동시성의 규율

레이스 조건, 로크, 그리고 1인 개발자가 가장 먼저 넘어지는 자리의 기술

ThakiCloud (Hyojung Han)

동시성의 규율

레이스 조건, 로크, 그리고 1인 개발자가 가장 먼저 넘어지는 자리의 기술

ThakiCloud (Hyojung Han)

Contents

1 1. 레이스 조건의 해부학

틀린 이유는 실행 순서다. 단일 스레드 프로그램에서는 실행 순서가 작성한 순서와 동일하다. 1번 줄이 먼저 실행되고 같은 입력이면 같은 결과가 나온다. 이 보장 덕분에 버그를 이해할 수 있다. 동시성을 추가하는 순간 이 보장이 사라진다. 같은 코드가 재실행되면 결과가 달라지고, 에러는 발생하지 않는다. 시스템은 "동작할 때만 작동한다".

이 장은 그런 시스템이 어디서 무너지는지, 그리고 무너지기 전에 그 자리를 어떻게 파악하는지를 다룬다.

1.1 실행 순서가 내 손에서 빠지는 순간

레이스 조건은 두 동작이 서로의 사이에 끼어들었을 때만 나타나는 결함이다. 이 끼어들기를 인터리빙이라고 부른다. 두 스레드, 혹은 두 비동기 태스크의 작업이 번갈아 섞이는 상태다.

가장 간단한 사례부터 본다. 두 스레드가 카운터를 1씩 올린다. 코드는 한 줄이다. 카운터에 1을 더하는 것. 기계는 이 한 줄을 세 단계로 나눈다. 먼저 카운터의 값을 읽는다. 다음에 1을 더한다. 마지막으로 쓴 값으로 덮어쓴다. 스레드 A와 B가 번갈아 실행되면 이렇게 된다. A가 카운터를 읽는다(0). B도 카운터를 읽는다(0). A가 1을 쓴다. B가 1을 쓴다. 두 스레드가 각각 1을 더했는데, 결과는 1이다. 한 번의 증가가 사라진 것이다. 로스트 업데이트라고 부른다.

이 작은 예시에는 몇 가지 순서가 가능한가. 스레드 각각의 3단계를 섞어 놓는 방법은 20가지다(전체 6단계 중 A가 차지할 3칸을 고르면 나머지는 B의 것이다). 그 20가지 중 19가지는 아무 손해를 내지 않고 1가지만 증가를 잃는다. 이 예시는 스레드 2개, 단계 3개다. 실제 코드에서는 단계가

수십이고, 동시에 도는 실행도 스레드 수십 개뿐 아니라 수십 개의 연결로 확장된다. 순서의 개수는 테스트와 리뷰로 커버할 수 없는 규모로 늘어난다. 핵심은 나쁜 순서가 존재한다는 점이다. 나쁜 순서가 하나만이라도 도달 가능하면, 결국 도달한다.

자바스크립트로 같은 이야기를 옮기자. 사용자의 설정을 불러오는 챗봇이다.

```
prefs = {} # 공유 상태

async function loadPrefs(key):
 if key is not in prefs:
 prefs[key] = await fetchPrefs(key) # await: 여기로 다른
 ↪ 태스크가 들어온다
 return prefs[key]
```

같은 key의 요청 두 개가 동시에 들어오면, 둘 다 key가 없음을 보고 둘 다 fetch를 쏘고 늦은 쪽이 덮어쓴다. 결과는 같으니 손해는 호출 한 번뿐이다. fetch가 읽기가 아니라 쓰기(할당, 결제, 예약)라면 이 예시는 뒤에서 다룰 이중 동작이 된다. await가 있는 단일 스레드는 스레드 둘과 본질적으로 다르지 않다. 다만 인터리빙 지점이 await 하나뿐이라는 차이가 있을 뿐이다.

1.2 체크-드 액트의 세 형태

대부분의 무너지는 자리는 하나의 문법을 따른다. 먼저 상태를 확인하고 나중에 행동한다. 확인과 행동 사이에 시간이 있고 그 시간으로 다른 실행이 들어온다. 흔한 형태는 세 가지다.

첫째, 앞서 본 로스트 업데이트다. 읽기, 수정, 쓰기. 읽은 값이 쓰기에 도달하기 전에 다른 값이 끼어들면, 둘 중 하나가 사라진다. 실제 예는 좋아요 수, 조회 수, 사용량 집계 같은 카운터류다. 각 요청이 정확히 1을

더하는데, 집계 결과는 늘 조금씩 모자란다. "왜 3개 없지"라고 물을 때, 답은 대개 여기다.

둘째, 이중 동작이다. "첫 번째일 때만" 확인하고 처리하는 구조다. 결제가 전형적이다. 결제 제공사가 타임아웃으로 웹훅을 두 번 보낸다고 하자. 핸들러는 "이 결제 id에 해당하는 주문이 없나" 확인하고 없으면 주문을 만든다. 두 웹훅이 동시에 처리되면, 둘 다 "없다"를 보고, 둘 다 주문을 만든다. 확인은 각각 정확했다. 원자적이었을 뿐이다.

셋째, TOCTOU, 즉 확인 시점과 사용 시점의 불일치다. 파일이 존재하는지 확인하고 그 파일을 연다고 하자. 확인과 열기 사이에 다른 프로세스가 파일을 지우면, 확인은 참이었고 행동은 실패한다. 더 나쁜 변형도 있다. 확인은 참이었고 행동은 다른 파일에 향한 경우다. 권한 확인 후 열기, 링크 확인 후 따르는 류의 코드에서 이 변형이 사고가 된다.

형태	예	수법
로스트 업데이트	카운터 증가	원자 연산
이중 동작	중복 결제	중복 방지 키
TOCTOU	확인 후 파일 열기	바로 열기

세 형태의 공통 문법을 보자. 확인이라는 동사와 행동이라는 동사, 그리고 그 사이에서 벌어지는 시간이다. 그래서 처방도 하나의 문법이다. 그 시간을 닫거나, 확인을 행동 안으로 들여라. 시간을 닫는 방법은 로크와 트랜잭션이다. 확인을 행동 안으로 드는 방법은, 상태가 바뀌면 실패를 보장하는 기계적 원자 연산이다. compare-and-swap이 대표적이고 데이터베이스에서는 확인을 WHERE 절에 담은 조건부 업데이트가 그렇다.

조건부 업데이트를 구체적으로 쓴다.

```
UPDATE accounts SET balance = balance - 5000
WHERE id = 42 AND balance >= 5000
```

이 한 문장이 “잔액이 충분한가”라는 확인과 “돈을 뺀다”는 행동을 데이터베이스의 원자 단위로 묶는다. 변경된 행이 0개면 확인이 실패했다는 뜻이고 그래서 조작이 일어나지 않았다는 것을 아는 것이다. 확인을 WHERE 절에 넣는 습관은 1인 시스템에서 가장 저평가된 원자성 도구다. 별도 로크 없이, 읽기 후 쓰기 대신 한 문장으로 문제를 닫을 수 있는 경우가 의외로 많다.

1.3 왜 찾기 어렵나

레이스 조건은 재현되지 않아서 어렵다. 인터리빙이 실제로 일어날지는 스케줄러, 부하, 네트워크 지연, 하드웨어 속도에서 정한다. 내 머신에서 100번 돌려서 통과했다는 것은 아무것도 증명하지 못한다. 그 실행에서는 해당 인터리빙이 안 일어났을 뿐이다.

더 나쁜 성질이 있다. 디버깅 자체가 타이밍을 바꾼다. 로그 한 줄을 넣으면 스레드의 속도가 바뀌고 깨지던 인터리빙이 더 이상 안 일어나기도 한다. 이 현상을 하이젠버그라고 부른다. 쫓을수록 도망가는 버그. 흔한 형태를 하나 더 보자. 하루에 몇 개씩 모자라는 사용량 집계다. 크래시는 없고 0으로 사라진 것도 아니라서 버그 리포트에 올라오지조차 않는다. 분석에서 “좀 부족하다”는 느낌을 한 달간 의심받다가, 원인이 카운터 코드의 읽기-쓰기 쌍에서 나온다. 요청이 겹치는 순간마다 하나가 잃어진다. 부하가 낮으면 겹침이 드물어 손실이 소수 퍼센트고 부하가 높으면 몇 퍼센트가 된다. 증상의 크기가 원인의 크기와 안 맞으니, 진단이 오래 걸린다. 규율은 같다. 숫자에서 출발하지 말고, 구조(인터리빙 지점)에서 출발한다.

테스트가 왜 못 잡나. 테스트도 개발자가 쓴 순서대로 인터리빙을 돌리기 때문이다. “A의 2단계 사이에 B가 끼어들면”이라는 경우를 의도적으로 쓰는 사람은 드물다. 그 생각을 하는 것 자체가 리뷰다. 동시성 스트레스 테스트나 연산 순서를 섞는 프로퍼티 테스트를 의도적으로 쓰지 않는 한, 테스트 스위트는 “내가 의도한 순서”만 커버한다. 나쁜 순서는 테스트

스위트에 없고 프로덕션의 스케줄러에 있다.

그래서 레이스 조건의 디버깅 단위는 “어떤 순서”여야 한다.

재현이 아닌 나열로 버그를 확인하는 절차는 세 단계다.

단계 1, 공유 상태를 나열한다. 이름과 위치를 같이. 전역 변수, 데이터베이스의 행, 파일, 캐시 항목.

단계 2, 각 상태에 대해 읽거나 쓰는 연산을 나열한다. 해당 연산이 인터리빙 지점을 포함하는지도 함께. await가 있나, 로크가 없나, 트랜잭션이 있나.

단계 3, 연산 쌍마다 “이 둘이 끼어들면 무슨 일이 생기는가”를 묻는다. 답이 나오면 결함이고 답이 안 나오면(순서가 무관하다면) 해당 쌍은 관리 대상에서 빠진다.

상태	연산 쌍	끼어들면
잔액	확인 후 쓰기	이중 차감
주문	존재 확인 후 생성	중복 주문
파일	확인 후 열기	다른 파일
카운터	읽기 후 쓰기	증가 분실

종이에 나쁜 순서를 쓸 수 있다면, 버그는 이미 확인된 것이다. 기계가 만들어주기를 기다릴 필요가 없다. 이 “재현이 아니라 나열”이 동시성 디버깅의 핵심 기술이다. 테스트는 나열의 결과, 즉 “이 순서는 허용된다”는 주장의 증거로 쓰면 된다. 나열 없이 테스트만 돌리면, 돌리지 않은 순서에 대해서는 아무 증거도 갖지 않게 된다.

1.4 무너지는 자리를 찾는 세 가지 질문

어떤 코드든, 혹은 어떤 프로세스든 세 가지 질문을 던져라.

첫째, 이 코드에서 공유 상태를 어디에서 읽나. 공유라 함은 두 개 이상의 실행이 닿을 수 있는 상태다. 전역 변수, 데이터베이스의 한 행, 파일, 캐시, 모듈 수준 객체. 함수 내부의 지역 변수처럼 한 실행만 쓰는 것은 범위 밖이다.

둘째, 그 읽기에서 행동까지 두 실행이 끼어들 수 있나. 스레드에서는 어디서든 가능하다. 비동기 코드에서는 await 경계에서다. 동기 단일 스레드 함수 안에서는 아니다. 답은 "어디서 가능하느냐"여야 한다. 인터리빙 지점의 위치를 아는 것이 반이다.

셋째, 끼어들면 결과가 어떻게 되나. "아무 일도 안 일어난다"(두 쓰기가 같은 값을 만들거나, 순서가 무관하다)면 무해하다. "한 업데이트가 사라진다", "두 번 처리된다", "낡은 값이 쓰인다"면 무너지는 자리다.

세 질문을 실제 구조 하나에 돌려보자. 재고 수량을 줄이는 웹훅 핸들러다. 첫째, 공유 상태는 데이터베이스의 재고 행이다. 둘째, 끼어들 수 있나. 제공사의 재시도, 다른 요청과의 동시 처리, 야간 동기화 잡. 예, 여러 경로로. 셋째, 끼어들면. 두 개의 "1을 뺀다"가 하나를 잃으면 초과 판매다. 두 개의 "주문을 만든다"가 겹으면 중복 주문이다. 두 자리, 열 분이면 목록이 나온다. 하나는 조건부 업데이트로, 하나는 중복 방지 키로 고친다. 테스트 없이도, 나열만으로.

이 장의 규율을 한 문장으로 압축하면 이렇다. 동시성은 코드를 틀리게 만들지 않는다. 가능한 순서를 늘린다. 프로그래머의 일은 그 늘어난 순서 중 어떤 것이 허용되는지를 정하는 것이다. 허용 순서를 하나만 남기는 방법이 로크다. 그리고 로크가 스스로 만드는 무너지는 자리가 데드락이다. 다음 장에서 그 대가를 치러야 하는 구조를 본다.

2 2. 로크와 그 대가

로크는 인터리빙 지점을 없애는 도구다. 로크를 쥔 동안 다른 실행은 크리티컬 섹션에 들어올 수 없고, 쪼개질 수 있던 순서가 원자적이 된다. 앞 장의 "확인과 행동을 한 단위로 묶어라"를 기계가 대신 강제하는 셈이다. 그러나 로크는 실패를 만든다. 데드락, 성능 저하, 이해하기 어려운 코드가 그 예다. 로크를 쓰는 순간 새로운 공유 상태를 하나 만든다. 바로 "누가 로크를 쥘 것인가"라는 상태. 로크 사용은 규율의 시작이다.

2.1 크리티컬 섹션의 규율

첫 번째 규칙은 크리티컬 섹션을 작게 만드는 것이다. 공유 상태를 직접 건드리는 연산만 넣고 계산과 입출력, 네트워크 호출은 로크 밖으로 뺀다.

로크를 쥔 동안 같은 로크가 필요한 다른 실행은 전부 대기한다. 쥔 시간이 길수록 모든 사람이 기다리는 시간이 길어진다. 로크 안에 데이터베이스 쿼리가 들어가면, 그 쿼리가 끝날 때까지 섹션이 묶인다. 쿼리가 느리면 시스템 전체가 기어간다. 로크는 병렬성을 직렬성으로 바꾸는 장치이고, 그 직렬화의 대가는 로크를 쥔 시간이다.

전형적인 실수를 코드 형태로 쓴다.

```
lock.acquire()
data = api_call() # 네트워크가 로크 안
db.update(data)
lock.release()
```

api_call은 로크 밖으로 뺄 수 있다. 그런데 빼는 순간 새 틈이 생긴다. release와 다시 acquire 사이에 다른 실행이 상태를 바꿀 수 있다. 규칙은 "빼고 난 뒤에도 확인과 행동이 원자적인지 검증하라"다. 검증이 안 되면

계속 안에 둔다. 조금 느린 안전한 섹션이 빠른 깨진 섹션보다 낫다. 이 트레이드오프를 매번 의식해서 선택하는 것이 크리티컬 섹션 규율의 전부다.

두 번째 규칙은 모든 경로가 로크를 내려놓는다는 것이다. 예외가 나는 경로에서도, 에러가 나는 경로에서도, 로크는 release돼야 한다. 내려놓지 않는 코드는 영구 봉쇄를 만든다. try/finally가 있는 언어에서는 finally가 그 보장을 담당한다. "에러 경로도 경로다"라는 문장은 다음 장의 비동기 에러 처리, 4장의 자동화 로그까지 반복될 핵심이다.

2.2 데드락: 네 조건

데드락은 두 개 이상의 스레드가 각각 로크를 쥔 채, 서로가 쥔 로크를 영원히 기다리는 상태다. A가 lock_1을 쥔 채 lock_2를 기다리고 B가 lock_2를 쥔 채 lock_1을 기다린다. 둘 다 진행할 수 없다.

아주 평범한 코드로 만든다. 세션 풀 A에서 B로 아이템을 옮기는 작업이다.

```
// 스레드 A
lock.acquire(L1)
lock.acquire(L2) // B가 쥐고 있으니 대기

// 스레드 B
lock.acquire(L2)
lock.acquire(L1) // A가 쥐고 있으니 대기
```

둘 다 "L1을 먼저, L2를 나중에"도 "L2를 먼저"도 아니다. 각자가 자기 코드 순서대로 얻었을 뿐이다. 순서가 반대인 순간 원이 성립하고 둘은 영원히 앉는다. 어느 쪽이 틀린 것도 없다. 구조가 틀린 것이다.

데드락이 무서운 이유는 크게 소리 높여 죽지 않는 것이다. 크래시가 아니다. 스레드가 그냥 앉아 있다. 메모리는 먹고 CPU는 쓰지 않는다. 지표로는

아무것도 안 보인다. “느려졌다”는 불만만 쌓인다. 그래서 발견될 때면 이미 운영 환경 어딘가에 며칠간 앉아 있는 경우가 많다.

운영에서 “느려졌다”를 만났을 때 첫 점검은 막힌 스레드 덤프다. 자바라면 jstack, 고라면 SIGQUIT 덤프를 쓰는 식이다. monitor entry를 기다리는 상태의 스레드가 여러 개이고, 서로의 로크를 기다리고 있으면 데드락이다. 스택이 정확히 어떤 로크를 기다리는지 말해준다. 덤프를 못 얻는 환경이라면 진단 자체가 불가능하다. 그래서 “우리 운영에서 스레드 덤프를 뽑을 수 있나”는 사고 전에 물어볼 질문이다.

데드락이 일어나려면 네 조건이 동시에 성립해야 한다.

조건	의미
상호 배타	로크는 하나만 쥘 수 있다
보유와 대기	로크 쥔 채 로크 대기
선점 불가	쥔 로크를 빼앗기지 않는다
순환 대기	A가 B, B가 A 대기

네 개 중 가장 쉽게 깨뜨릴 수 있는 것은 순환 대기다. 방법이 로크 오더링이다. 모든 로크에 전역 순위를 정하고 모든 스레드가 같은 순서로만 획득하게 한다. 번호가 낮은 로크를 먼저, 높은 로크를 나중에. 모든 실행이 같은 방향(번호 증가 방향)으로만 기다리면, 원이 만들어질 수 없다. 원이 없으면 데드락이 없다.

구현의 함정은 “주석으로 순서를 정하는 것”이다. “여기서 lock_A를 먼저 얻어라”는 주석은 세 달 후에 아무도 지키지 않는다. 순서는 코드의 구조로 강제해야 한다.

```
with locks_in_order([L1, L2]): // 번호순 정렬 후 획득
 move_item()
```

이처럼 로크 두 개를 함께 얻는 작업을 하나의 래퍼로 묶고, 그 안에서

순서를 고정하면, 호출하는 쪽은 순서를 알 필요도, 실수할 수도 없게 된다. 데이터베이스도 같은 문법이다. 트랜잭션 두 개가 행을 다른 순서로 건드리면 데드락이 일어난다. 거의 항상 두 UPDATE가 서로 다른 순서로 행을 만지는 모양이다. 고치는 방법은 같다. 모든 트랜잭션이 같은 순서(예를 들어 기본키 오름차순)로 행을 만지게 한다.

타임아웃은 탐지 수단이다. 5초를 기다렸는데 못 얻으면 에러를 반환하게 하면, 적어도 "데드락이 아니다"를 알아챌 수 있다. 근본적인 해법은 순환의 가능성을 구조에서 제거하는 것이다. 그리고 가장 싼 제거법은, 한 실행이 동시에 다루는 로크의 개수를 1로 줄이는 것이다. 이 생각은 다음 절로 이어간다.

2.3 입자의 선택: 거칠게 vs 가늘게

로크를 몇 개로 나눌 것인가. 하나가 전부 상태를 덮는 거친 로크는 단순하고 안전하다. 순서 문제 자체가 없다. 대신 모든 접근이 직렬화되어 느리다. 상태를 조각조각 나눠 쥔 가느다란 로크는 빠르다. 대신 데드락과 오더링 문제가 늘고 불변식 검증이 조각 사이로 흩어져 어려워진다.

1인 개발자의 기본값은 거칠어야 한다. 이유는 명확하다. 성능 문제는 나중에 오지만 데드락은 처음부터 올 수 있다. 가늘게 시작해서 부하가 없으면 이득이 없고 위험만 안은 상태다. 거칠게 시작하면 정확성을 확보한 채로 출발하고 나중에 측정해서 가늘게 갈 수 있다. "나중에"는 프로파일링이다.

구체적 사례를 하나 보자. 웹 세션 저장소다. 거친 선택은 저장소 전체에 로크 하나. 모든 요청의 읽기·쓰기가 그 하나에 직렬화된다. 가느다란 선택은 사용자당 로크. 빠르다. 그런데 세션을 사용자에게 옮기는 작업, 만료 세션 정리처럼 "여러 사용자를 한 번에" 건드리는 작업이 오면, 로크 둘 이상을 동시에 쥘 필요가 생긴다. 로크가 둘인 순간, 오더링 규율이 필요해진다. 그래서 순서는 이렇다. 거칠게 시작하고 분할의 트리거는 측정

숫자다. 로크 대기 시간이 요청 전체의 상당 부분(예를 들어 10퍼센트)을 넘어서 프로파일에서 반복 확인되면, 그 때 가장 뜨거운 조각만 분리한다.

가늘게 갈 때는 이전 장의 세 질문을 다시 돌릴 준비가 되어 있어야 한다. 로크 하나를 둘로 쪼개면, 그 하나의 로크가 지키던 "확인-행동 원자성"이 조각 사이로 깨질 수 있다. 분할은 동시성 설계의 변경이고 변경마다 재검증이 따른다.

결정 규칙을 더 단단히 쓴다. 상태가 적고 접근점이 다섯 미만이고 연산이 몇 밀리초 이내라면 거친 로크다. 프로파일링에서 로크 대기가 병목으로 반복 확인되고 각 로크가 지키는 불변식을 이름으로 말할 수 있다면 가늘게 간다. 불변식이 이름으로 안 나오면, 가늘게 갈 준비가 안 된 것이다. 그리고 언제든 먼저 물어야 할 질문이 하나 있다. 이건 싱글 라이터로 만들 수 있나. 된다면 로크를 통째로 피할 수 있는 경우가 많다.

방법	적용	대가
거친 로크	상태 적고 접근점 적음	단순, 느림
가느다란 로크	접근점 많고 병목 실측	빠름, 리스크
싱글 라이터	쓰기가 중앙화 가능	큐 설계

2.4 싱글 라이터: 로크가 필요 없는 구조

가장 싼 동시성 제어는 로크가 필요 없는 구조다. 아이디어는 단순하다. 한 상태에 쓰는 쪽이 하나라면, 쓰기 경쟁 자체가 없으니 로크가 필요 없다. 읽기는 스냅샷이나 원자 읽기로 충분하다.

실제 시스템에는 이 구조가 여러 형태로 있다. 고(Go)의 채널은 상태가 서로를 호출하지 않고 메시지를 보내는 방식이다. 메시지를 받는 쪽, 그 하나만 상태를 건드린다. 자바스크립트는 애초에 단일 스레드라, 모든 상태 변경이 한 이벤트 루프를 지난다. 이것은 구조적 싱글 라이터다.

아웃박스 패턴을 구체적으로 보자. 문제가 먼저 필요하다. 주문 서비스가 데이터베이스에 주문을 쓰고 동시에 큐에 이벤트를 보낸다고 하자. 두 시스템이므로 원자성이 없다. 데이터베이스 쓰기가 성공하고 전송이 실패하면 이벤트가 사라진다. 반대로 전송이 성공하고 쓰기가 실패하면 유령 이벤트가 된다. 아웃박스는 "이벤트 사실"에 대한 싱글 라이터다. 보낼 이벤트를 같은 데이터베이스 트랜잭션 안에 행으로 먼저 적는다(주문과 함께, 원자적). 그리고 프로세스 하나가 그 표를 순서대로 스캔해 큐로 보내고 보낸 기록을 남긴다. 쓰는 쪽이 하나이고 순서가 정해지고 재시도가 안전하다. 로크 없이, 직렬화로 풀린 셈이다.

싱글 라이터는 "동시에 고치는 것을 보호하는" 것 대신, "동시에 고치는 상황이 아예 안 생기게 하는" 것이다. 로크가 만들던 데드락과 오더링 문제가 소거된다. 대신 큐라는 새로운 병목이 생기는데, 그 큐를 다룰 규율이 바로 다음 장이다. 1인 개발자에게는 "쓰는 쪽을 하나로 모을 수 있나"가 로크 설계보다 먼저 던져야 할 질문인 경우가 많다.

2.5 로크 리뷰의 다섯 가지 질문

로크를 쓰는 모든 곳에 다섯 가지 질문을 던진다.

첫째, 크리티컬 섹션이 최소인가. 로크 안에 입출력이나 계산이 들어 있나.

둘째, 로크의 순서가 고정되어 있나. 동시에 두 개 이상의 로크를 얻는 경로가 있나. 있다면 그 순서가 전역적으로 하나인가.

셋째, 모든 경로에서 로크가 내려지는가. 예외 경로 포함.

넷째, 이 상태를 싱글 라이터로 만들 수 있나. 쓰는 쪽을 하나로 모을 수 있으면 로크를 없앨 수 있다.

다섯째, 이 로크를 제거하면 무엇이 깨지나. 답이 나오지 않으면, 그 로크가 지키는 불변식을 아직 모르는 것이다. 불변식을 모르고 지킨다고 믿는

로크는, 모르고 건드린 공유 상태와 똑같이 위험하다.

이 목록은 세 시점에 돌린다. 새로운 로크를 추가하기 전(오더링이 전역 순서를 깬다). "느려졌다"는 불만이 왔을 때(대기가 로크에 쌓이는 중일 수 있다). 그리고 잠을 자기 전(내가 이해하지 못하는 로크를 내일의 나에게 남겨두는 건가).

로크는 경계의 표기다. 어디까지 원자적이어야 하는지, 그 경계의 위치를 정한 흔적이다. 경계의 위치는 설계자의 판단이고 판단은 리뷰로 살핀다. 다음 장에서는 스레드 자체가 없는 시스템으로 간다. 이벤트 루프에서는 인터리빙 지점이 다른 곳에 있고 그래서 무너지는 모양도 달라진다. 로크라는 도구가 아예 없는 세계에서, 원자성은 어떻게 확보되는지 본다.

3 3. 이벤트 루프와 백프레셔

지금까지 여러 스레드가 동시에 도는 상황에 대해 이야기했다. 하지만 시스템의 상당수는 스레드가 하나뿐인데도 수많은 요청을 동시에 처리한다. 웹 서버, 챗봇, 메시지 프로세서, 대부분의 자바스크립트 런타임이 이 경우에 해당한다. 단일 스레드에서 동시성이 가능한 이유는 수행 중인 작업이 여러 개이기 때문이다. 이 장에서는 이런 시스템이 무너지는 두 가지 지점을 다룬다. 이벤트 루프의 인터리빙 지점, 그리고 유입 속도보다 처리 속도가 느릴 때 발생하는 압력인 백프레셔다.

3.1 이벤트 루프는 스케줄러다

이벤트 루프 시스템은 실행 주체가 하나이고 대기열이 존재한다. 네트워크, 디스크, 타이머 등 작업이 끝나면 콜백이 대기열에 들어가고 루프가 순서대로 꺼내어 실행한다. 핵심은 이 루프가 스케줄러라는 점이다. 각 콜백이 언제 실행될지 결정하지만, 실제 실행하는 주체는 하나뿐이다.

그렇다면 인터리빙 지점은 어디인가. 실행이 루프를 넘기는 지점이다. 자바스크립트에서는 await 경계, 파이썬의 asyncio에서는 await, 노드의 콜백 스타일에서는 나머지 작업을 콜백으로 넘기고 함수가 종료되는 자리다. 1장에서 인터리빙 지점을 “아무 실행이든 들어올 수 있는 통로”라고 정의했다. 이벤트 루프에서 이 통로는 정확히 yield와 resume 사이다.

전형적인 무너짐을 하나 보자.

```
cache = null

async function getData():
```

```
 if cache is null:
 cache = await fetch() # yield 지점
 return cache
```

두 태스크가 동시에 들어오면, 둘 다 cache를 null로 인식하여 fetch를 호출하고 후착이 덮어쓴다. 읽기만이라면 손해를 보는 수준이다. 하지만 fetch가 결제, 예약, 할당 같은 쓰기 작업이라면 2장의 이중 동작이 그대로 재현된다. 단일 스레드는 안전하지 않다. 단지 인터리빙 지점이 적을 뿐이다. 규율은 동일하다. 확인과 행동을 원자적으로 만들거나, 쓰는 쪽을 하나로 만들어야 한다.

해결 형태를 보자. 로크가 없는 환경에서 로크 역할을 하는 것은 공유되는 promise다.

```
inflight = {} # key -> promise

async function getData(key):
 if key is not in inflight:
 inflight[key] = fetch(key)
 return await inflight[key]
```

첫 번째 태스크가 fetch를 등록하고 두 번째 태스크는 새 요청을 보내지 않고 같은 promise를 await한다. "이 key의 호출"이라는 자원에 대해 쓰는 쪽, 즉 첫 진입자가 하나인 구조다. inflight는 2장의 싱글 라이터에, await는 1장의 인터리빙 지점에 해당한다. 로크 없이 직렬화한 셈이다.

이벤트 루프에서 "원자적"이라는 뜻은 "yield 없이 끝나는" 것이다. await가 없는 절은 다른 태스크가 끼어들 수 없다. 그래서 처방은 대부분 "크리티컬 섹션 안에서 yield하지 않도록 구조를 고치는 것"이다. 예를 들어 "처리 중" 플래그를 두고 두 번째 태스크는 플래그를 확인하는 순간 진입하지 않게 만든다. 로크가 없어 보이는 곳에서 로크 역할을 하는 것이 이 플래그다.

3.2 조용히 무너지는 자리: 미처리 에러와 순서 가정

스레드 시스템은 에러가 발생하면 크래시로 알린다. 이벤트 루프는 그럴 필요가 없다. 리젝트된 promise, 잡히지 않은 콜백 에러는 아무도 catch하지 않으면 그냥 사라진다. 태스크는 죽었지만 시스템은 계속 돌아가고 상태는 반쯤 손상된 채로 남는다.

규율은 하나다. 모든 비동기 작업은 에러를 catch하는 주인이 있어야 한다. 코드에서는 await를 감싼 try/catch, promise에는 rejection 핸들러가 필요하다. 설계에서는 "fire and forget이 없다"는 규칙이다. 결과를 기다리지 않는 작업은 실패를 알 수도 없는 작업이다. 정말 결과가 필요 없다면, 그 사실을 명시적으로 표기하라. "best effort"라고 로그에 한 줄 쓰는 식이다. 판단의 흔적이 없으면 나중에 실패가 발생했을 때 "왜 안 걸렸나"라는 질문에 답할 수 없다.

순서 가정도 같은 부류다. "A가 B보다 먼저 도착한다"는 가정이 아니라 약속이 필요한 말이다. 네트워크 응답, 타이머, 콜백은 어떤 순서로 와도 된다. 코드가 도착 순서에 의존하면, 그 의존을 명시적으로 처리하는 장치가 필요하다. 시퀀스 번호, 버전 비교, "더 새로운 값이 올 때까지 무시" 같은 규칙이다. 의존을 없애는 것이 더 어려우면, 의존을 데이터(시퀀스)로 바꿔야 한다.

가장 흔한 순서 가정은 "응답이 타임아웃보다 먼저 온다"는 것이다. 클라이언트가 5초 타임아웃을 걸고 서버가 5.1초에 응답을 보낸다고 하자. 응답과 타임아웃 둘 다 도착한다. 둘 중 무엇이 진실인가. 코드가 먼저 온 것을 기준으로 하면, 타임아웃으로 재시도한 결과와 원래 응답의 결과가 다를 수 있다. 그래서 규칙은, 응답에 시퀀스 번호나 요청 id를 붙이고 코드가 "먼저 온 것이 진실, 나머지는 폐기"를 명시적으로 결정하는 것이다. 가정이 타이밍에 있으면 보이지 않고 데이터에 있으면 리뷰된다.

3.3 백프레셔: 들어오는 속도보다 느릴 때

백프레셔는 생산 속도가 처리 속도를 초과하는 상태의 이름이다. 그 차이는 큐에 쌓인다. 큐는 두 형태를 가진다. 유한한 큐는 차면 결정을 요구하고 무한한 큐는 커진다. 메모리가 끝날 때까지. OOM(메모리 부족 강제 종료)은 무한 큐의 최종 형태다. 큐 로직의 실패가 아니라 운영체제의 살해가 원인이 되어 디버깅은 더 어려워진다.

숫자를 하나 세워보자. API가 초당 100개의 요청을 받고 핸들러는 초당 20개를 처리한다. 차이는 초당 80개다. 10초 뒤 큐에는 800개가, 1분 뒤에는 4800개가 쌓여 있다. 큐가 메모리 안에 있다면 이 숫자는 곧 메모리 사용의 예시다. 한 요청이 10KB의 상태를 갖고 있다면 4800개는 약 48MB다. 적어 보이지만 이것은 큐가 비어 있을 때의 값이다. 실제 시스템은 대기하는 동안 상태가 불어날 수 있다. 재시도, 타임아웃, 부분 결과의 보존 등이 그 예다. 방향만 봐도 답이 나온다. 도착이 처리보다 빠르면 깊이는 시간과 함께 선형으로 늘고, 선형 증가는 멈추지 않는다는 뜻이다.

1인 개발자가 놓치는 무너짐의 한 유형은 "큐를 만든 줄도 모르고 만드는 것"이다. 자기가 만든 것뿐 아니라, 쓰는 것에도 큐가 있다. HTTP 클라이언트의 연결 풀(다운스트림이 느리면 요청이 클라이언트 안에서 기다린다), 메시지 브로커의 전송 버퍼(소비자가 죽으면 커진다), 데이터베이스의 레플리케이션 지연(쓰기는 빨랐는데 읽기는 낡은 데이터를 본다), 외부 API의 레이트 리미트(큐는 남의 것이지만, 차면 내 요청이 버려진다). 전부 "요청이 자원을 기다리는 자리"다. 1인 시스템은 이 모든 큐를 마스터할 수는 없어도, 어디에 있는지는 알아야 한다.

측정이 먼저다. 세 숫자. 도착률(초당 몇 개 들어오나), 처리율(초당 몇 개 해내나), 큐 깊이(지금 몇 개 쌓여 있나). 도착이 처리를 이기면 깊이는 늘고 그 반대면 줄어든다. 큐 깊이의 방향이 곧 시스템의 상태다. 리틀의 법칙이 이 관계를 공식으로 준다. 시스템 내 평균 개수는 도착률과 체류 시간의 곱과

같다는 뜻이다. 작업 부하의 세부에 따라 비례 상수는 달라지지만, 방향은 이 곱이 정한다. 즉 큐가 얼마나 깊어질지 알고 싶으면 두 숫자를 곱하면 된다. 실행하지 않고도 알 수 있다.

감시 지표로 가장 중요한 것은 큐 깊이다. CPU 사용률과 응답 시간은 사실을 한 박자 늦게 말해준다. 큐 깊이는 "여기가 병목이다"를 가장 먼저 말해주는 숫자다.

3.4 오버플로우의 4가지 정책

큐가 찼을 때의 정책은 네 가지다.

대기. 생산 쪽으로 압력을 되돌려 속도를 늦춘다. HTTP의 429 응답, TCP의 윈도우, 가득 찬 채널에서 보내는 쪽이 막히는 것. 전부 이 정책이다. 잃는 것이 없으므로 가장 깨끗하다. 대가는 지연이 전체 상류로 전파된다는 점이다. 생산자가 사용자라면, 사용자는 "느리다"를 경험한다.

드롭. 버린다. 가장 싼 정책이다. "일부를 놓쳐도 된다"는 경우에만 쓸 수 있다. 로그, 메트릭, 분석 이벤트, 하트비트 등이 해당한다. 그런데 드롭에도 규율이 있다. 의도적으로 드롭하는 것과, 큐가 차서 우연히 잃는 것은 다르다. 전자만 정책이고 후자는 버그다. 코드에서 둘을 가르는 것은 드롭한 개수를 세는 카운터의 유무다. 버린 것을 아는 시스템과 모르는 시스템은 장애 대응에서 완전히 다른 등급이다.

샘플링. N개 중 1개를 남긴다. 대기와 드롭의 중간이다. 깊이가 너무 커서 전부를 처리할 수 없으면, 전체의 모습을 아는 것이 목표가 아닌 경우가 많다. 10퍼센트 샘플이 추세를 말해준다. 단, 샘플링된 지표는 "비율과 방향"으로만 읽어야 한다.

거부. 보내는 쪽에 "안 된다"고 답한다. 드롭과 다른 것은 답장을 보낸다는 점이다. 429나 "full" 응답이 이 형태다. 보내는 쪽은 자기 작업이 들어가지 않았다는 것을 알 수 있으므로 재시도를 결정할 수 있다. 드롭은 침묵이고

거부는 대화다. API에서는 거부 표준이다. 호출자는 작업이 성립했는지 보통 알아야 하므로.

정책	뜻	대가
대기	생산자 늦춤	지연 전파
드롭	버림	조용한 손실
샘플링	N 중 1	부분 시야
거부	가득함 답장	재시도 부담

어느 것을 골라야 하는가. 큐의 의미를 묻는 것부터다. 반드시 해야 할 일(주문, 결제, 사용자 데이터)이면 대기가거나 거부가다. 잃어도 되는 것(로그, 메트릭, 하트비트)이면 카운터 달린 드롭이다. 추세만 필요한 것(분석)이면 샘플링이다. 서비스의 경계(공개 API)면 거부다. 호출자가 결과를 알아야 하므로. "빠른 것"을 고르는 게 아니라 "이 큐는 무엇을 의미하는가"를 고르는 것이다.

숫자를 한 번만 더 붙이자. 유한 큐의 크기는 두 수의 곱이다. 하나를 처리하는 데 걸리는 시간, 그리고 사용자가 견딜 수 있는 대기 시간. 처리가 200밀리초이고 견딜 수 있는 대기가 2초라면 큐는 10개다. 10을 넘으면 더 기다리는 것이 아니라 정책의 영역이다. 드롭 정책에는 드롭한 개수를 세는 카운터가, 거부 정책에는 호출자 쪽의 백오프 규칙(얼마나 기다리고 재시도할지)이 함께 간다. 이 숫자들이 코드나 설정의 한 자리에 모여 있으면 좋겠다. 숫자가 어디에 있는지 가리킬 수 없다면, 숫자는 존재하지 않는 것이다.

정책 선택은 설계 결정이다. "차면 어떻게 하느냐"는 질문에 다이어그램을 그리는 단계에서 답해야 하는 이유다. 답이 없으면 그 시스템의 한계를 아직 생각하지 않은 것이기 때문이다.

3.5 5단계 검증

비동기 시스템은 5단계로 검증한다.

첫째, 모든 큐가 유한한가. 채널, 버퍼, pending 목록을 전부 찾고 숫자가 있는지 확인한다.

둘째, 오버플로우 정책이 정해진가. 대기, 드롭, 샘플링, 거부 중 어느 것인지.

셋째, 드롭이 카운트되는가. 버리기로 결정했다면, 몇 개를 버렸는지 아는 지표가 있는가.

넷째, 생산자가 소비자의 상태를 아는가. 압력이 되돌아갈 경로가 있는가.

다섯째, 큐 깊이가 감시되는가. 장애가 나면 가장 먼저 보는 숫자인가.

이 검증을 두 시점에 돌린다. 비동기 구성 요소를 추가하기 전(큐가 새로 생기는 순간이다). 그리고 "메모리가 늘어난다"는 증상이 왔을 때(대답은 대개 이 다섯 문항 안에 있다).

이벤트 루프 시스템은 "스레드를 늘리자"라는 도피처가 없다. 실행자가 하나이고 큐가 하나다. 오버플로우는 정책으로 처리한다. 그리고 정책은 큐가 차기 전에 적는 것이지, 차는 동안 즉흥적으로 고르는 것이 아니다. 다음 장에서는 코드를 떠난다. 1인 개발자의 시스템은 구조가 그대로다. 실행자가 하나이고 끝내지 못한 일의 큐가 무한에 가깝고 중복이 없다. 무너지는 자리의 법칙은 같으므로, 규율도 같은 데서부터 시작해야 한다.

4 4. 1인 개발자의 병목 구조

앞 장들은 코드 안의 동시성을 다뤘다. 1인 개발자에게 동시성은 스레드 문제만이 아니다. 1인 운영 자체가 동시성 시스템이다. 여러 작업이 동시에 도착하고 실행자는 하나이며 중복이 없고 터지면 대신할 사람이 없다. 무너지는 법칙은 코드와 정확히 같다. 이 장은 앞 장의 규율을 1인 시스템의 구조로 번역하고, 리뷰를 위한 체크리스트를 낸다.

4.1 1인은 중복 없는 아키텍처다

시스템 공학에서 한 요소의 실패가 전체를 멈추게 하는 구성을 단일 실패 점이라고 부른다. 1인 팀에는 그런 요소가 정확히 하나 있고, 그 요소는 사람이다. 수면, 병, 미팅, 급한 용건은 전부 프로세스 중단이다. 중단된 동안, 끝내지 못한 일의 큐는 계속 늘어난다.

구조가 원인이다. 단일 실행자 시스템은 직렬화를 피할 수 없다. 1인 시스템이 결정해야 하는 것은 작업량이지, 순서와 거부의 시기가 아니다. 3장의 큐 규율이 그대로 적용된다. 일이 처리 속도로 도착을 이기면, 차면 할 정책을 정해둬야 한다. 대기, 드롭, 샘플링, 거부. 코드와 다른 점은 큐가 눈에 보인다는 것이다. 작업 보드, 할 일 목록. 보지 못하게 하면 큐는 머릿속에 있고, 깊이를 재지 못한다.

1인 개발자가 멀티스레드 시스템과 같은 자리에 넘어지는 이유는 구조가 같아서다. 공유 상태(하나의 머리, 하나의 일정), 인터리빙 지점(중단, 메시지, 미팅), 체크-드 액트(확인한 뒤 계획이 바뀌어 행동), 한계 없는 큐(머릿속). 차이가 있다면 규모다. 코드의 인터리빙은 밀리세컨드이고 사람의 인터리빙은 일 단위다. 법칙은 규모를 묻지 않는다.

1인 시스템의 첫 번째 규율은 상태의 외부화다. 캐시는 빠르지만 사라질

수 있고 영속 저장소는 느리지만 남는다. 기억은 캐시이고 보드는 영속 저장소다. 기록되지 않은 일은 존재하지 않는다. 코드에서 로그와 데이터베이스에 없는 것은 사실이 아님을 강조하는 원칙과 같다.

뭘 남길까. 네 가지다. 첫 번째, 결정. 왜 이 경로를 골랐고 무엇을 거절했나. 두 번째, 미완 상태. 어떤 작업이 어디까지 왔고 다음 단위가 무엇인가. 세 번째, 불변식. 이 시스템에서는 절대 일어나지 말아야 할 것. 같은 결제에 두 개의 주문, 큐 깊이가 1000을 넘는 것. 네 번째, 자동화의 상태. 무엇이 몇 시에 도는지, 지난번에 무엇을 썼는지, 실패한 적은 있는가. 첫 번째와 두 번째는 내일의 나를 위한 것이고, 세 번째와 네 번째는 비상 대응의 나를 위한 것이다. 1인 시스템의 장애 대응은 두 주 뒤, 내가 쓴 로그를 이해하지 못한 채 읽는 나다. 로그의 품질은 오늘 쓰는 것이 정한다.

4.2 컨텍스트 스위칭의 진짜 대가

1인 개발자의 가장 흔한 무너짐은 WIP, 즉 동시에 진행 중인 작업의 개수다. 두세 개의 프로젝트를 동시에 진행 중으로 두는 것. 매번 전환에는 대가가 있다. 컨텍스트를 다시 읽고 환경을 다시 올리고 상태를 다시 만든다. 이 대가는 작지 않고 전환한 시간과 비례하지도 않다. 컨텍스트가 클수록 전환이 비싸다.

3장의 리틀의 법칙이 여기서도 일한다. WIP가 많을수록 각 작업의 리드타임이 길어진다. 큐가 길어지기 때문이다. WIP를 1로 줄이면 남은 작업의 리드타임이 준다. 여기서 1은 완결 작업 단위 하나를 뜻한다. 검증 가능한 결과를 만들어내는 단위. 테스트를 통과하는 커밋, 배포 가능한 기능, 보낼 수 있는 문서. 두 일의 반은 두 개의 긴 큐 항목이다.

구체적 루틴을 하나 제시한다. 아침에 보드에서 완결 단위 하나를 고르고 그 단위의 완료 정의를 쓴다. '로그인 기능을 구현한다'가 아니라 '로그인 엔드포인트가 정상 경로 200, 실패 경로 401을 반환하고 테스트가 통과한다'까지다. 완료 정의는 한 줄의 명령으로, 또는 한 번의 클릭으로

확인되는 형태여야 한다. 끝내면 결과를 기록하고, 무엇이 바뀌었는지, 무엇이 남은지, 다음으로 간다. 규율은 결과를 기록한다는 단계에 있다. 그 단계가 재실행, 즉 중단과 잊음을 싼 것으로 만든다.

하루의 구조를 이 책의 어휘로 하나 세워보자. 아침, 보드에서 완결 단위 하나를 고르고 완료 정의를 쓴다. 큐 깊이를 1로 선언하는 순간. 작업 중, 한가운데서 떨어지면 다음 단위를 남긴다. 상태의 외부화. 밤에 자동화가 돌면, 그 로그가 멱등인지, 두 번 돌면 무슨 일이 생기는지에 답할 수 있는지 확인한다. 안 보는 프로세스의 동시성 리뷰. 저녁, 한 줄만 남긴다. 무엇이 바뀌었고 무엇이 남았고 어떤 불변식을 전제하고 있는가. 네 번의 기록, 10분도 안 걸린다. 내일의 내가 오늘의 나를 재탐색하지 않고 이어서 하는 것은 이 네 번이 만든다.

느리게 느껴진다. 실제로 한 번에 하나만 하면 단위당 시간은 늘어난다. 하지만 큐 깊이가 1이므로, 어느 작업이 언제 끝나는지가 처음으로 예측 가능해진다. 예측 가능성이 사라진 1인 시스템은, 인터리빙이 나열되지 않은 코드와 같은 등급이다.

4.3 멱등성과 재실행: 자동화는 동시성의 한 축이다

1인 시스템은 항상 재실행한다. 잊고 끼어들고 틀리고 며칠 뒤 돌아온다. 작업이 재실행되어도 안전하면, 중단 비용은 크게 떨어진다. 이것이 멱등성이다. 같은 작업을 몇 번이나 해도 결과는 한 번과 같다.

코드에서는 상태 머신과 고유 키다. '하나 더한다' 대신 '이 값으로 설정한다'. '새로운 행을 넣는다' 대신 '고유 ID로 업서트한다'. 사람의 일에서는 기록이다. 무엇을 했고 무엇이 남았는지의 로그가 있으면, 다시 시작하지 않고 이어서 할 수 있다.

구체적 사례를 하나 보자. 밤 3시에 도는 리포트 잡이다. 중간에서 실패하고 아침에 다시 돌렸다고 하자. 잡이 어제의 통계를 대시보드에 전송한다면,

재실행은 대시보드에 두 번의 막대를 만든다. 멱등한 형태는 실행에 고유 식별자(날짜, 잡 이름)를 부여하고 대시보드를 그 식별자로 업서트하게 하는 것이다. 재실행은 덮어쓰기가 된다. 어떤 자동화든 같은 문법이다. 조작에 신원을 부여하고 저장 쪽을 설정으로 만든다.

자동화 관점에서 보면, cron 잡, CI 파이프라인, 밤에 도는 스크립트는 전부 내가 타이밍을 정하지 않는 프로세스다. 동시성의 한 축이다. 두 자동화가 같은 자원을 건드리면, 그것들은 레이스를 한다. 규율은 같다. 작업을 멱등하게 만들고 잠금이나 싱글 라이터를 두고 로그를 남긴다. 1인 시스템이 가장 약해지는 순간은 내가 없는 동안 자동화가 돌았을 때다. 보는 사람이 없으니까. 로그가 유일한 증인이다. 자동화 한 개를 추가할 때마다 묻는 질문은 하나다. 이 작업이 두 번 돌면, 혹은 다른 작업과 겹쳐서 돌면, 무슨 일이 생기는가.

1인 시스템의 전형적 무너짐은 세 가지다. 첫 번째, 쓰기 도중 중단. 반쯤 고친 파일, 기억나지 않는 상태. 두 번째, 상태 망각. 한 것 같은데 안 한 것. 세 번째, 부재 중 자동화. 내가 알지 못하는 변경. 셋의 처방은 같다. 외부 기록. 첫 번째에는 보드, 두 번째에는 로그, 세 번째에는 실행 기록. 1인 시스템의 디버깅 단위는 사람의 일 단위 인터리빙이다. 나열은 기록 위에서 한다.

4.4 설계 리뷰의 10문항

앞 네 장의 규율을 10문항으로 모은다. 각각 한 문장 질문이고 순서대로 훑으면 10분면 리뷰가 끝난다. 코드에 쓰고 일의 구조에도 쓴다.

첫째, 공유 상태가 보이는가. 한 파일, 한 함수로 가리킬 수 있나.

둘째, 크리티컬 섹션이 최소인가. 로크 안에 입출력이나 계산이 있나.

셋째, 로크의 순서가 고정되어 있나. 두 개 이상의 로크를 얻는 경로가 있나.

넷째, 모든 큐가 유한한가. 숫자가 있나.

다섯째, 오버플로우 정책이 정해진가. 대기, 드롭, 샘플링, 거부 중 어느 것이나.

여섯째, 작업이 멱등한가. 재실행해도 안전한가.

일곱째, 에러 경로가 성공 경로와 같은 대가를 치르는가. 자원이 내려지는가.

여덟째, 지표가 있는가. 큐 깊이와 실패율을 알 수 있나.

아홉째, 인터리빙 지점이 기록되어 있나. 어디에서 순서가 쪼개질 수 있는지.

열째, 이 요소를 제거하면 무엇이 깨지나. 모든 요소에 답이 나오나.

이 목록을 돌리는 시점은 세 가지다. 새로운 구성 요소(큐, 로크, 자동화)를 추가하기 전. 느려졌다, 메모리가 늘었다는 증상이 왔을 때. 그리고 작업을 넘기기 전. 내일의 나에게, 이해할 수 있는 상태를 남기는가. 마지막 문항이 이해를 확인하는 질문이다. 요소를 제거하면 무엇이 깨지는지 답하지 못하면, 그 요소를 아직 이해하지 못한 것이다. 로크든, 큐든, 정책이든, 사람의 일든 같다.

4.5 무너짐의 순서

시스템은 순서대로 무너진다. 먼저 동작하지 않는다, 다음 느리다, 마지막으로 가끔만 동작한다. 1인 개발자는 보통 첫 번째에 시간을 다 쓰거나, 세 번째를 경험하고서야 뒤늦게 두 번째를 고친다. 그 결과 동작할 때만 작동하는 시스템이 남는다.

규율의 부재가 원인이다. 로크의 순서가 정해지지 않았고 큐에 한계가 없었으며 오버플로우 정책이 없었고 상태가 외부화되지 않았다. 이 네 가지는 코드에서도, 1인의 하루에서도 같은 모양으로 나타난다.

1인 개발자의 이점은 판단하는 사람이 한 명이라는 것이다. 그 판단이

가시적이라면, 시스템은 읽을 수 있다. 로크의 순서가 코드 구조에 있고 큐의 정책이 다이어그램에 있으며 일의 순서가 보드에 있고, 자동화의 흔적이 로그에 있다. 어디에 무엇을 남길 것인가. 그게 앞 장들의 내용 전체다.

규율은 추가 비용이 아니다. 무너지기 전에 실패 방식을 가시화하는 행위다. 이유 없이 무너지는 시스템은, 그 이유가 쓰이지 않은 시스템이다. 쓰면 무너짐이 예측 가능해지고 예측 가능해지면 방지할 수 있다. 동시성의 규율은 결국 이 한 문장이다. 가능한 순서를 줄여라, 그리고 줄이지 않은 순서를 가시화해라.