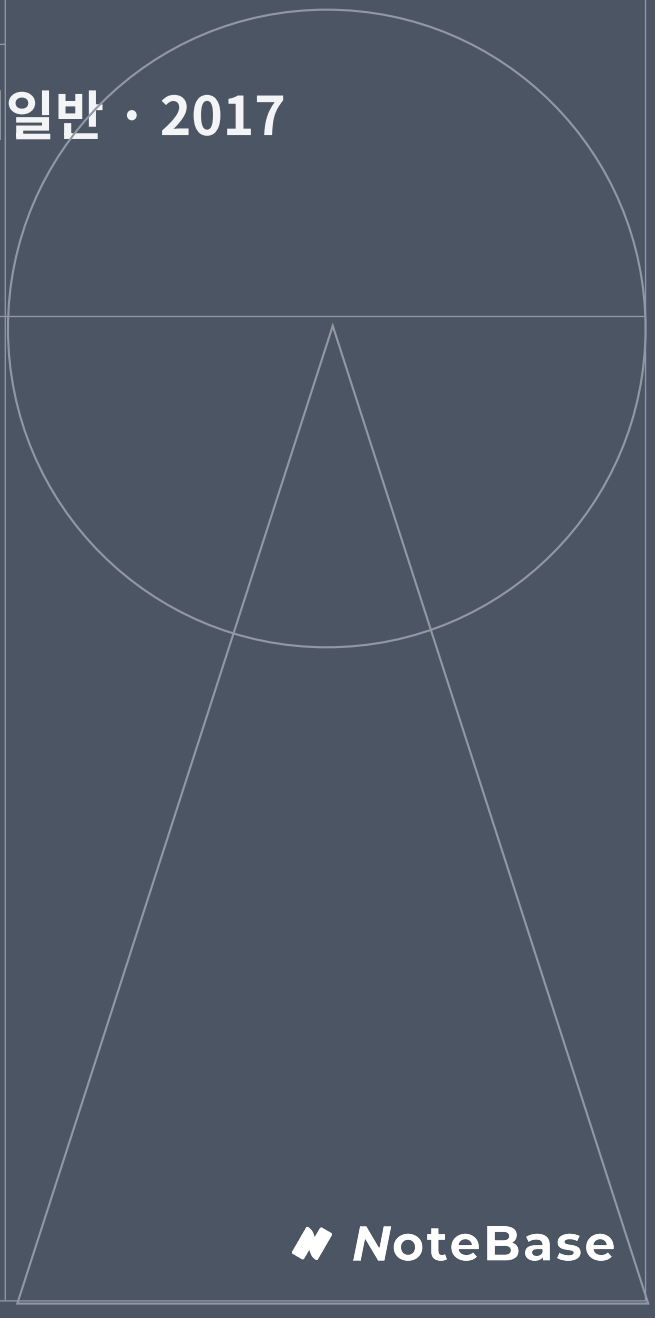
기출OX 해설집

공무원/서울시/9급 · 컴퓨터일반 · 2017



NoteBase

기출OX 해설집

공무원/서울시/9급 · 컴퓨터일반 · 2017

공무원/서울시/9급 · 컴퓨터일반 · 2017

문 1. 다음은 컴퓨터 언어처리에 관련된 시스템 S/W의 기능을 설명한 것이다. 옳지 않은 것은?

① 컴파일러: 고급언어를 이진목적모듈로 변환기능

② 어셈블러: 객체지향언어를 이진목적모듈로 변환기능

③ 링커: 여러 목적모듈을 통합하여 실행 가능한 하나의 모듈로 변환기능

④ 로더: 실행 가능한 모듈을 주기억장치에 탑재기능

정답 2번

간단한 해설

어셈블러가 번역하는 대상은 객체지향언어가 아니라 **어셈블리어**이다. 어셈블리어로 작성한 명령을 기계어로 바꾸어 이진 목적모듈을 만든다.

선지별 해설

① O: 컴파일러는 C와 같은 **고급언어로 작성한 소스 프로그램**을 번역하여 기계어로 된 목적모듈을 생성한다. 이렇게 생성된 목적모듈은 필요한 다른 모듈이나 라이브러리와 연결하는 과정을 거쳐 실행 파일이 된다.

② X: 어셈블러는 **어셈블리어를 기계어로 번역**하여 이진 목적모듈을 생성한다. 어셈블리어는 기계 명령을 사람이 읽기 쉬운 기호로 표현한 언어이다. 따라서 선지의 '객체지향언어'를 '어셈블리어'로 고쳐야 한다.

③ O: 링커는 여러 목적모듈과 필요한 라이브러리를 결합하여 하나의 실행 가능한 모듈을 만든다. 이때 한 모듈에서 호출한 함수가 다른 모듈에 정의되어 있으면 **그 참조를 실제 정의와 연결**한다.

④ O: 로더는 실행 가능한 모듈을 **주기억장치에 적재**하여 실행을 준비한다. 링커가 모듈 사이의 연결을 담당한다면, 로더는 만들어진 실행 모듈을 메모리에 배치하는 역할을 한다.

NoteBase

문 2. 컴퓨터에서 사건이 발생하면 이를 처리하기 위해 인터럽트 기술을 사용한다. 사건의 발생지에 따라 동기와 비동기 인터럽트로 분류된다. 다음 중 비동기 인터럽트는?

① 프로세스가 실행 중에 0으로 나누기를 할 때 발생하는 인터럽트

② 키보드 혹은 마우스를 사용할 때 발생하는 인터럽트

③ 프로세스 내 명령어 실행 때문에 발생하는 인터럽트

④ 프로세스 내 명령어가 보호 메모리영역을 참조할 때 발생하는 인터럽트

정답 2번

간단한 해설

키보드나 마우스의 입력은 **현재 실행 중인 명령어와 무관하게 외부 장치에서 발생**하므로 비동기 인터럽트에 해당한다.

공통 해설

동기·비동기 인터럽트를 구분하는 기준은 **현재 실행 중인 명령어가 사건의 직접적인 원인인지**이다. 동기 인터럽트는 해당 명령어의 실행 때문에 발생하고, 비동기 인터럽트는 명령어 실행과 독립적으로 외부 장치 등의 사건에 의해 발생한다.

키보드나 마우스의 입력에 따른 인터럽트는 사용자의 입력을 장치가 CPU에 알리는 것이다. 예를 들어 CPU가 계산 명령어를 실행하는 동안에도 사용자가 키를 누르면 입력 장치의 인터럽트가 발생할 수 있다. 계산 명령어 자체가 입력 사건을 일으킨 것이 아니므로 **비동기 인터럽트**로 분류한다.

반면 0으로 나누는 연산의 오류나 접근 권한이 없는 보호 메모리영역의 참조는 해당 명령어를 실행하는 과정에서 검출된다. 이처럼 프로세스 내 명령어의 실행 때문에 발생하는 인터럽트는 **동기 인터럽트**에 해당한다.

NoteBase

문 3. 데이터통신에서 에러 복구를 위해 사용되는 Go-back-N ARQ에 대한 설명으로 옳지 않은 것은?

① Go-back-N ARQ는 여러 개의 프레임들을 순서번호를 붙여서 송신하고, 수신 측은 이 순서번호에 따라 ACK 또는 NAK를 보낸다.

② Go-back-N ARQ는 송신 측은 확인응답이 올 때까지 전송된 모든 프레임의 사본을 갖고 있어야 한다.

③ Go-back-N ARQ는 재전송 시 불필요한 재전송 프레임들이 존재하지 않는다.

④ Go-back-N ARQ는 송신 측은 n개의 Sliding Window를 가지고 있어야 한다.

정답 3번

간단한 해설

Go-back-N ARQ는 오류가 발생한 프레임부터 이후의 미확인 프레임까지 **함께 재전송**한다. 오류가 없던 후속 프레임도 다시 보낼 수 있으므로, '불필요한 재전송 프레임들이 존재하지 않는다'는 설명은 잘못이다.

선지별 해설

① O: 송신 측은 여러 프레임에 순서번호를 붙여 전송하고, 수신 측은 이 번호로 수신 순서를 확인한다. ACK는 정상 수신을 알리는 응답이고, NAK는 재전송을 요구하는 응답이다. **NAK를 사용하는 방식**에서는 누락되거나 손상된 프레임의 번호를 알려 그 프레임부터 재전송하도록 한다. NAK 없이 ACK와 타임아웃으로 동작하는 방식도 있다.

② O: 송신 측은 재전송에 대비하여 **확인응답을 받지 못한 프레임의 사본**을 보관해야 한다. 확인응답이 오기 전에는 해당 프레임이 정상적으로 수신되었는지 확정할 수 없기 때문이다. 정상 수신이 확인된 프레임의 사본은 버릴 수 있다.

③ X: Go-back-N ARQ는 오류가 발생한 프레임뿐 아니라 **그 뒤에 보낸 미확인 프레임도 함께 재전송**하므로, 오류가 없던 프레임의 재전송이 발생할 수 있다. 예를 들어 1~4번 프레임을 보냈는데 2번이 유실되면, 수신 측은 순서에 맞지 않게 도착한 3·4번도 버린다. 이후 송신 측은 2·3·4번을 다시 보내므로, 처음 전송할 때 손상되지 않았던 3·4번까지 재전송한다.

④ O: 선지의 'n개의 Sliding Window'는 **n개 프레임을 관리하는 크기 n의 송신 윈도**를 사용한다는 취지로 이해한다. 송신 측은 이 범위 안에서 확인응답을 기다리지 않고 여러 프레임을 연속 전송하며, 확인응답을 받으면 윈도를 앞으로 이동시킨다. n은 윈도 자체의 개수가 아니라 확인응답 없이 전송할 수 있는 최대 프레임 수를 뜻한다.

NoteBase

문 5. 초기에 빈 Binary Search Tree를 생성하고, 입력되는 수는 다음과 같은 순서로 된다고 가정한다. 입력되는 값을 이용하여 Binary Search Tree를 만들고 난 후 Inorder Traversal을 했을 때의 방문하는 순서는?

7, 5, 1, 8, 3, 6, 0, 2

① 01235678

② 02316587

③ 75103268

④ 86230157

정답 1번

간단한 해설

이진 탐색 트리를 **중위 순회**하면 왼쪽 부분 트리, 현재 노드, 오른쪽 부분 트리 순으로 방문한다. 각 노드보다 작은 값은 왼쪽에, 큰 값은 오른쪽에 있으므로 방문 결과는 **오름차순**이 된다.

NoteBase

올바른 순서

0 → 1 → 2 → 3 → 5 → 6 → 7 → 8

이진 탐색 트리는 삽입할 값을 현재 노드와 비교하여 작으면 왼쪽으로, 크면 오른쪽으로 내려가 빈 위치에 넣는다. 처음 삽입한 7이 루트가 된다. 중위 순회는 각 부분 트리에서도 **왼쪽 부분 트리 → 현재 노드 → 오른쪽 부분 트리** 순서를 적용한다.

0: 삽입할 때 7, 5, 1보다 차례로 작으므로 1의 왼쪽 자식이 된다. 루트 7에서 왼쪽으로 끝까지 내려간 노드이므로 가장 먼저 방문한다.

1: 왼쪽 자식 0의 방문을 마쳤으므로 현재 노드 1을 방문한다. 오른쪽 부분 트리는 그다음에 처리한다.

2: 1의 오른쪽 자식은 3이다. 3을 바로 방문하지 않고 **3의 왼쪽 자식인 2부터** 방문한다.

3: 왼쪽 자식 2의 방문을 마친 뒤 3을 방문한다. 3에는 오른쪽 자식이 없으므로 1을 루트로 하는 부분 트리의 순회가 끝난다.

5: 7의 왼쪽 자식인 5는 왼쪽에 1을 루트로 하는 부분 트리를 둔다. 그 부분 트리의 방문이 끝났으므로 5를 방문한다.

6: 6은 7보다 작고 5보다 커서 5의 오른쪽 자식이 된다. 5를 방문한 뒤 오른쪽의 6을 방문한다.

7: 5를 루트로 하는 왼쪽 부분 트리의 순회가 모두 끝났으므로 전체 트리의 루트 7을 방문한다.

8: 8은 7보다 커서 7의 오른쪽 자식이 된다. 7을 방문한 뒤 오른쪽 부분 트리의 유일한 노드인 8을 방문하여 순회를 마친다.

NoteBase

문 6. 통신 S/W 구조에서 제2계층인 데이터링크 계층의 주기능이 아닌 것은?

① 데이터링크 계층에서 전송할 프레임(Frame) 제작 기능

② 점대점(Point to Point) 링크 간의 오류제어 기능

③ 종단(End to End) 간 경로설정 기능

④ 점대점(Point to Point) 링크 간의 흐름제어 기능

정답 3번

간단한 해설

③의 '종단 간 경로설정'은 **제3계층인 네트워크 계층**의 기능이다. 데이터링크 계층은 하나의 링크로 연결된 인접 장치 사이의 프레임 전송을 담당한다.

선지별 해설

① O: 데이터링크 계층은 상위 계층에서 받은 데이터를 **프레임 단위로 구성**한다. 프레임의 경계를 구분하고 전송에 필요한 제어 정보를 붙여 물리 계층을 통해 보낼 수 있게 한다.

② O: 데이터링크 계층의 오류제어는 **인접 장치 사이의 링크**에서 발생한 전송 오류를 다룬다. 오류 검출 정보를 이용해 손상된 프레임을 찾아내며, 사용하는 프로토콜에 따라 확인 응답과 재전송으로 오류를 복구하기도 한다.

③ X: 출발지에서 목적지까지 여러 네트워크를 거쳐 갈 경로를 정하는 것은 **네트워크 계층의 경로설정 기능**이다. 데이터링크 계층은 전체 경로를 정하는 대신 그 경로를 이루는 개별 링크에서 인접 장치로 프레임을 전달한다. 따라서 선지의 기능은 제2계층이 아니라 제3계층에 해당한다.

④ O: 데이터링크 계층의 흐름제어는 인접한 수신 장치가 처리할 수 있는 범위에서 프레임을 받도록 **송신 측의 전송량이나 속도를 조절**하는 기능이다. 수신 측이 처리하기 어려울 만큼 프레임이 몰리는 것을 막는다.

NoteBase

문 7. 다음 중 집적도가 가장 높은 회로와 가장 큰 저장용량 단위를 나타낸 것은?

GB, PB, MB, TB
VLSI, MSI, ULSI, SSI

① ULSI, PB
② VLSI, TB
③ MSI, GB
④ SSI, PB

정답 1번

간단한 해설

제시된 회로 중에서는 **ULSI**의 집적도가 가장 높다. 저장용량 단위는 MB, GB, TB, PB 순으로 커지므로 이 중 가장 큰 단위는 **PB**이다.

구성 요소별 해설

회로의 집적도: 집적도는 한 칩에 얼마나 많은 회로 소자를 집적했는지를 나타낸다. 제시된 회로를 집적도가 낮은 것부터 나열하면 SSI < MSI < VLSI < ULSI이다. 따라서 가장 높은 집적도를 나타내는 회로는 **ULSI**이며, VLSI·MSI·SSI는 이보다 집적도가 낮다.

저장용량 단위: MB는 메가바이트, GB는 기가바이트, TB는 테라바이트, PB는 페타바이트이다. 단위의 크기는 **MB < GB < TB < PB** 순이므로 제시된 단위 중 PB가 가장 크다. GB나 TB는 PB보다 작은 용량 단위이다.

가장 높은 집적도의 회로와 가장 큰 저장용량 단위를 함께 나타낸 조합은 ①의 ULSI, PB이다.

NoteBase

문 8. 다음은 다중스레드(Multi-Thread)에 관련된 설명이다. 옳지 않은 것은?

① 하나의 프로세스에 2개 이상의 스레드들을 생성하여 수행한다.

② 스레드별로 각각의 프로세스를 생성하여 실행하는 것보다 효율적이다.

③ 스레드들 간은 IPC(InterProcess Communication) 방식으로 통신한다.

④ 각각의 스레드는 프로세스에 할당된 자원을 공유한다.

정답 3번

간단한 해설

같은 프로세스의 스레드들은 **주소 공간을 공유**하므로 공유 변수 등에 직접 접근하여 데이터를 주고받을 수 있다. 따라서 스레드 간 통신에 별도의 프로세스 간 통신 방식인 IPC가 필요한 것은 아니다.

선지별 해설

① O: 스레드는 프로세스 안에서 실행되는 작업의 흐름이다. 다중스레드는 **하나의 프로세스 안에 둘 이상의 실행 흐름**을 두어 여러 작업을 수행하는 방식이다.

② O: 스레드는 같은 프로세스의 메모리와 자원을 함께 사용하므로, 작업마다 별도 프로세스를 만드는 것보다 일반적으로 **생성과 문맥 교환의 부담이 작다**. 다만 실제 실행 성능은 작업의 성격과 동기화 비용 등에 따라 달라진다.

③ X: IPC는 서로 다른 프로세스가 데이터를 주고받기 위한 통신을 뜻한다. 같은 프로세스의 스레드들은 주소 공간을 공유하므로 **별도의 IPC 없이 공유 데이터에 직접 접근**할 수 있다. 예를 들어 한 스레드가 공유 변수에 저장한 값을 다른 스레드가 읽을 수 있다. 여러 스레드가 같은 데이터를 동시에 변경한다면 뮤텍스 등으로 접근을 동기화해야 한다.

④ O: 같은 프로세스의 스레드들은 **코드·데이터·힙 영역과 열린 파일 등의 자원**을 공유한다. 반면 각 스레드의 실행 상태를 유지하는 프로그램 카운터, 레지스터, 스택은 스레드마다 따로 가진다.

NoteBase

문 9. 다음과 같이 3개의 프로세스가 있다고 가정한다. 각 프로세스의 도착 시간과 프로세스의 실행에 필요한 시간은 아래 표와 같다. CPU 스케줄링 알고리즘으로 RR(Round Robin)을 사용한다고 가정한다. 3개의 프로세스가 CPU에서 작업을 하고 마치는 순서는? (단, CPU를 사용하는 타임 슬라이스(time slice)는 2이다.)

프로세스	도착시간	프로세스의 실행에 필요한 시간
P1	0	5
P2	1	7
P3	3	4

① P2, P1, P3

② P2, P3, P1

③ P1, P2, P3

④ P1, P3, P2

정답 4번

간단한 해설

P3이 도착했을 때 **P1이 이미 준비 큐에서 기다리고 있으므로** P1이 먼저 CPU를 다시 받는다. P1이 완료되기 전 P2는 한 차례 더 실행해도 3이 남지만, P3은 **남은 2만 실행하면 완료**되므로 P2보다 먼저 끝난다.

NoteBase

올바른 순서

P1 → P3 → P2

RR은 준비 큐의 맨 앞 프로세스에 CPU를 배정하고, 타임 슬라이스 2를 모두 사용해도 끝나지 않으면 그 프로세스를 큐의 맨 뒤로 보낸다. 실행 중 새로 도착한 프로세스도 준비 큐의 뒤에 들어간다. 표에서 P1·P2·P3의 도착 시각은 각각 0·1·3이고, 필요한 총 실행 시간은 각각 5·7·4이다.

P1 완료: 시각 0~2에 P1이 실행되어 3이 남는다. 시각 1에 도착한 P2가 먼저 기다리고 있으므로, 이어서 시각 2~4에는 P2가 실행되어 5가 남는다. P3이 시각 3에 도착할 때에는 **P1이 이미 준비 큐에서 기다리고 있으므로**, 시각 4~6에는 P1이 실행되어 1이 남고 시각 6~8에는 P3이 실행되어 2가 남는다. 시각 8의 준비 큐는 P2 → P1 → P3이므로, 시각 8~10에는 P2가 실행되어 3이 남는다. 이어서 P1은 시각 10~11에 남은 1만 실행하고 완료된다. 작업이 끝났으므로 타임 슬라이스 2를 다 채울 필요는 없다.

P3 완료: P1이 끝난 시각 11에는 P3이 P2보다 앞에서 기다린다. 따라서 시각 11~13에 P3이 남은 2를 실행해 완료된다. P2가 먼저 CPU를 받았더라도 **한 차례 배정된 시간 안에 끝나지 않았으므로** P3보다 늦게 완료된다.

P2 완료: 시각 13에는 P2만 남고, 필요한 실행 시간은 3이다. 시각 13~15에 2를 실행한 뒤 남은 1을 시각 15~16에 실행하여 완료된다. 다른 대기 프로세스가 없으므로 **타임 슬라이스가 끝나도 P2가 곧바로 다시 CPU를 받는다**.

NoteBase

문 10. 다음의 C프로그램을 실행한 결과로 옳은 것은?

```
#include <stdio.h>
void main() {
  int num[4] = {1, 2, 3, 4};
  int *pt = num;

  pt++;
  *pt++ = 5;
  *pt += 10;
  pt--;
  *pt += 20;
  printf("%d %d %d %d", num[0], num[1], num[2], num[3]);
}
```

① 1 5 10 20

② 1 5 20 4

③ 1 5 30 4

④ 에러 발생

정답 3번

간단한 해설

후위 증가 연산이 붙은 *pt++는 **기존 포인터가 가리키는 원소를 사용하고 포인터를 다음 원소로 이동**시킨다.
마지막 *pt++ += 20은 num[2]의 값 10에 20을 더하므로 출력은 1 5 30 4이다.

NoteBase

계산 과정

초기 상태: 배열 num의 값은 1, 2, 3, 4이고, int *pt = num에 의해 pt는 첫 번째 원소 num[0]을 가리킨다. 정수형 포인터를 1 증가시키면 다음 정수 원소로 이동한다.

pt++ 실행: 포인터가 한 원소 앞으로 이동하여 num[1]을 가리킨다. 이 문장은 배열 원소의 값을 바꾸지 않는다.

*pt++ = 5 실행: 후위 증가 ++가 간접 참조 *보다 우선하므로 *pt++는 ***(pt++)로 해석**한다. 증가 전 포인터가 가리키던 num[1]에 5를 저장하고, 문장이 끝나면 pt는 num[2]를 가리킨다. 배열은 1, 5, 3, 4가 된다.

*pt++ = 10 실행: 같은 원리에 따라 num[2]에 10을 저장하고 pt는 num[3]을 가리키게 된다. 배열은 1, 5, 10, 4가 된다.

pt-- 실행: 포인터가 한 원소 뒤로 이동하여 **num[2]를 다시 가리킨다**. 배열의 값은 그대로이다.

*pt++ += 20 실행: 이 식은 *(pt++) += 20으로 해석한다. 복합 대입 +=는 왼쪽 원소의 기존 값에 오른쪽 값을 더하여 저장하므로, **num[2]가 10 + 20 = 30**으로 바뀐다. 문장이 끝나면 pt는 num[3]을 가리키지만 num[3]의 값 4는 바뀌지 않는다.

출력: printf는 num[0]부터 num[3]까지 차례로 출력한다. 최종 배열 값은 1, 5, 30, 4이므로 출력은 1 5 30 4이며, ③과 일치한다.

NoteBase

문 11. Flynn의 병렬컴퓨터 분류방식에 대한 설명으로 옳지 않은 것은?

① SISD - 명령어와 데이터를 순서대로 처리하는 단일프로세서 시스템이다.

② SIMD - 단일 명령어 스트림을 처리하고 배열프로세서라고도 한다.

③ MISD - 여러 개의 프로세서를 갖는 구조로 밀결합 시스템(tightly-coupled system)과 소결합 시스템(loosely-coupled system)으로 분류한다.

④ MIMD - 여러 개의 프로세서들이 서로 다른 명령어와 데이터를 처리하는 진정한 의미의 병렬프로세서이다.

정답 3번

간단한 해설

③의 밀결합·소결합 시스템 구분은 **MIMD**에 대한 설명이다. MISD는 **하나의 데이터 스트림에 여러 명령어 스트림을 적용**하는 구조이다.

선지별 해설

Flynn은 동시에 처리하는 명령어 스트림과 데이터 스트림이 각각 하나인지 여러 개인지를 기준으로 컴퓨터 구조를 분류한다. 명령어 스트림은 실행할 명령어의 흐름이고, 데이터 스트림은 그 명령어가 처리하는 데이터의 흐름이다.

① O: SISD는 단일 명령어 스트림과 단일 데이터 스트림을 사용하는 구조이다. 하나의 프로세서가 명령어에 따라 데이터를 순차적으로 처리하는 **전통적인 순차 처리 방식**에 해당한다.

② O: SIMD는 하나의 명령어 스트림으로 여러 처리 장치가 **서로 다른 데이터에 같은 연산**을 수행하는 구조이다. 배열의 여러 원소를 동시에 처리하는 배열프로세서가 대표적이다. 예를 들어 배열의 각 원소에 1을 더하는 연산을 여러 처리 장치가 나누어 동시에 수행한다.

③ X: 밀결합 시스템과 소결합 시스템으로 나누는 것은 **MISD가 아니라 MIMD**에 대한 설명이다. 밀결합 시스템은 여러 프로세서가 공통 메모리를 공유하고, 소결합 시스템은 각 프로세서가 자신의 메모리를 가지며 통신으로 데이터를 교환한다. MISD는 여러 명령어 스트림이 하나의 데이터 스트림을 처리하는 구조이므로, 선지의 'MISD'를 'MIMD'로 고쳐야 한다.

④ O: MIMD는 여러 프로세서가 각자의 명령어 스트림과 데이터 스트림을 처리하는 구조이다. 프로세서들이 **서로 다른 명령어를 독립적으로 실행**할 수 있어 여러 작업을 병렬로 수행한다. 모든 처리 장치가 같은 명령어를 실행하는 SIMD와 이 점에서 구별된다.

NoteBase

문 12. 네트워크 토폴로지(Topology) 중 버스(Bus) 방식에 대한 설명으로 옳지 않은 것은?

① 버스 방식은 네트워크 구성이 간단하고 작은 네트워크에 유용하며 사용이 용이하다.

② 버스 방식은 네트워크 트래픽이 많을 경우 네트워크 효율이 떨어진다.

③ 버스 방식은 통신 채널이 단 한 개이므로 버스 고장이 발생하면 네트워크 전체가 동작하지 않으므로 여분의 채널이 필요하다.

④ 버스 방식은 노드의 추가·삭제가 어렵다.

정답 4번

간단한 해설

버스 방식은 하나의 공통 통신선에 노드를 연결하므로 **노드의 추가·삭제가 비교적 쉽다**. 따라서 ④의 '어렵다'는 설명은 버스 방식의 일반적인 특징과 맞지 않는다.

선지별 해설

① O: 버스 방식은 여러 노드가 하나의 공통 통신선인 버스에 접속하는 구조이다. 연결 구조가 단순하고 필요한 배선이 비교적 적어 **소규모 네트워크를 구성하기 쉽다**.

② O: 여러 노드가 **하나의 통신 채널을 공유**하므로 트래픽이 많아지면 전송 기회를 기다리는 시간이 늘어난다. 충돌을 허용하는 매체 접근 방식에서는 충돌과 재전송도 증가할 수 있어 네트워크 효율이 떨어진다.

③ O: 모든 노드의 통신이 공통 버스에 의존하므로 **버스 자체에 고장이 나면 전체 네트워크의 통신이 중단**될 수 있다. 선지의 '여분의 채널이 필요하다'는 이러한 고장에 대비하여 통신을 계속하려면 대체 채널을 마련해야 한다는 뜻이다. 여분의 채널이 버스 방식을 구성하는 필수 요소라는 뜻은 아니다.

④ X: 버스 방식은 기존 공통 통신선에 노드를 접속하거나 그 접속을 해제하는 방식이므로 **노드의 추가·삭제가 비교적 쉽다**. 다른 노드 사이의 연결 구조 전체를 다시 구성할 필요가 없으므로 '추가·삭제가 어렵다'를 '추가·삭제가 비교적 쉽다'로 고쳐야 한다.

NoteBase

문 13. 정보은닉(information hiding)에 대한 설명으로 옳지 않은 것은?

① 필요하지 않은 정보는 접근을 제한하는 것이다.

② 모듈 사이의 독립성을 유지시킨다.

③ 설계전략을 지역화하여 전략의 변경에 따른 영향을 최소화한다.

④ 모듈 사이의 결합도를 높여 신뢰성을 향상시킨다.

정답 4번

간단한 해설

정보은닉은 다른 모듈에 필요한 기능만 공개하고 내부 구현은 감추어 **모듈 사이의 결합도를 낮추는** 설계 원리이다. 다른 모듈의 내부 구현에 대한 의존을 줄여 변경이나 오류의 영향을 제한하므로, 결합도를 높인다는 설명은 반대이다.

선지별 해설

① O: 정보은닉은 모듈을 사용하는 데 **필요한 인터페이스만 공개**하고 내부 자료구조나 처리 방법은 외부에서 직접 접근하지 못하게 한다. 여기서 필요하지 않은 정보란 다른 모듈이 해당 기능을 이용할 때 알 필요가 없는 내부 구현 정보를 뜻한다.

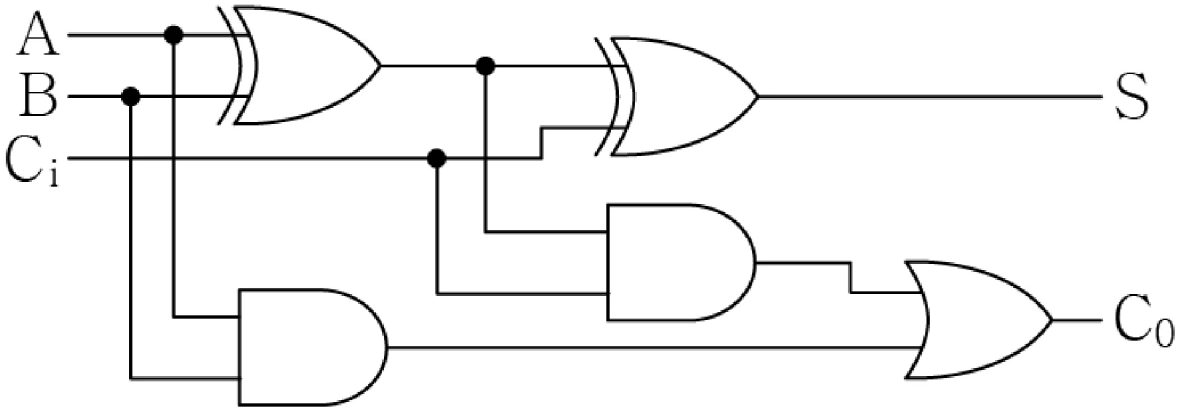
② O: 다른 모듈이 내부 구현에 의존하지 않고 공개된 인터페이스를 통해 기능을 사용하게 하므로 **모듈 사이의 독립성**을 높인다. 인터페이스가 유지되면 내부 구현을 바꾸더라도 이를 사용하는 모듈의 수정 필요성을 줄일 수 있다.

③ O: 변경될 수 있는 설계 결정을 **해당 모듈 내부에 한정**하여 변경의 파급을 줄인다. 예를 들어 자료 저장 방식을 배열에서 연결 리스트로 바꾸어도 외부에 공개한 삽입·삭제 기능의 사용 방법과 동작이 같다면, 그 기능을 이용하는 다른 모듈은 내부 저장 방식의 변경을 알 필요가 없다.

④ X: 결합도는 모듈들이 서로 의존하는 정도이다. 정보은닉은 내부 구현에 대한 의존을 줄이므로 '결합도를 높여'를 **'결합도를 낮추어'**로 고쳐야 한다. 모듈 간 의존이 줄면 한 모듈의 변경이나 오류가 다른 모듈로 미치는 영향을 제한할 수 있어 신뢰성 향상에 도움이 된다.

NoteBase

문 14. 다음 전가산기 논리회로에 대한 설명으로 옳지 않은 것은?



① 전가산기는 캐리를 포함하여 연산처리하기 위해 설계되었다.

② $S = (A \oplus B) \oplus C_i$

③ $C_0 = AB + AC_i + BC_i$

④ 전가산기는 두 개의 반가산기만으로 구성할 수 있다.

정답 4번

간단한 해설

전가산기는 두 반가산기가 만든 캐리를 합치기 위해 **OR 게이트 1개가 추가로 필요**하다. 따라서 두 개의 반가산기만으로는 그림의 전가산기를 완성할 수 없다.

NoteBase

선지별 해설

① O: 전가산기는 두 입력 비트 A, B와 **이전 자리에서 넘어온 캐리** C_i를 함께 더한다. 결과는 현재 자리의 합 S와 다음 자리로 보내는 캐리 C_0로 나뉜다. 반가산기는 A, B만 더하므로 별도의 입력 캐리를 처리하지 못한다.

② O: 그림 위쪽의 첫 XOR 게이트가 $A \oplus B$를 만들고, 두 번째 XOR 게이트가 여기에 C_i를 XOR하여 합 S를 출력한다. 따라서 $S = (A \oplus B) \oplus C_i$이다. XOR는 두 입력이 서로 다를 때 1을 출력하므로, S는 세 입력 중 **1인 입력의 개수가 홀수일 때** 1이 된다.

③ O: 출력 캐리는 세 입력 중 **적어도 두 개가 1일 때** 발생한다. A와 B, A와 C_i, B와 C_i 중 어느 한 쌍이라도 모두 1이면 되므로 $C_0 = AB + AC_i + BC_i$로 나타낸다. 여기서 곱은 AND, +는 OR 연산을 뜻한다. 그림의 두 AND 게이트와 마지막 OR 게이트가 만드는 $C_0 = AB + (A \oplus B)C_i$도 같은 조건을 나타낸다.

④ X: 반가산기 하나는 XOR 게이트로 합을, AND 게이트로 캐리를 만든다. 첫 반가산기는 A와 B를 더하여 $A \oplus B$와 AB를 만들고, 두 번째 반가산기는 첫 합과 C_i를 더하여 S와 $(A \oplus B)C_i$를 만든다. 이때 두 반가산기의 캐리 출력을 하나로 합치는 회로가 따로 필요하다. 그림 오른쪽 아래의 OR 게이트가 그 역할을 하므로, 전가산기는 **반가산기 2개와 OR 게이트 1개**로 구성할 수 있다.

NoteBase

문 15. 다음은 그래프 순회에서 깊이 우선 탐색 방법에 대한 수행 순서를 설명한 것이다. (ㄱ)~(ㄹ)에 알맞은 내용으로 짝지어진 것은?

(1) 시작 정점 v를 결정하고 방문한다.
(2) 정점 v에 인접한 정점 중에서
(2-1) 방문하지 않은 정점 w가 있으면 정점 v를 (ㄱ)에 (ㄴ)하고 w를 방문한다. 그리고 w를 v로 하여 (2)를 수행한다.
(2-2) 방문하지 않은 정점이 없으면 (ㄱ)을/를 (ㄷ)하여 받은 가장 마지막 방문 정점을 v로 설정한 뒤 다시 (2)를 수행한다.
(3) (ㄹ)이/가 공백이 될 때까지 (2)를 반복한다.

① (ㄱ): Stack, (ㄴ): push, (ㄷ): pop, (ㄹ): Stack
② (ㄱ): Stack, (ㄴ): pop, (ㄷ): push, (ㄹ): Queue
③ (ㄱ): Queue, (ㄴ): enQueue, (ㄷ): deQueue, (ㄹ): Queue
④ (ㄱ): Queue, (ㄴ): enQueue, (ㄷ): deQueue, (ㄹ): Stack

정답 1번

간단한 해설

깊이 우선 탐색은 한 경로를 따라 탐색하다가 더 갈 곳이 없으면 **가장 최근에 저장한 이전 정점**으로 되돌아간다. 이를 위해 후입선출 구조인 스택에 정점을 push로 저장하고 pop으로 꺼낸다.

NoteBase

빈칸별 해설

깊이 우선 탐색은 현재 정점에서 방문하지 않은 인접 정점으로 계속 이동하고, 더 진행할 수 없으면 이전 정점으로 되돌아가 다른 경로를 탐색하는 방법이다.

(ㄱ) Stack: 나중에 저장한 정점으로 먼저 되돌아가야 하므로 **후입선출** 구조인 스택을 사용한다. 먼저 넣은 정점을 먼저 꺼내는 큐는 이 되돌아가기 순서에 맞지 않는다.

(ㄴ) push: 방문하지 않은 정점 w로 이동하기 전에 현재 정점 v를 스택에 저장한다. 나중에 v로 돌아와 다른 인접 정점을 탐색할 수 있도록 **되돌아갈 정점을 넣는 연산**이 push이다.

(ㄷ) pop: 현재 정점에 방문하지 않은 인접 정점이 없으면 스택의 맨 위 정점을 꺼내 그 정점으로 되돌아간다. 이처럼 **가장 최근에 저장한 정점을 꺼내는 연산**이 pop이다. 예를 들어 A에서 B를 거쳐 C로 이동하면서 A, B를 차례로 저장했다면, C에서 더 갈 곳이 없을 때 B를 먼저 꺼내 B의 다른 경로를 살핀다.

(ㄹ) Stack: 공백 여부를 검사하는 대상은 (ㄱ)에서 정점을 저장한 **동일한 스택**이다. 스택에 정점이 남아 있으면 그 정점으로 되돌아가 탐색을 이어 갈 수 있다. 제시된 절차는 이 스택이 빌 때까지 반복한다.

(ㄱ) Stack, (ㄴ) push, (ㄷ) pop, (ㄹ) Stack의 조합은 ①과 일치한다.

NoteBase

문 16. 소프트웨어 개발 생명 주기(Software Development Life Cycle)의 순서로 옳은 것은?

① 계획 → 분석 → 설계 → 구현 → 테스트 → 유지보수

② 분석 → 계획 → 설계 → 구현 → 테스트 → 유지보수

③ 분석 → 계획 → 설계 → 테스트 → 구현 → 유지보수

④ 계획 → 설계 → 분석 → 구현 → 테스트 → 유지보수

정답 1번

간단한 해설

개발의 범위와 일정 등을 계획한 뒤, 분석에서 **무엇을 만들지** 정하고 설계에서 **어떻게 만들지** 구체화한다. 설계를 코드로 구현해야 실제 동작을 테스트할 수 있으며, 사용 중 발견되는 오류나 변경 요구는 유지보수에서 처리한다.

올바른 순서

계획 → 분석 → 설계 → 구현 → 테스트 → 유지보수

계획: 개발 목적과 범위를 정하고 타당성, 비용, 일정, 필요한 인력 등을 검토한다. 개발을 추진할 방향과 자원을 정한 뒤 구체적인 요구사항을 분석한다.

분석: 사용자가 필요로 하는 기능과 제약조건을 파악하여 **무엇을 개발할지** 명확히 한다. 이 요구사항이 있어야 다음 단계에서 이를 만족시키는 구조를 설계할 수 있다.

설계: 분석한 요구사항을 바탕으로 시스템 구조, 데이터베이스, 인터페이스, 처리 방식 등을 구체화하여 **어떻게 구현할지** 정한다. 요구사항을 파악하기 전에 설계를 진행하면 필요한 기능과 구조를 제대로 연결하기 어렵다.

구현: 설계 내용을 프로그래밍 언어로 작성하여 실행 가능한 프로그램으로 만든다. 설계가 구현의 기준이 되며, 구현된 프로그램이 실제 동작을 검사하는 테스트의 대상이 된다.

테스트: 구현된 프로그램을 실행하고 결과를 확인하여 **요구사항 충족 여부와 오류**를 검사한다. 테스트 계획이나 사례는 앞서 준비할 수 있지만, 이 순서에서 말하는 구현 결과의 실행 검사는 프로그램이 만들어진 뒤 수행한다.

유지보수: 테스트를 거쳐 사용 중인 소프트웨어의 오류를 수정하고, 운영 환경이나 사용자 요구의 변화에 맞추어 기능을 변경·개선한다. 개발 결과를 실제로 사용하는 과정에서 계속 관리하는 단계이다.

NoteBase

문 17. 다음은 postfix 수식이다. 이 postfix 수식은 스택을 이용하여 연산을 수행한다. 그리고 ^는 지수함수 연산자이다. 처음 *(곱하기 연산) 계산이 되고 난 후 스택의 top과 top-1에 있는 두 원소는 무엇인가? (단, 보기의 '(top)'은 스택의 top 위치를 나타낸다.)

27 3 3 ^ / 2 3 * -

① (top) 6, 1

② (top) 9, 1

③ (top) 2, 3

④ (top) 2, 7

정답 1번

간단한 해설

나눗셈 결과인 1이 스택에 남은 상태에서 2와 3을 넣고 곱하면, 두 수 대신 6이 맨 위에 저장된다. 따라서 곱셈 직후 **top은 6, top-1은 1**이다.

계산 과정

후위 표기식은 왼쪽부터 읽으며, 피연산자는 스택에 넣고 연산자를 만나면 맨 위의 두 값을 꺼내 계산한 결과를 다시 넣는다. 이때 **먼저 꺼낸 값이 오른쪽 피연산자**이고, 다음에 꺼낸 값이 왼쪽 피연산자이다. 아래 스택 상태는 왼쪽이 바닥, 오른쪽이 top이다.

^ 처리: 27, 3, 3을 차례로 넣으면 스택은 [27, 3, 3]이다. ^를 만나면 위의 3 두 개를 꺼내 $3^3 = 27$을 계산하고 27을 넣는다. 처음 넣었던 27은 그대로 남으므로 스택은 [27, 27]이 된다.

/ 처리: 위의 27 두 개를 꺼내 $27 \div 27 = 1$을 계산하고 1을 넣는다. 이때 스택은 [1]이다.

* 처리: 이어서 2와 3을 넣으면 스택은 [1, 2, 3]이다. *를 만나면 **맨 위의 3과 그 아래의 2만 꺼내** $2 \times 3 = 6$을 계산하고 6을 넣는다. 1은 꺼내지 않았으므로 스택은 [1, 6]이 된다.

곱셈 직후 top에는 6, 바로 아래인 top-1에는 1이 있으므로 ①의 (top) 6, 1과 일치한다. 문제에서 묻는 시점은 곱셈 직후이므로 마지막 - 연산은 아직 수행하지 않는다.

NoteBase

문 18. 빅데이터에 대한 설명으로 옳은 것은?

① 빅데이터는 정형데이터로만 구성되며, 소셜 미디어 데이터는 해당되지 않는다.

② 빅데이터를 구현하기 위한 대표적인 프레임워크는 하둡이 있으며, 하둡의 필수 핵심 구성 요소는 맵리듀스(MapReduce)와 하둡분산파일시스템(Hadoop Distributed File System)이다.

③ 빅데이터 처리과정은 크게 수집 → 저장 → 처리 → 시각화(표현) → 분석 순서대로 수행된다.

④ NoSQL은 관계 데이터 모델을 사용하는 RDBMS 중 하나이다.

정답 2번

간단한 해설

하둡은 **HDFS로 데이터를 분산 저장**하고, **맵리듀스로 분산 병렬 처리**하는 대표적인 빅데이터 프레임워크이다. 두 구성 요소는 각각 저장과 처리 역할을 담당한다.

선지별 해설

① X: 빅데이터에는 **정형·반정형·비정형 데이터**가 모두 포함될 수 있다. 정형데이터는 표처럼 일정한 구조를 갖는 데이터이고, 반정형데이터는 JSON처럼 태그나 키 등의 구조 정보를 갖는 데이터이며, 비정형데이터는 자유로운 형식의 글·사진·동영상 등이다. 따라서 소셜 미디어의 게시글과 사진도 빅데이터의 수집·분석 대상이 된다.

② O: 하둡은 여러 컴퓨터를 묶어 대용량 데이터를 저장하고 처리하는 프레임워크이다. 전통적인 하둡 구조의 핵심인 HDFS는 파일을 블록으로 나누어 여러 컴퓨터에 저장하고 복제하여 장애에 대비한다. 맵리듀스는 작업을 나누어 병렬로 수행한 뒤 그 중간 결과를 모아 처리한다. 즉, **HDFS는 분산 저장, 맵리듀스는 분산 처리**를 담당한다.

③ X: 일반적인 처리 흐름은 수집 → 저장 → 처리 → **분석 → 시각화(표현)**로 설명한다. 수집한 데이터를 저장하고 분석에 적합하도록 정제·가공한 뒤, 분석으로 패턴과 의미를 찾아 그 결과를 그래프나 도표로 표현한다. 선지는 분석과 시각화의 순서를 바꾸었다. 시각화가 탐색적 분석 중에도 활용될 수 있지만, 분석 결과를 표현하는 단계로 설명할 때는 분석 뒤에 둔다.

④ X: NoSQL은 관계형 데이터베이스 관리 시스템인 RDBMS의 한 종류가 아니라 **비관계형 데이터베이스**를 가리킨다. RDBMS는 행과 열로 이루어진 관계형 테이블을 기본으로 하지만, NoSQL은 키-값, 문서, 그래프 등 다양한 데이터 모델을 사용한다. 예를 들어 문서형 데이터베이스는 JSON 형태의 문서 단위로 데이터를 저장할 수 있다.

NoteBase

문 19. 최근 컴퓨팅 환경이 클라우드 환경으로 진화됨에 따라 가상화 기술이 중요한 기술로 부각되고 있다. 이에 대한 설명으로 옳지 않은 것은?

① 하나의 컴퓨터에 2개 이상의 운영체제 운용이 가능하다.

② VM(Virtual Machine)하에서 동작되는 운영체제(Guest OS)는 실 머신에서 동작되는 운영체제보다 효율적이다.

③ 특정 S/W를 여러 OS플랫폼에서 실행할 수 있어 S/W 이식성이 제고된다.

④ VM하에서 동작되는 운영체제(Guest OS)의 명령어는 VM명령어로 시뮬레이션되어 실행된다.

정답 2번

간단한 해설

②의 '실 머신보다 효율적이다'는 설명은 가상화에 따르는 **추가 처리 부담**을 간과한 것이다. 여러 운영체제가 하드웨어를 공유해 자원 활용률은 높일 수 있지만, 개별 게스트 운영체제의 실행 효율이 물리 머신에서 직접 실행할 때보다 높아지는 것은 아니다.

선지별 해설

① O: 가상화 계층인 하이퍼바이저가 CPU·메모리 등의 물리 자원을 나누어 여러 가상 머신에 제공한다. 각 가상 머신에 운영체제를 설치하면 **한 물리 컴퓨터에서 여러 운영체제를 동시에** 운용할 수 있다. 예를 들어 한 컴퓨터에서 Windows와 Linux를 각각의 가상 머신으로 실행할 수 있다.

② X: 게스트 운영체제는 가상화 계층을 통해 자원을 이용하므로 자원 배분, 명령어 처리, 입출력 중개 등에 추가 처리 부담이 생긴다. 따라서 같은 하드웨어에서 같은 작업을 수행할 때 물리 머신에서 직접 실행하는 경우보다 **일반적으로 실행 효율이 낮아질 수 있다**. 가상화로 유휴 자원을 통합하여 전체 자원 활용률을 높이는 것과, 개별 게스트 운영체제의 실행 성능이 높아지는 것은 구별해야 한다.

③ O: 가상 머신이 소프트웨어에 필요한 운영체제와 실행 환경을 제공하면, 호스트 운영체제가 달라도 그 환경을 유지하며 소프트웨어를 실행할 수 있어 이식성이 높아진다. 예를 들어 Windows용 프로그램을 Linux 호스트의 Windows 게스트에서 실행할 수 있다. 이는 프로그램이 모든 운영체제에서 직접 실행된다는 뜻이 아니라 **호환되는 가상 실행 환경을 이용한다**는 뜻이다.

④ O: 가상 머신을 소프트웨어로 모사하는 방식에서는 게스트 운영체제의 명령 동작을 해석하거나 실행 가능한 명령열로 변환하여 구현한다. 선지의 'VM명령어로 시뮬레이션'은 이러한 **가상화 계층에 의한 명령 동작의 모사**를 가리킨다. 다만 모든 가상화 구현이 모든 명령어를 모사하는 것은 아니며, 실제 구현에서는 실행 가능한 명령어를 CPU에서 직접 실행하고 필요한 명령어만 별도로 처리하기도 한다.

NoteBase

문 20. 다음 C프로그램의 실행 결과는?

```
#include <stdio.h>
void change(int *px, int *py, int pc, int pd);
void main(void)
{
    int a=10, b=20, c=30, d=40;
    change(&a, &b, c, d);
    printf("a=%d  b=%d  c=%d  d=%d", a, b, c, d);
}
void change(int *px, int *py, int pc, int pd)
{
    *px = *py + pd;   *py = pc + pd;
     pc = *px + pd;     pd = *px + *py;
}
```

① a=60 b=70 c=50 d=30

② a=60 b=70 c=30 d=40

③ a=10 b=20 c=50 d=30

④ a=10 b=20 c=30 d=40

정답 2번

간단한 해설

px와 py는 a와 b의 주소를 받으므로 ***px와 *py에 대입하면 a와 b가 바뀐다**. 반면 pc와 pd는 c와 d의 값을 복사해 받은 별개의 변수이므로, 함수 안에서 변경해도 c=30, d=40은 유지된다.

NoteBase

계산 과정

C에서는 함수 인자를 값으로 전달한다. change(&a, &b, c, d)를 호출하면 px와 py에는 각각 a와 b의 주소가 복사되고, pc와 pd에는 각각 30과 40이 복사된다. 따라서 *px는 a를, *py는 b를 가리키는 반면, **pc와 pd는 c와 d와 별개의 변수**이다.

첫 대입문 *px = *py + pd: 이때 *py는 b의 초기값 20이고 pd는 40이므로 20 + 40 = 60이다. 이 값을 *px에 저장하여 a가 10에서 60으로 바뀐다. b는 아직 20이다.

둘째 대입문 *py = pc + pd: pc는 30, pd는 40이므로 30 + 40 = 70이다. 이 값을 *py에 저장하여 b가 20에서 70으로 바뀐다.

셋째 대입문 pc = *px + pd: 앞에서 바뀐 a의 값 60을 사용하므로 60 + 40 = 100이 pc에 저장된다. 바뀌는 대상은 함수의 매개변수 pc이며, main의 c는 30으로 유지된다.

넷째 대입문 pd = *px + *py: 바뀐 a와 b의 값을 사용하므로 60 + 70 = 130이 pd에 저장된다. pd만 바뀌므로 main의 d는 40으로 유지된다.

함수 호출이 끝난 뒤 printf가 출력하는 것은 **main의 a, b, c, d**이다. 따라서 출력값은 a=60, b=70, c=30, d=40이며, ②의 값과 일치한다.

NoteBase