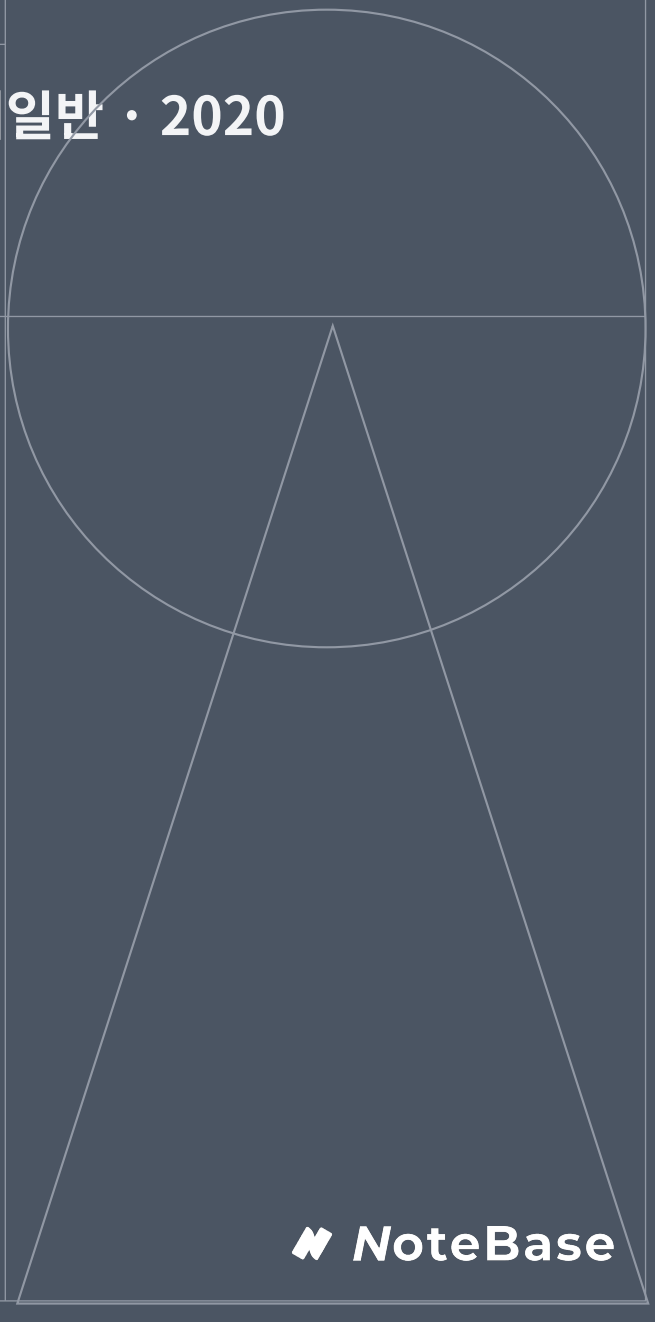
기출OX 해설집

공무원/지방직/9급 · 컴퓨터일반 · 2020



NoteBase

기출OX 해설집

공무원/지방직/9급 · 컴퓨터일반 · 2020

공무원/지방직/9급 · 컴퓨터일반 · 2020

문 1. 인터프리터(Interpreter) 방식의 언어로 옳지 않은 것은?

① JavaScript

② C

③ Basic

④ LISP

정답 2번

간단한 해설

C는 일반적으로 소스 코드를 미리 번역해 실행 파일을 만드는 **컴파일러 방식**의 언어로 분류한다. 인터프리터 방식은 실행 시 명령이나 식을 해석하여 수행하는 방식이다.

선지별 해설

인터프리터는 실행 시 프로그램의 명령이나 식을 해석하여 수행한다. 컴파일러는 소스 코드를 목적 코드 등으로 미리 번역하며, C에서는 보통 컴파일과 링크를 거쳐 실행 파일을 만든다. 이 문항의 구분은 대표적인 구현 방식에 따른 분류이며, 언어마다 실행 방식이 하나로 고정된다는 뜻은 아니다.

① O: JavaScript는 실행 엔진이 코드를 받아 처리하는 **인터프리터 방식의 언어**로 분류한다. 현대의 실행 엔진은 성능을 높이기 위해 실행 중 기계어로 번역하는 JIT 컴파일을 함께 사용하기도 한다.

② X: C는 일반적으로 **컴파일러 방식의 언어**로 분류한다. 소스 코드를 기계어 목적 코드로 번역하고, 필요한 라이브러리 등과 연결하여 만든 실행 파일을 실행하므로, 대표적인 인터프리터 언어로 보는 것은 부적절하다.

③ O: BASIC은 GW-BASIC처럼 **인터프리터로 명령을 해석하여 실행**하는 구현이 널리 사용되어 인터프리터 방식의 대표적인 언어로 분류한다. 다만 BASIC에도 컴파일러를 사용하는 구현이 있다.

④ O: LISP는 인터프리터가 **식을 평가하여 결과를 구하는 방식**으로 실행할 수 있어 인터프리터 방식의 언어로 분류한다. 예를 들어 Emacs Lisp의 인터프리터는 입력된 식을 평가한다. LISP에도 컴파일러를 사용하는 구현이 있으므로 인터프리터로만 실행된다는 뜻은 아니다.

NoteBase

문 2. CPU 스케줄링 기법 중 라운드 로빈(Round Robin) 방식에 대한 설명으로 옳지 않은 것은?

① 선점 스케줄링 기법이다.

② 여러 프로세스에 일정한 시간을 할당한다.

③ 시간할당량이 작으면 문맥 교환수와 오버헤드가 증가한다.

④ FIFO(First-In-First-Out) 방식 대비 높은 처리량을 제공한다.

정답 4번

간단한 해설

라운드 로빈은 CPU를 번갈아 사용하게 하여 응답성을 높이는 방식이며, FIFO보다 **처리량이 높다고 일반화할 수는 없다**. 시간할당량이 작으면 문맥 교환에 드는 시간이 늘어 실제 작업을 처리하는 효율이 낮아질 수 있다.

선지별 해설

① O: 실행 중인 프로세스가 시간할당량을 모두 쓰면 운영체제가 **CPU 사용을 중단시킨다**. 아직 실행할 작업이 남아 있으면 준비 큐의 뒤로 보내고 다음 프로세스에 CPU를 주므로 선점 스케줄링에 해당한다.

② O: 준비 큐의 프로세스들에 같은 크기의 **시간할당량을 차례로 부여**한다. 다만 시간할당량은 한 번에 연속으로 실행할 수 있는 시간의 상한이므로, 그 전에 작업이 끝나거나 입출력을 기다리게 되면 CPU를 내놓는다.

③ O: 실행할 프로세스가 여럿 있을 때 시간할당량을 작게 하면 프로세스들이 더 자주 번갈아 실행되어 문맥 교환수가 증가한다. 문맥 교환에는 실행 상태를 저장하고 다음 프로세스의 상태를 복원하는 작업이 필요하므로, **실제 작업 외에 드는 오버헤드**도 증가한다.

④ X: 처리량은 **단위 시간에 완료한 프로세스 수**이다. FIFO는 먼저 들어온 프로세스가 종료되거나 입출력 등으로 CPU를 내놓을 때까지 실행하는 반면, 라운드 로빈은 시간할당량이 끝나면 다른 프로세스로 전환한다. 이때 추가되는 문맥 교환 비용 때문에 라운드 로빈의 처리량이 더 낮아질 수도 있다. 처리량은 작업 특성과 시간할당량 등에 따라 달라지므로, FIFO보다 높은 처리량을 제공한다고 일반화할 수 없다.

NoteBase

문 3. 프로세서의 수를 늘려도 속도를 개선하는 데 한계가 있다는 주장으로서, 병렬처리 프로세서의 성능 향상의 한계를 지적한 법칙은?

① 무어의 법칙(Moore’s Law)

② 암달의 법칙(Amdahl’s Law)

③ 구스타프슨의 법칙(Gustafson’s Law)

④ 폰노이만 아키텍처(von Neumann Architecture)

정답 2번

간단한 해설

프로세서를 늘려도 **병렬화할 수 없는 순차 실행 부분**은 줄어들지 않는다. 암달의 법칙은 이 부분 때문에 전체 작업의 속도 향상에 한계가 생긴다고 설명한다.

개념 구분

암달의 법칙: **전체 작업량이 고정**되어 있을 때, 작업을 여러 프로세서가 나누어 처리할 수 있는 부분과 순서대로 실행해야 하는 부분으로 구분한다. 프로세서 수를 늘리면 병렬 처리 부분의 시간은 줄일 수 있지만 순차 실행 부분의 시간은 그대로 남는다. 예를 들어 한 프로세서로 100초 걸리는 작업 중 순차 실행에 20초가 필요하다면, 병렬 처리 시간을 이상적으로 거의 없애도 전체 시간은 20초보다 짧아질 수 없다. 따라서 속도 향상은 최대 5배로 제한된다. 이러한 병렬 처리의 한계를 설명하는 것이 암달의 법칙이다.

구스타프슨의 법칙과의 구분: 암달의 법칙이 같은 크기의 작업을 얼마나 빨리 끝낼 수 있는지에 주목한다면, 구스타프슨의 법칙은 프로세서 수가 늘어날 때 **병렬 처리할 작업량도 함께 늘리는 경우**를 다룬다. 같은 실행 시간에 더 큰 문제를 처리할 수 있다는 관점이므로, 고정된 작업에서 순차 실행 부분이 성능 향상을 제한한다는 문항의 설명과 구별된다.

NoteBase

문 4. 교착상태 발생의 조건에 대한 설명으로 옳지 않은 것은?

① 상호 배제 조건: 최소한 하나의 자원이 비공유 모드로 점유되며, 비공유 모드에서는 한 번에 한 프로세스만 해당 자원을 사용할 수 있다.

② 점유와 대기 조건: 프로세스는 최소한 하나의 자원을 점유한 채, 현재 다른 프로세스에 의해 점유된 자원을 추가로 얻기 위해 반드시 대기해야 한다.

③ 비선점 조건: 프로세스에 할당된 자원은 사용이 끝날 때까지 다른 프로세스가 강제로 빼앗을 수 없다.

④ 순환 대기 조건: 대기 체인 내 프로세스들의 집합에서 이전 프로세스는 다음 프로세스가 점유한 자원을 대기하고, 마지막 프로세스는 자원을 대기하지 않아야 한다.

정답 4번

간단한 해설

순환 대기에서는 마지막 프로세스도 **첫 번째 프로세스가 점유한 자원을 기다려야** 대기 관계가 닫힌 고리를 이룬다. ④의 '마지막 프로세스는 자원을 대기하지 않아야 한다'는 설명은 이 조건에 어긋난다.

선지별 해설

교착상태는 프로세스들이 서로에게 필요한 자원을 점유한 채 기다려 더 이상 진행하지 못하는 상태이다.
교착상태가 발생하려면 다음 네 가지 필요조건이 동시에 성립해야 한다.

① O: 상호 배제는 자원을 **한 번에 한 프로세스만 사용할 수 있는** 조건이다. 한 프로세스가 비공유 자원을 점유하면 다른 프로세스는 그 자원을 동시에 사용할 수 없어 기다려야 한다.

② O: 점유와 대기는 프로세스가 **이미 가진 자원을 놓지 않은 채** 다른 프로세스가 점유한 자원을 추가로 요청하여 기다리는 조건이다. 기존 자원을 계속 점유하기 때문에 그 자원이 필요한 다른 프로세스도 기다리게 될 수 있다.

③ O: 비선점은 할당된 자원을 **강제로 회수할 수 없는** 조건이다. 자원을 점유한 프로세스가 사용을 마치고 스스로 반납해야 하므로, 다른 프로세스가 필요하다는 이유로 빼앗아 대기를 해소할 수 없다.

④ X: 순환 대기는 각 프로세스가 다음 프로세스의 자원을 기다리고, **마지막 프로세스도 첫 번째 프로세스의 자원을 기다리는** 조건이다. 따라서 '마지막 프로세스는 자원을 대기하지 않아야 한다'를 '마지막 프로세스는 첫 번째 프로세스가 점유한 자원을 대기해야 한다'로 고쳐야 한다. 예를 들어 P가 자원 A를 가진 채 Q의 자원 B를 기다리고, Q도 B를 가진 채 P의 A를 기다리면 대기 관계가 원을 이룬다. 마지막 프로세스가 아무 자원도 기다리지 않으면 이 대기 고리는 닫히지 않는다.

NoteBase

문 5. CPU(중앙처리장치)의 성능 향상을 위해 한 명령어 사이클 동안 여러 개의 명령어를 동시에 처리할 수 있도록 설계한 CPU구조는?

① 슈퍼스칼라(Superscalar)

② 분기 예측(Branch Prediction)

③ VLIW(Very Long Instruction Word)

④ SIMD(Single Instruction Multiple Data)

정답 1번

간단한 해설

슈퍼스칼라는 여러 실행 장치를 두고, **서로 의존하지 않는 여러 명령어를 한 클록에 병렬로 실행**하여 처리량을 높이는 구조이다. CPU가 실행 중 명령어 사이의 의존성과 실행 장치의 사용 가능 여부를 판단한다.

개념 구분

슈퍼스칼라: CPU 내부에 여러 실행 장치를 두고, 하드웨어가 명령어 사이의 의존성을 확인하여 **동시에 실행할 수 있는 명령어들을 발행**한다. 예를 들어 서로의 결과를 사용하지 않는 덧셈과 곱셈은 필요한 실행 장치가 각각 사용 가능하면 병렬로 실행할 수 있다. 모든 명령어를 무조건 동시에 처리하는 것은 아니다.

VLIW: 여러 연산을 하나의 긴 명령어에 묶어 병렬로 실행한다. 슈퍼스칼라와 달리, 어떤 연산을 함께 실행할지는 주로 **컴파일러가 미리 결정**한다. 문제의 '여러 개의 명령어를 동시에 처리'하는 구조와, 하나의 긴 명령어에 여러 연산을 담는 방식을 구분한다.

SIMD: **하나의 명령어를 여러 데이터에 동시에 적용**한다. 여러 명령어의 병렬 실행을 뜻하는 슈퍼스칼라와는 병렬 처리의 대상이 다르다.

분기 예측: 분기의 실행 방향이나 목적지를 미리 예상하여 다음 명령어를 준비하는 기법이다. 분기 결과를 기다리면서 생기는 지연을 줄이는 데 목적이 있으며, 여러 명령어를 동시에 실행하도록 하는 CPU 구조 자체를 뜻하지는 않는다.

NoteBase

문 6. 캐시기억장치 접근시간이 20 ns, 주기억장치 접근시간이 150 ns, 캐시기억장치 적중률이 80%인 경우에 평균 기억장치 접근시간은? (단, 기억장치는 캐시와 주기억장치로만 구성된다)

① 32 ns

② 46 ns

③ 124 ns

④ 170 ns

정답 2번

간단한 해설

캐시 적중률이 80%이므로 미스율은 20%이다. 이 문항에서는 두 접근시간을 **각 발생 확률로 가중평균**하여, 0.8 × 20 + 0.2 × 150 = 46 ns를 얻는다.

계산 과정

조건과 확률: 캐시 접근시간은 20 ns, 주기억장치 접근시간은 150 ns이다. 캐시 적중은 필요한 데이터가 캐시에 있는 경우이고, 미스는 캐시에 없어 주기억장치에서 읽어야 하는 경우이다. 적중률을 h라 하면 h = 0.8이며, **미스율은 1 − h = 0.2**이다.

평균 계산의 기준: 이 문항에서는 적중 시 접근시간을 20 ns, 미스 시 접근시간을 150 ns로 두고 가중평균한다. 평균 접근시간을 T, 캐시와 주기억장치 접근시간을 각각 T_c, T_m이라 하면, 각 경우의 시간에 발생 확률을 곱해 더하므로 $T = hT_c + (1 - h)T_m$이다.

대입과 중간값: 적중하는 경우가 평균에 기여하는 시간은 0.8 × 20 = 16 ns이고, 미스하는 경우가 기여하는 시간은 0.2 × 150 = 30 ns이다. 이 두 값은 개별 접근에 걸리는 시간이 아니라 **발생 확률을 반영한 평균 기여분**이다.

계산 결과: 평균 기억장치 접근시간은 16 + 30 = 46 ns이며, ②의 46 ns와 일치한다.

NoteBase

문 7. 아날로그 컴퓨터에 대한 설명으로 옳지 않은 것은?

① 입력형식은 부호, 코드화된 숫자, 문자, 기호이다.

② 출력형식은 곡선, 그래프 등이다.

③ 미적분 연산방식을 가지며, 정보처리속도가 빠르다.

④ 증폭회로 등으로 회로 구성을 한다.

정답 1번

간단한 해설

①의 '부호, 코드화된 숫자, 문자, 기호'는 **디지털 컴퓨터의 입력형식**이다. 아날로그 컴퓨터는 전압·전류처럼 **연속적으로 변하는 물리량**으로 정보를 나타내어 입력한다.

선지별 해설

① X: 아날로그 컴퓨터는 **연속적으로 변하는 물리량**을 입력으로 사용한다. 예를 들어 시간에 따라 변하는 온도를 그에 대응하는 전압으로 나타내어 입력할 수 있다. 부호나 코드화된 숫자·문자·기호를 입력하는 것은 디지털 컴퓨터의 특성이다.

② O: 아날로그 컴퓨터의 연산 결과는 연속적인 물리량으로 나타나므로, 이를 시간 등의 기준에 따라 표시하면 **곡선이나 그래프**로 출력할 수 있다.

③ O: 아날로그 컴퓨터는 연속 신호를 이용하여 **미분·적분 연산**을 수행한다. 구성된 회로에서 신호의 변화가 곧 연산으로 이어지므로, 해당 회로로 모형화한 문제를 빠르게 처리할 수 있다. 이는 모든 문제에서 디지털 컴퓨터보다 빠르다는 뜻은 아니다.

④ O: 아날로그 컴퓨터는 **증폭회로 등의 아날로그 회로**를 조합하여 구성한다. 연산증폭기와 저항·축전기 등을 적절히 연결하면 덧셈이나 적분 등의 연산을 수행하는 회로를 만들 수 있다.

NoteBase

문 8. RAID(Redundant Array of Inexpensive Disks)에 대한 설명으로 옳지 않은 것은?

① RAID-0은 디스크 스트라이핑(Disk Striping) 방식으로 중복 저장과 오류 검출 및 교정이 없는 방식이다.

② RAID-1은 디스크 미러링(Disk Mirroring) 방식으로 높은 신뢰도를 갖는다.

③ RAID-4는 데이터를 비트(bit) 단위로 여러 디스크에 분할하여 저장하는 방식이며, 별도의 패리티(parity) 디스크를 사용한다.

④ RAID-5는 별도의 패리티 디스크 대신 모든 디스크에 패리티 정보를 나누어 기록하는 방식이다.

정답 3번

간단한 해설

RAID-4는 데이터를 비트가 아니라 **블록 단위**로 나누어 저장하고, 별도의 패리티 디스크를 사용한다. 따라서 '비트 단위로 분할하여 저장한다'는 부분이 잘못되었다.

선지별 해설

① O: RAID-0은 데이터를 여러 디스크에 나누어 기록하는 스트라이핑 방식이다. 병렬로 읽고 쓸 수 있지만, **복제본이나 패리티를 저장하지 않으므로** RAID 자체의 오류 검출·교정 및 디스크 장애 복구 기능은 없다.

② O: RAID-1은 **동일한 데이터를 다른 디스크에도 복제**하는 미러링 방식이다. 한 디스크가 고장 나도 정상인 디스크의 복제본을 이용할 수 있으므로 신뢰도가 높다.

③ X: RAID-4는 데이터를 **블록 단위로 분할**하여 여러 데이터 디스크에 저장하고, 별도의 전용 디스크에 패리티를 기록한다. 패리티는 데이터 디스크 하나가 고장 났을 때 나머지 데이터와 함께 손실된 데이터를 복구하는 데 쓰이는 정보이다. 따라서 '비트 단위'를 '블록 단위'로 고쳐야 하며, 별도의 패리티 디스크를 사용한다는 부분은 맞다.

④ O: RAID-5는 데이터를 블록 단위로 나누어 저장하면서 **패리티도 여러 디스크에 분산**하여 기록한다. RAID-4처럼 하나의 디스크를 패리티 전용으로 고정하지 않고, 스트라이프마다 패리티를 저장하는 디스크를 달리한다.

NoteBase

문 9. 다음 재귀 함수를 동일한 기능의 반복 함수로 바꿀 때, ㉠과 ㉡에 들어갈 내용을 바르게 연결한 것은?

```
int func (int n) {                  //재귀 함수
     if (n == 0)
          return 1;
     else
          return n * func (n - 1);
}
int iter_func (int n) {             //반복 함수
     int f = 1;
     while ( ㉠ )
          ㉡
     return f;
}
```

① ㉠: n < 0, ㉡: f = f * n--;

② ㉠: n < 0, ㉡: f = f * n++;

③ ㉠: n > 0, ㉡: f = f * n--;

④ ㉠: n > 0, ㉡: f = f * n++;

정답 3번

간단한 해설

재귀 호출에서 n을 1씩 줄이듯, 반복문도 **현재 n을 곱한 뒤 1씩 감소**시켜야 한다. **n > 0인 동안만 반복**하면 0을 곱하지 않고 종료하며, 처음부터 n이 0이면 초기값 1을 그대로 반환한다.

NoteBase

빈칸별 해설

주어진 재귀 함수는 n이 0이면 1을 반환하고, 양수이면 현재 n에 func(n - 1)의 결과를 곱한다. 따라서 0 이상의 정수 n에 대해 팩토리얼을 구하는 구조이다. 반복 함수에서는 f를 1로 시작하여 곱셈 결과를 누적한다.

㉠: n > 0. 곱할 양수가 남아 있는 동안 반복하고, n이 0이 되면 종료해야 한다. **0일 때는 반복문을 실행하지 않아야** f에 0이 곱해지지 않는다. 처음 입력이 0인 경우에도 반복을 건너뛰고 f의 초기값 1을 반환하여 재귀 함수와 같아진다. n < 0을 쓰면 양수 입력에서 반복문을 한 번도 실행하지 않아 필요한 곱셈이 생략된다.

㉡: f = f * n--;. n--는 **감소 전의 n을 곱셈에 사용한 뒤 n을 1 감소**시키는 후위 감소 연산이다. 예를 들어 입력이 3이면 반복할 때마다 (f, n)이 (1, 3) → (3, 2) → (6, 1) → (6, 0)으로 바뀌고, 다음 조건 검사에서 종료하여 6을 반환한다. n++를 쓰면 n이 증가하므로 재귀 호출의 n - 1과 반대로 움직이며 종료 조건인 0에 가까워지지 않는다.

따라서 ㉠은 n > 0, ㉡은 f = f * n--;이며, ③의 조합과 일치한다.

NoteBase

문 10. 데이터의 종류 및 처리에 대한 설명으로 옳지 않은 것은?

① 크롤링(Crawling)을 통해 얻은 웹문서의 텍스트 데이터는 대표적인 정형 데이터(Structured Data)이다.

② XML로 작성된 IoT 센서 데이터는 반정형 데이터(Semi-structured Data)로 분류할 수 있다.

③ 반정형 데이터는 데이터 구조에 대한 메타 데이터(Meta-data)를 포함한다.

④ NoSQL과 Hadoop은 대규모 비정형 데이터(Unstructured Data) 처리에 적합하다.

정답 1번

간단한 해설

크롤링은 웹문서를 수집하는 방법이며, 수집했다는 이유로 데이터가 정형이 되지는 않는다. 웹문서의 자연어 텍스트는 고정된 행·열 구조가 없는 대표적인 **비정형 데이터**이다.

선지별 해설

① X: 정형 데이터는 미리 정해진 스키마에 따라 행과 열 등 일정한 구조로 저장되는 데이터이다. 크롤링으로 얻은 웹문서의 자연어 텍스트는 이러한 고정 구조를 따르지 않으므로 **비정형 데이터**로 분류한다. 크롤링은 수집 방법이고, 정형·비정형은 데이터 구조에 따른 구분이다.

② O: XML은 태그와 계층 관계를 이용해 데이터의 항목과 구조를 표현한다. 예를 들어 센서 측정 시각과 온도를 각각의 태그로 구분할 수 있다. 관계형 테이블처럼 고정된 행·열 스키마를 반드시 따르지는 않지만 **태그로 구조를 나타내므로** 반정형 데이터로 분류할 수 있다.

③ O: 메타 데이터는 데이터의 의미나 구조 등을 설명하는 데이터이다. 반정형 데이터는 값과 함께 **구조를 설명하는 정보**를 포함한다. XML에서 온도 값을 감싸는 태그는 그 값이 어떤 항목에 해당하는지 알려 주는 메타 데이터 역할을 한다.

④ O: NoSQL은 문서형·키-값형 등 유연한 데이터 모델을 활용하여 다양한 구조의 대규모 데이터를 저장·관리하는 데 쓰인다. Hadoop은 여러 컴퓨터에 데이터를 나누어 저장하고 병렬로 처리할 수 있는 기반을 제공한다. 이처럼 두 기술은 **대규모 비정형 데이터의 저장·처리**에 활용하기 적합하다.

NoteBase

문 11. 페이지 부재율(Page Fault Ratio)과 스래싱(Trashing)에 대한 설명으로 옳은 것은?

① 페이지 부재율이 크면 스래싱이 적게 일어난다.

② 페이지 부재율과 스래싱은 전혀 관계가 없다.

③ 스래싱이 많이 발생하면 페이지 부재율이 감소한다.

④ 다중 프로그램의 정도가 높을수록 스래싱이 증가한다.

정답 4번

간단한 해설

같은 크기의 주기억장치에서 동시에 실행하는 프로그램이 늘면 각 프로그램이 사용할 수 있는 메모리가 부족해지기 쉽다. 필요한 페이지를 계속 내보내고 다시 읽어 들이면서 **실제 실행보다 페이지 교체에 더 많은 시간을 쓰는 스래싱**이 발생할 수 있다.

선지별 해설

① X: 페이지 부재율은 전체 페이지 참조 중 필요한 페이지가 주기억장치에 없어 페이지 부재가 발생한 비율이다. 부재율이 높아지면 보조기억장치에서 페이지를 가져오는 일이 잦아지므로, 스래싱이 적게 일어나는 것이 아니라 **발생하기 쉬워진다**.

② X: 스래싱은 필요한 페이지를 주기억장치에 유지하지 못해 페이지 교체를 반복하며, 실제 프로그램 실행보다 페이지 교체에 더 많은 시간을 쓰는 상태이다. 따라서 페이지 부재율과 스래싱은 **밀접한 관계가 있다**.

③ X: 스래싱이 심해지면 곧 다시 사용할 페이지까지 내보냈다가 재참조할 때 다시 읽어 들이는 일이 반복된다. 그러므로 페이지 부재율은 감소하는 것이 아니라 **높은 상태를 보인다**.

④ O: 다중 프로그램의 정도는 메모리에 올려 동시에 실행하는 프로그램의 수를 뜻한다. 주기억장치 용량이 일정할 때 이 수를 지나치게 늘리면 각 프로그램에 필요한 페이지를 유지할 공간이 부족해져 페이지 부재와 교체가 잦아진다. 이처럼 **메모리 수요가 가용 공간을 넘어서면** 스래싱이 심해진다.

NoteBase

문 12. 전자상거래 관련 기술 중 고객의 요구에 맞춰 자재조달에서부터 생산, 판매, 유통에 이르기까지 공급사슬 전체의 기능통합과 최적화를 지향하는 정보시스템은?

① ERP(Enterprise Resource Planning)

② EDI(Electronic Data Interchange)

③ SCM(Supply Chain Management)

④ KMS(Knowledge Management System)

정답 3번

간단한 해설

자재조달부터 생산·판매·유통까지 **공급사슬 전체를 통합하고 최적화**하는 것이 SCM(공급사슬관리)이다. 고객 수요에 맞춰 공급사슬에 참여하는 기업들의 활동을 연계한다.

개념 구분

SCM(공급사슬관리): 공급업체에서 제조업체·유통업체를 거쳐 고객에게 이르는 자재·제품·정보 등의 흐름을 통합적으로 관리한다. 문제의 결정적 단서는 자재조달·생산·판매·유통을 연결하여 **공급사슬 전체의 최적화**를 추구한다는 점이다. 예를 들어 판매 현장의 수요 정보를 생산업체와 공급업체가 공유하면, 예상 판매량에 맞춰 생산량과 자재조달량을 함께 조정할 수 있다.

ERP(전사적 자원관리)와의 구분: ERP는 회계·인사·생산 등 **기업 내부의 업무와 자원을 통합 관리**하는 데 초점을 둔다. SCM은 공급업체·제조업체·유통업체 사이의 연계까지 고려하여 공급사슬 전체를 조정한다. 따라서 이 문제에서는 기업 내부의 자원 통합보다 공급사슬 전반의 기능 통합이라는 범위가 SCM을 식별하는 기준이다.

NoteBase

문 13. 프로토콜과 이에 대응하는 TCP/IP 프로토콜 계층 사이의 연결이 옳지 않은 것은?

① HTTP – 응용 계층

② SMTP – 데이터링크 계층

③ IP – 네트워크 계층

④ UDP – 전송 계층

정답 2번

간단한 해설

SMTP는 전자우편을 보내고 메일 서버 사이에서 전달하는 **응용 계층** 프로토콜이다. 데이터링크 계층은 같은 링크에 연결된 장치 사이에서 프레임을 전달하는 역할을 한다.

선지별 해설

① O: HTTP는 웹 클라이언트와 서버가 요청과 응답을 주고받는 규칙을 정하는 **응용 계층** 프로토콜이다. 웹 문서 등의 자원을 요청하고 전달하는 기능을 담당한다.

② X: SMTP는 전자우편을 보내고 메일 서버 사이에서 전달하는 프로토콜이므로, '데이터링크 계층'을 **'응용 계층'**으로 고쳐야 한다. 데이터링크 계층은 같은 링크에 연결된 장치 사이의 프레임 전달을 담당하며, 전자우편의 송신·전달 규칙을 정하는 계층이 아니다.

③ O: IP는 IP 주소를 이용하여 패킷을 목적지까지 전달하는 **네트워크 계층** 프로토콜이다. 서로 다른 네트워크를 거쳐 패킷이 전달되도록 주소 지정과 경로 선택의 기반을 제공한다. TCP/IP 모형에서는 이 계층을 인터넷 계층이라고도 한다.

④ O: UDP는 포트 번호를 이용하여 호스트에서 실행 중인 응용 프로그램 사이에 데이터를 전달하는 **전송 계층** 프로토콜이다. 연결 설정 없이 데이터그램을 전송하며, 전달 성공이나 도착 순서를 보장하지 않는다.

NoteBase

문 14. 관계 데이터베이스 스키마 STUDENT(SNO, NAME, AGE)에 대하여 다음과 같은 SQL 질의 문장을 사용한다고 할 때, 이 SQL 문장과 동일한 의미의 관계대수식은? (단, STUDENT 스키마에서 밑줄 친 속성은 기본키 속성을, 관계대수식에서 사용하는 관계대수 연산자 기호 π는 프로젝트 연산자를, σ는 셀렉트 연산자를 나타낸다)

<SQL 질의문>

```
SELECT SNO, NAME
FROM STUDENT
WHERE AGE > 20;
```

① $\sigma_{SNO,\ NAME}(\pi_{AGE\ >\ 20}(STUDENT))$

② $\pi_{SNO,\ NAME}(\sigma_{AGE\ >\ 20}(STUDENT))$

③ $\sigma_{AGE\ >\ 20}(\pi_{SNO,\ NAME}(STUDENT))$

④ $\pi_{AGE\ >\ 20}(\sigma_{SNO,\ NAME}(STUDENT))$

정답 2번

간단한 해설

WHERE의 AGE > 20은 **조건에 맞는 행을 고르는 셀렉트**에, SELECT의 SNO, NAME은 **출력할 열을 고르는 프로젝트**에 대응한다. AGE로 행을 먼저 걸러 낸 뒤 SNO와 NAME만 남긴다.

공통 해설

관계대수에서 셀렉트 σ는 조건을 만족하는 행을 고르고, 프로젝트 π는 지정한 속성의 열만 남긴다. 따라서 SQL의 WHERE 조건은 셀렉트의 아래첨자에, SELECT의 속성 목록은 프로젝트의 아래첨자에 쓴다. SQL의 SELECT라는 이름과 관계대수의 셀렉트 연산을 혼동하지 않아야 한다.

이 질의는 STUDENT에서 AGE > 20인 행을 고른 다음, 그 행들의 SNO와 NAME만 출력한다. 관계대수식은 괄호 안쪽부터 적용하므로 $\pi_{SNO,\ NAME}(\sigma_{AGE\ >\ 20}(STUDENT))$로 나타낸다. 안쪽 셀렉트의 결과에는 조건을 만족하는 학생의 SNO, NAME, AGE가 모두 남고, 바깥쪽 프로젝트에서 AGE 열이 제외된다.

핵심은 **조건을 검사할 때 AGE 속성이 남아 있어야 한다**는 점이다. SNO와 NAME만 먼저 프로젝트하면 AGE가 사라져 AGE > 20을 검사할 수 없다. 또한 SNO, NAME은 열의 목록이므로 프로젝트에, AGE > 20은 행의 조건이므로 셀렉트에 붙여야 한다.

NoteBase

문 15. 두 프로토콜 개체 사이에서 흐름제어와 오류제어 및 메시지 전달 등의 기능을 수행하며, 연결성과 비연결성의 두 가지 운용 모드를 제공하는 OSI 참조 모델 계층은?

① 데이터링크 계층(Datalink Layer)

② 네트워크 계층(Network Layer)

③ 전송 계층(Transport Layer)

④ 응용 계층(Application Layer)

정답 3번

간단한 해설

전송 계층은 통신 양 끝의 프로토콜 개체 사이에서 **종단 간 데이터 전달**을 담당하며, 흐름제어와 오류제어 기능을 제공한다. **연결형과 비연결형** 전송 서비스를 모두 지원한다.

핵심 풀이

전송 계층: 송신 측과 수신 측 사이의 **종단 간 데이터 전달**을 담당한다. 흐름제어는 수신 측의 처리 능력을 넘지 않도록 보내는 양이나 속도를 조절하는 기능이며, 오류제어는 전송 중 발생한 오류나 손실을 검출하고 복구하는 기능이다.

두 가지 운용 모드: 연결형은 데이터를 보내기 전에 논리적 연결을 설정하고, 통신이 끝나면 연결을 해제한다. 비연결형은 **사전 연결 설정 없이** 데이터 단위를 전송한다. 전송 계층의 대표 프로토콜로는 연결형인 TCP와 비연결형인 UDP가 있다. 다만 모든 전송 계층 프로토콜이 동일한 제어 기능을 제공하는 것은 아니며, UDP는 TCP와 같은 흐름제어나 재전송을 통한 오류 복구를 제공하지 않는다.

다른 계층과의 구분: 데이터링크 계층도 흐름제어와 오류제어를 수행할 수 있지만, 그 범위는 **하나의 링크로 연결된 인접 노드 사이**이다. 네트워크 계층은 목적지까지 패킷을 보내기 위한 경로 선택과 전달을, 응용 계층은 응용 프로그램이 이용하는 통신 서비스를 담당한다. 따라서 흐름제어·오류제어라는 단어만 보기보다 종단 간 전달과 두 운용 모드를 함께 살펴 전송 계층으로 판단한다.

NoteBase

문 16. 소프트웨어 개발 언어에 대한 설명으로 옳지 않은 것은?

① C#은 마이크로소프트 닷넷 프레임워크를 지원하는 객체지향 언어이다.

② Python은 인터프리터 방식의 객체지향 언어로서 실행시점에 데이터 타입을 결정하는 동적 타이핑 기능을 갖는다.

③ Kotlin은 그래픽 요소를 강화한 게임 개발 전용 언어이다.

④ Java는 컴파일된 프로그램이 JVM상에서 인터프리터 방식으로 실행되는 플랫폼 독립적 프로그래밍 언어이다.

정답 3번

간단한 해설

Kotlin은 **게임 개발 전용 언어가 아니라 범용 프로그래밍 언어**이다. 안드로이드 앱이나 서버 프로그램 등 다양한 소프트웨어를 개발하는 데 사용할 수 있다.

선지별 해설

① O: C#은 마이크로소프트가 개발한 객체지향 프로그래밍 언어로, .NET Framework에서 실행되는 프로그램을 작성하는 데 사용된다. 클래스와 객체를 바탕으로 데이터와 기능을 묶고, 상속과 다형성 등을 활용할 수 있다.

② O: Python은 인터프리터 방식으로 실행되며 객체지향 프로그래밍을 지원한다. **동적 타이핑**은 변수의 자료형을 미리 고정해 선언하지 않고, 실행 중 변수가 참조하는 객체에 따라 자료형이 정해지는 방식이다. 예를 들어 같은 변수에 처음에는 정수를, 나중에는 문자열을 대입할 수 있다.

③ X: Kotlin은 객체지향과 함수형 프로그래밍을 지원하는 **범용 프로그래밍 언어**이다. 안드로이드 앱, 서버 프로그램 등 여러 분야에 사용되므로 '그래픽 요소를 강화한 게임 개발 전용 언어'라는 설명은 잘못되었다. Kotlin/JVM은 Java와 호환되는 바이트코드로 컴파일되며 Java 코드와 함께 사용할 수 있다.

④ O: Java는 소스 코드를 **플랫폼에 독립적인 바이트코드**로 컴파일하고, 이를 각 운영체제에 맞는 JVM에서 실행한다. 따라서 호환되는 JVM이 있는 환경에서는 같은 바이트코드를 실행할 수 있다. 선지의 '인터프리터 방식'은 JVM이 바이트코드를 해석하여 실행하는 기본 구조를 설명한 것이다. 실제 JVM은 실행 성능을 높이기 위해 바이트코드를 기계어로 바꾸는 JIT 컴파일도 함께 사용할 수 있다.

NoteBase

문 17. 소프트웨어 시스템은 기능 관점, 동적 관점 및 정보 관점으로 분류할 수 있다. 동적 관점에서 시스템을 기술할 때 사용할 수 있는 도구로 옳지 않은 것은?

① 사건 추적도(Event Trace Diagram)

② 자료 흐름도(Data Flow Diagram)

③ 상태 변화도(State Transition Diagram)

④ 페트리넷(Petri Net)

정답 2번

간단한 해설

자료 흐름도는 자료가 어디에서 들어와 어떤 처리를 거쳐 어디로 전달되는지를 나타내는 **기능 관점의 도구**이다. 시간에 따른 사건의 발생이나 상태 변화를 표현하는 동적 관점의 도구와 구별한다.

선지별 해설

① O: 사건 추적도는 객체나 시스템 구성 요소 사이에서 주고받는 사건·메시지를 **시간 순서**에 따라 나타낸다. 특정 상황에서 어떤 상호작용이 차례로 일어나는지를 보여 주므로 동적 관점의 기술에 사용한다.

② X: 자료 흐름도는 외부 개체, 처리 과정, 자료 저장소 사이의 자료 이동과 변환을 나타낸다. 이는 시스템이 자료를 받아 어떤 기능을 수행하는지 보여 주는 **기능 관점의 도구**이다. 자료 흐름을 잇는 화살표는 자료의 이동을 뜻하며, 사건의 발생 순서나 시간에 따른 상태 전이를 뜻하지 않는다.

③ O: 상태 변화도는 시스템이 가질 수 있는 상태와 사건에 의해 일어나는 **상태 사이의 전이**를 나타낸다. 예를 들어 대기 상태에서 요청을 받으면 처리 상태로 바뀌는 동작을 표현하므로 동적 관점의 도구이다.

④ O: 페트리넷은 장소, 전이, 토큰을 이용하여 시스템의 동작을 모델링한다. 전이가 실행되면 토큰의 배치가 바뀌며, 이를 통해 여러 작업의 **동시 진행과 동기화**를 표현할 수 있으므로 동적 관점의 기술에 사용한다.

NoteBase

문 18. 다음에서 설명하는 네트워크 데이터 오류 검출 방법은?

송신측: 첫 번째 비트가 1로 시작하는 임의의 n+1비트의 제수를 결정한다. 그리고 전송하고자 하는 데이터 끝에 n비트의 0을 추가한 후 제수로 모듈로-2 연산을 한다. 그러면 n비트의 나머지가 구해지는데 이 나머지가 중복 정보가 된다.
수신측: 계산된 중복 정보를 데이터와 함께 전송하면 수신측에서는 전송받은 정보를 동일한 n+1제수로 모듈로-2 연산을 한다. 나머지가 0이면 오류가 없는 것으로 판단하고, 나머지가 0이 아니면 오류로 간주한다.

① 수직 중복 검사(Vertical Redundancy Check)
② 세로 중복 검사(Longitudinal Redundancy Check)
③ 순환 중복 검사(Cyclic Redundancy Check)
④ 체크섬(Checksum)

정답 3번

간단한 해설

데이터 뒤에 0을 붙인 뒤 제수로 **모듈로-2 나눗셈을 하여 얻은 나머지**를 중복 정보로 사용하는 방식은 순환 중복 검사(CRC)이다. 수신측도 같은 제수로 나누어 나머지가 0인지 검사한다.

핵심 풀이

순환 중복 검사(CRC)는 송신측과 수신측이 같은 제수를 사용하여 **모듈로-2 나눗셈의 나머지**로 전송 오류를 검사하는 방법이다. 지문의 '데이터 끝에 n비트의 0을 추가', '나머지가 중복 정보', '동일한 제수로 검사'가 이 방식을 가리킨다.

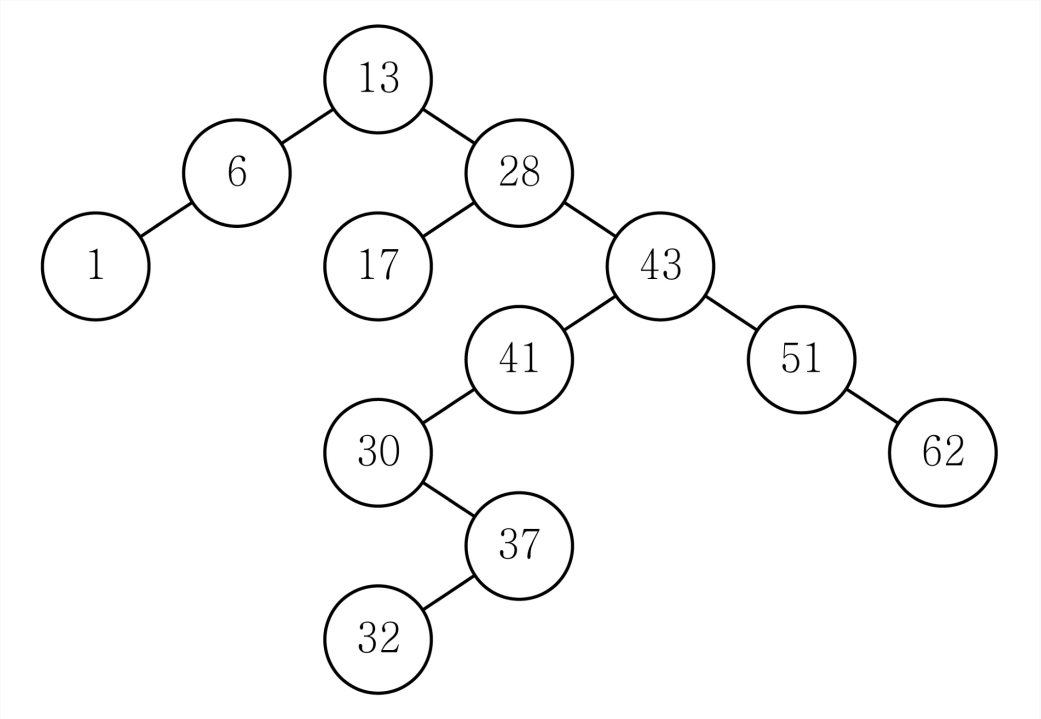
송신측: 첫 비트가 1인 n+1비트 제수를 정하고 데이터 뒤에 n비트의 0을 붙여 나눈다. 이때 뺄셈은 자리 빌림 없이 XOR로 수행한다. 같은 두 비트의 XOR는 0, 다른 두 비트의 XOR는 1이다. 구한 나머지를 n비트로 나타내어 처음 붙였던 0 대신 넣으면, 전송할 전체 비트열을 같은 제수로 나누었을 때 나머지가 0이 되도록 만들 수 있다.

수신측: 데이터와 중복 정보를 합친 전체 비트열을 송신측과 같은 제수로 나눈다. 나머지가 0이 아니면 오류가 검출된 것이다. 나머지가 0이면 오류가 없는 것으로 판단하지만, 이는 **오류가 검출되지 않았다는 뜻**이며 모든 오류가 없음을 보장하지는 않는다.

다른 방식과의 구분: VRC와 LRC는 비트들의 패리티를 검사하는 방식이고, 체크섬은 데이터 단위들을 더해 얻은 검사값을 이용하는 방식이다. 지문처럼 제수로 나누어 나머지를 구하는 절차는 CRC의 특징이다.

NoteBase

문 19. 다음 이진검색트리에서 28을 삭제한 후, 28의 오른쪽 서브트리에 있는 가장 작은 원소로 28을 대치하여 만들어지는 이진검색트리에서 41의 왼쪽 자식 노드는?



① 13

② 17

③ 32

④ 37

정답 4번

간단한 해설

28의 오른쪽 서브트리에서 가장 작은 원소는 **30**이다. 30으로 28을 대치한 뒤 원래 30의 자리를 없애면, 그 오른쪽 자식인 **37이 41의 왼쪽 자식**으로 연결된다.

NoteBase

공통 해설

이진검색트리는 각 노드의 왼쪽 서브트리에 그 노드보다 작은 값을, 오른쪽 서브트리에 큰 값을 둔다. 자식이 둘인 노드를 오른쪽 서브트리의 최솟값으로 대치할 때는 그 값을 옮긴 뒤 **원래 위치의 노드도 제거**한다. 최솟값 노드는 왼쪽 자식이 없으므로, 오른쪽 자식이 있으면 그 자식을 원래 부모에 연결한다.

그림에서 28의 오른쪽 자식은 43이다. 여기서 왼쪽으로 43 → 41 → 30을 따라가면, 30에는 왼쪽 자식이 없으므로 오른쪽 서브트리의 최솟값은 30이다. 따라서 28의 위치에 30을 놓는다.

원래 30은 41의 왼쪽 자식이고, 오른쪽 자식으로 37을 갖는다. 원래 30을 제거하면서 41의 왼쪽 연결을 37로 바꾸므로, 삭제 후 **41의 왼쪽 자식은 37**이다. 32는 37의 왼쪽 자식으로 그대로 남으므로 41에 직접 연결되지 않는다.

NoteBase

문 20. 다음은 리눅스 환경에서 fork() 시스템 호출을 이용하여 자식 프로세스를 생성하는 C 프로그램이다. 출력 결과로 옳은 것은? (단, “pid = fork();” 문장의 수행 결과 자식 프로세스의 생성을 성공하였다고 가정한다)

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<errno.h>
#include<sys/wait.h>

int main(void) {
   int i=0, v=1, n=5;
   pid_t pid;

   pid = fork();

   if( pid < 0 ) {
      for(i=0; i<n; i++) v+=(i+1);
      printf("c = %d ", v);
   } else if( pid == 0 ) {
      for(i=0; i<n; i++) v*=(i+1);
      printf("b = %d ", v);
   } else {
      wait(NULL);
      for(i=0; i<n; i++) v+=1;
      printf("a = %d ", v);
   }
   return 0;
}
```

① b = 120, a = 6

② c = 16, b = 120

③ b = 120, c = 16

④ a = 6, c = 16

정답 1번

간단한 해설

자식은 곱셈을 수행해 b = 120을, 부모는 별도의 v에 1을 다섯 번 더해 a = 6을 출력한다. 부모가 **wait(NULL)로 자식의 종료를 기다리므로** b가 a보다 먼저 출력된다.

NoteBase

계산 과정

프로세스 분기: fork()가 성공하면 자식에서는 pid가 0이고, 부모에서는 pid가 자식의 프로세스 ID인 양수이다. 따라서 자식은 pid == 0인 분기, 부모는 마지막 else 분기를 실행한다. pid < 0인 실패 분기는 실행되지 않으므로 c는 출력되지 않는다. 두 프로세스는 각각 i = 0, v = 1, n = 5에서 시작하며, **한쪽의 변수 변경은 다른 쪽에 반영되지 않는다**.

자식의 계산: 반복 조건이 i < n이고 n = 5이므로 i는 0, 1, 2, 3, 4로 변한다. v *= (i + 1)은 현재 v에 i + 1을 곱해 저장하므로, 1부터 5까지 차례로 곱한다. v는 각 반복 후 1 → 2 → 6 → 24 → 120이 되어 b = 120을 출력한다.

부모의 계산과 대기: 부모는 **wait(NULL)에서 자식의 종료를 기다린 뒤** 반복문을 실행한다. 자식은 b를 출력한 뒤 return 0으로 종료하므로 부모의 a 출력보다 b 출력이 앞선다. 부모의 v는 여전히 1이며, v += 1을 다섯 번 수행하면 2 → 3 → 4 → 5 → 6이 되어 a = 6을 출력한다.

출력 결과는 b = 120 다음에 a = 6이며, ①의 결과와 일치한다.

NoteBase