONLN);
for (int i = 0; i < cores; i++) {
    if (fork() == 0) {
        run_server();  // 동일 포트 listen + epoll
        exit(0);
    }
}
```

→ 각 프로세스는 커널에 의해 자동 분산 처리됨 (부하균형)

✅ 6. 워커 스레드 기반 비동기 처리 예시 흐름

- `epoll_wait()` → 이벤트 발생
- 연결된 소켓 fd를 워커 스레드 풀의 큐로 전달
- 각 스레드는 fd를 읽고, 처리 후 응답

```c
// 메인 스레드
while (1) {
    events = epoll_wait(...);
    for (...) {
        enqueue(fd); // 워커에 전달
    }
}

// 워커 스레드
void* worker(void*) {
    while (1) {
        int fd = dequeue();
        read(fd, buf, ...);
        write(fd, buf, ...);
    }
}
```

✅ 7. 고성능 멀티코어 서버 구현 팁

- `CPU Core` 수에 맞는 워커 수 조정
- `SO_REUSEPORT` vs `accept mutex` 선택 전략
- NUMA 환경에선 CPU 바인딩 고려 (`sched_setaffinity`)
- 부하 측정 도구 병행 (`htop`, `perf`, `strace`)

✅ 8. 실전 적용 예

프레임워크 / 서버	멀티코어 방식
nginx	`prefork + SO_REUSEPORT`
Node.js Cluster	`fork()` + IPC
libevent	쓰레드풀 or 프로세스 fork 병행 가능
libuv	자체 워커 스레드 + 이벤트루프

14.4 커널 우회 네트워킹 (DPDK, XDP 개요)

🎯 학습 목표

리눅스 커널 네트워크 스택을 **우회(bypass)**하거나 **최적화**하는 기술은 초고속 트래픽 처리, 패킷 필터링, 네트워크 기능 가상화(NFV) 등에서 필수적이다.
이 절에서는 대표적인 커널 우회 기술인 **DPDK**와 **XDP/eBPF**에 대해 개념과 구조, 실제 적용 방향을 이해하는 데 초점을 둔다.

✅ 1. 왜 커널 우회가 필요한가?

기존 커널 네트워크 스택 문제점:

- 복잡한 계층 구조 (L2 → L3 → L4)
- 많은 context switch, copy overhead
- user space에서 패킷 수신/전송 시 지연 증가

필요성:

- 초당 **수백만 패킷 처리 (Mpps)**
- 지연 수 μs 이하로 단축
- NIC → Application 직접 연결하는 구조

✅ 2. DPDK (Data Plane Development Kit)

🔹 개요

- **Intel**이 주도 개발한 **user-space 고속 패킷 처리 프레임워크**
- **polling 기반**, **zero-copy**, **NUMA-aware**

🔹 특징

항목	설명
Polling 방식	NIC으로부터 인터럽트 없이 반복 수신
Zero-copy	NIC → 메모리 → App로 직접 전달 (커널 우회)

항목	설명
HugePages 사용	대용량 페이지를 통한 TLB miss 최소화
다중 코어 처리	멀티큐, 멀티스레드 기반 병렬 수신 처리

🔹 구조

```
[NIC] → [DPDK Poller] → [User App]
        ↘ HugePages / DMA
```

🔹 설치 개념

```
sudo apt install dpdk dpdk-dev
```

→ 실제 사용은 바인딩된 NIC에서만 가능 (`vfio-pci`, `uio_pci_generic` 등 사용)

✅ DPDK 코드 예시 스케치

```
rte_eth_rx_burst(port_id, queue_id, &mbuf, 32);  // 패킷 수신
rte_pktmbuf_mtod(mbuf, void*);                   // 패킷 접근
rte_eth_tx_burst(port_id, queue_id, &mbuf, 1);   // 패킷 전송
```

✅ 3. XDP (eXpress Data Path)

🔹 개요

- 리눅스 커널 내에 **eBPF 프로그램**을 NIC에 **가장 가깝게 실행**하는 기술
- `kernel bypass`는 아니고, `kernel fastpath`에 해당

🔹 동작 위치

```
[NIC] → [XDP/eBPF] → drop / pass / redirect
```

- 커널 **L2/L3 스택 진입 전**에 처리
- `drop`, `tx`, `pass`, `redirect`, `abort` 등의 액션 제공

🔹 XDP 장점

항목	설명
빠른 실행 속도	NIC 수신 직후 eBPF 실행
커널 기능 사용 가능	필요 시 netfilter, tc 등과 연계
필터링/통계 목적에 강함	DDoS 차단, 통계 수집에 적합

항목	설명
전용 드라이버 필요 없음	대부분의 커널에서 사용 가능

🔹 간단한 eBPF/XDP 프로그램 흐름 (C 코드 기반)

```c
SEC("xdp")
int xdp_prog(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    struct ethhdr *eth = data;
    if (data + sizeof(*eth) > data_end)
        return XDP_ABORTED;

    if (eth->h_proto == htons(ETH_P_IP))
        return XDP_PASS;  // 통과
    else
        return XDP_DROP;  // 차단
}
```

컴파일 후 `ip link set dev eth0 xdp obj myprog.o` 식으로 NIC에 바인딩

✅ 4. DPDK vs XDP 비교

항목	DPDK	XDP/eBPF
속도	매우 빠름 (1000만 PPS 이상)	빠름 (μs 단위)
위치	완전 유저 공간	커널 진입 직후
목적	고속 패킷 처리, 트래픽 엔진	필터링, 통계, 보안
사용 난이도	높음 (NIC 드라이버, HugePages 필요)	낮음 (표준 커널만으로 가능)
NIC 제한	특정 드라이버만 사용 가능 (`vfio`)	대부분 드라이버 지원 (`mlx, ixgbe, ...`)

✅ 5. 실전 적용 예시

분야	적용 방식
IDS/Firewall	XDP로 의심 패킷 빠르게 차단
DDoS 방어	XDP 프로그램으로 drop 처리
패킷 생성/스푸핑	DPDK로 직접 패킷 조작
벤치마크 도구	DPDK + custom app

분야	적용 방식
Load balancer	XDP + eBPF redirect

✅ 6. 참고 도구

- DPDK 예제: `/usr/share/dpdk/examples/*`
- XDP 튜토리얼: https://github.com/xdp-project
- DPDK 성능 측정: `testpmd`, `pktgen`

14.5 IoT 디바이스 네트워크 연동 (C 기반 TCP/IP Stack 내장)

🎯 학습 목표

소형 IoT 디바이스(센서, MCU, 무선 모듈 등)는 일반 리눅스 커널이나 OS를 쓰지 못하므로,
네트워크 기능을 구현하기 위해 자체 내장 TCP/IP 스택이 필요하다.
이 절에서는 **경량 TCP/IP 스택 (uIP, lwIP)** 을 C 언어로 IoT 장비에 연동하는 방법을 학습한다.

✅ 1. 왜 경량 TCP/IP 스택이 필요한가?

일반 리눅스 vs IoT MCU 차이

항목	리눅스 시스템	IoT MCU (STM32 등)
OS 존재	Linux 커널	보통 없음 또는 RTOS
메모리	수십 MB~GB	수십 KB~수 MB
시스템 콜	사용 가능	없음
BSD Socket API	사용 가능	직접 구현해야 함

➤ 그래서 소형 C 기반 TCP/IP 스택을 MCU에 포팅해야 한다.

✅ 2. 대표적인 경량 TCP/IP 스택

스택 이름	설명
uIP	8/16비트 MCU용으로 만든 초경량 TCP/IP 스택
lwIP	FreeRTOS 등과 자주 함께 쓰이며, TCP/UDP/IPv4/IPv6, HTTP 지원
CycloneTCP	상업적/산업용 목적의 안정적 스택
nanoIP, **uC/TCP-IP** 등도 있음	

✅ 3. lwIP 개요 및 구조

🔹 구조도

```
[ Application (C code) ]
         ↓
 [ lwIP API (sockets / raw API) ]
         ↓
[ lwIP Core (tcp, udp, ip, dhcp) ]
         ↓
[ HAL 드라이버 (eth, wifi 등) ]
```

🔹 주요 구성 모듈

- `tcp.c`, `udp.c`, `ip.c`, `netif.c` 등
- 인터페이스 드라이버: `netif/ethernetif.c`, `low_level_output()`

✅ 4. lwIP 사용 방식

설치 (일반적으로는 직접 포팅)

- STM32CubeMX → **LWIP Enable** 체크 → 자동 생성

인터페이스 구성

- 인터페이스 초기화: `netif_add()`
- 패킷 수신 후 입력 큐에 삽입: `netif_input()`
- 송신 시 하드웨어 전송 함수 호출: `low_level_output()`

메인 루프 예시 (No OS 모드)

```
void main_loop() {
    struct pbuf* p;
    while (1) {
        p = ethernet_input(); // NIC로부터 직접 수신
        if (p != NULL) {
            if (netif.input(p, &netif) != ERR_OK)
                pbuf_free(p);
        }
        sys_check_timeouts(); // 타이머 관리
    }
}
```

✅ 5. TCP 서버 예제 (`lwip/tcp_echo.c`)

```
static err_t recv_cb(void* arg, struct tcp_pcb* tpcb, struct pbuf* p, err_t err) {
    if (!p) return ERR_OK;
    tcp_write(tpcb, p->payload, p->len, TCP_WRITE_FLAG_COPY);
    pbuf_free(p);
    return ERR_OK;
}

void tcp_echo_init(void) {
    struct tcp_pcb* pcb = tcp_new();
    tcp_bind(pcb, IP_ADDR_ANY, 1234);
    pcb = tcp_listen(pcb);
    tcp_accept(pcb, [](void* arg, struct tcp_pcb* newpcb, err_t err) {
        tcp_recv(newpcb, recv_cb);
        return ERR_OK;
    });
}
```

✅ 6. 연결된 하드웨어 계층 연동

- STM32, ESP32, NRF52 등 MCU에서는 아래 계층 필요:

```
[ lwIP TCP/IP ]
      ↑
[ Ethernet HAL ] ← `HAL_ETH_Transmit`, `HAL_ETH_GetReceivedFrame`
      ↑
[ DMA/PHY/Driver ]
```

- RTOS 없이도 동작 가능 (`NO_SYS=1` 설정 시)

✅ 7. 실전 예: STM32 + lwIP + DHCP + ping

- STM32CubeMX에서:
  - `LWIP enabled`
  - `DHCP client enabled`
- 코드:

```
MX_LWIP_Init(); // 자동 생성된 초기화 코드
while (1) {
  MX_LWIP_Process();  // Polling 기반 lwIP 처리
}
```

- IP 할당 → ping → TCP 접속 가능

✅ 8. 확장 응용

기능	방법
HTTP 서버	`httpd.c` 내장
MQTT Client	`lwip/apps/mqtt`
TLS/HTTPS	mbedTLS와 연동
DNS/DHCP/NTP	별도 옵션 활성화
OTA 업데이트	TFTP 또는 HTTP 기반 전송 처리

✅ 9. 고급 개발 팁

- `pbuf` 구조는 버퍼 체인으로 구성됨 → 포인터 해제 주의
- TCP 세션 유지 위해 `tcp_poll`, `tcp_sent` 이벤트도 등록 필요
- 인터럽트 내에서 직접 `tcp_write()` 사용 금지 → 이벤트 큐 활용
- 메모리 풀이 부족할 경우 `MEM_SIZE`, `PBUF_POOL_SIZE` 조정