

“모니터링은 정상인데 OOM?”

Ubuntu 환경에서 eBPF로 추적하는 PyTorch 유령 메모리 누수

UbuCon Korea 2026 | 박민진



UbuCon x
MiniDebConf Korea 2026



박민진

Modernization Solutions Architect
@MegazoneCloud
Microsoft Certified Trainer

Training & Speaking

Delivering DevOps(Azure, GitHub) and AI(GitHub Copilot)
@Hyundai AutoEver, Hanwha Vision, KT, MS's US partner Corp etc.

Korea MCT Summit 2025 & 2026 / DDD Seoul 2024 / .NET Conf 2024 @MS Korea

I ♥ Open Source

Contributing Azure OpenAI SDK Proxy etc.

Aya(Rust eBPF) based K8s Selective LLM Observability Tool Co-Developer
(mentioned by eBPF Foundation Chair)



AI 앱(Pod)이 갑자기 죽었어요!



원인은 OOMKilled (Exit Code 137)

```
kubectl describe pod oom-demo
```

State: Terminated

Reason: OOMKilled

Exit Code: 137

Limits:

memory: 50Mi

Requests:

memory: 50Mi

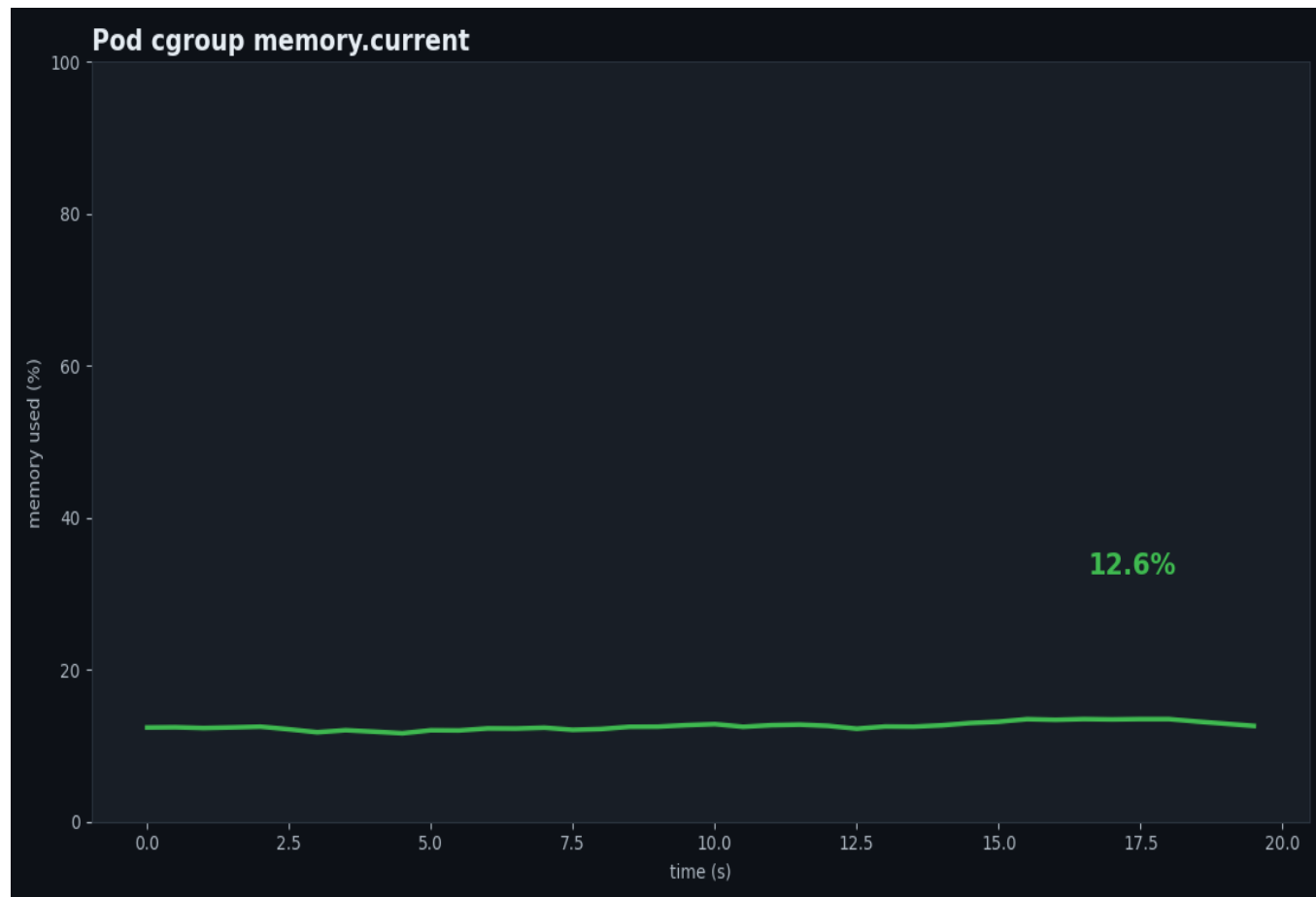
※ 참고:

Exit Code 137 - 프로세스가 SIGKILL(9)를 받아 강제 종료.
Exit Code 137의 주된 원인은 메모리 부족(OOM)

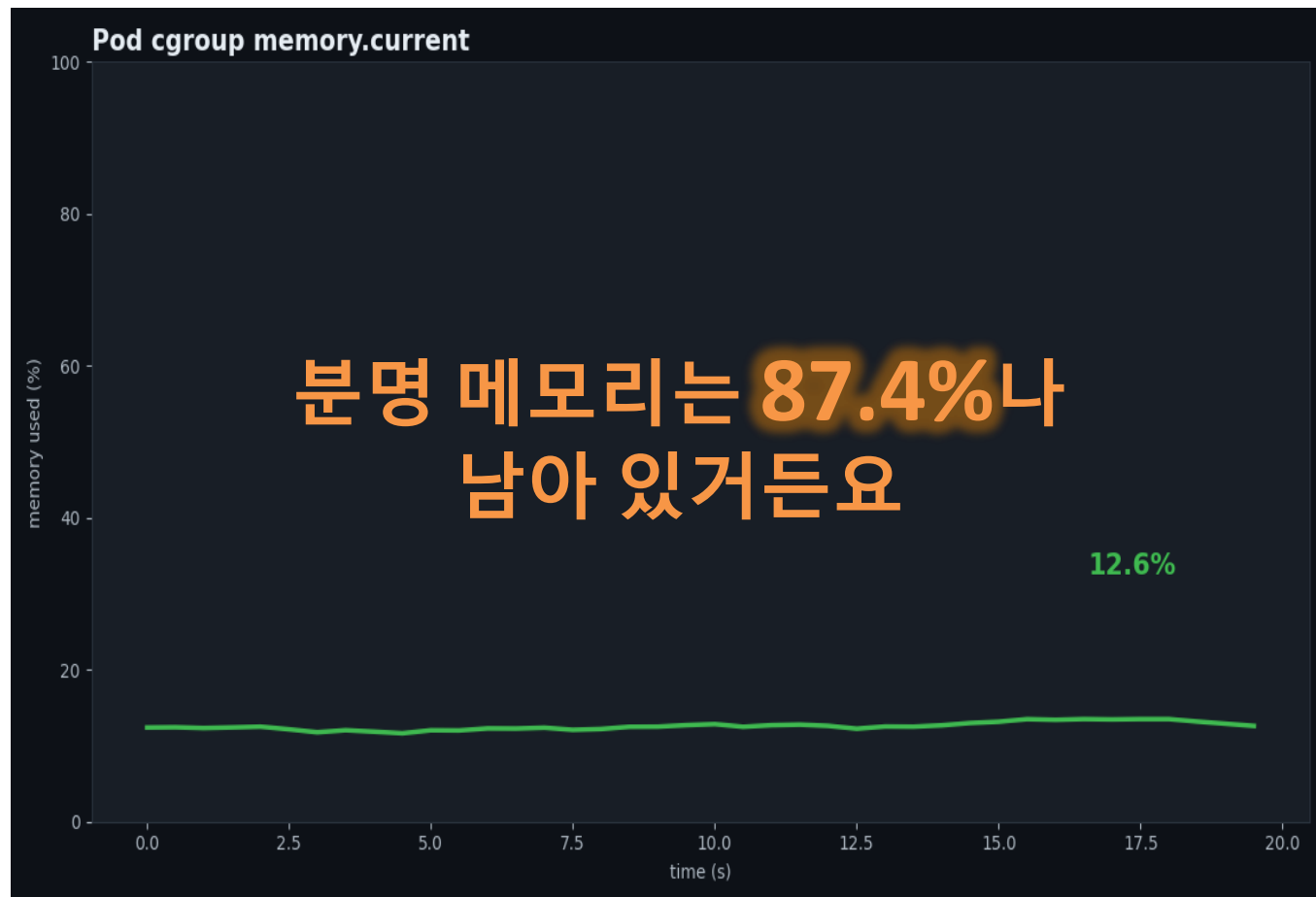
참고) 재현을 위해 의도적으로 작게 설정한 값(50Mi) — 실제 권장치 아님



OOM으로 프로세스(Pod)는 죽었지만.. 메모리 충분한데요?

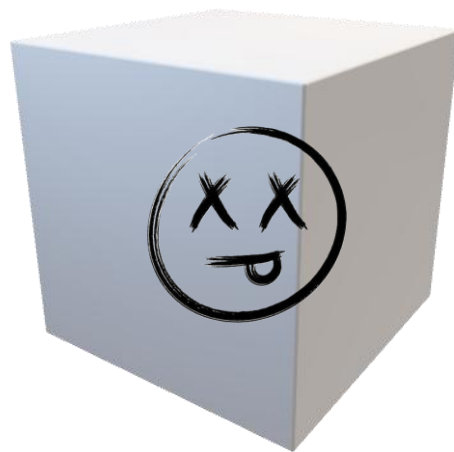


OOM으로 프로세스(Pod)는 죽었지만.. 메모리 충분한데요?



문제 정의

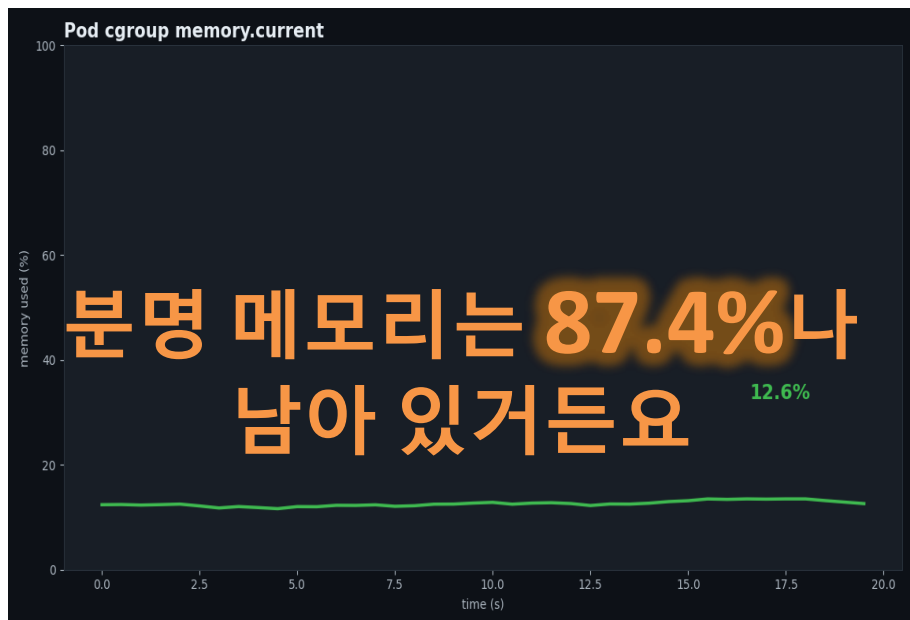
메모리는 충분한데, OOM Killed로 죽은 Pytorch 앱



그리고 죽은 앱은
Pytorch 워크로드 Pod 에요



메모리는 충분한데, OOM Killed로 죽은 Pytorch 앱



```
kubectl describe pod oom-demo
```

State: Terminated

Reason: OOMKilled

Exit Code: 137

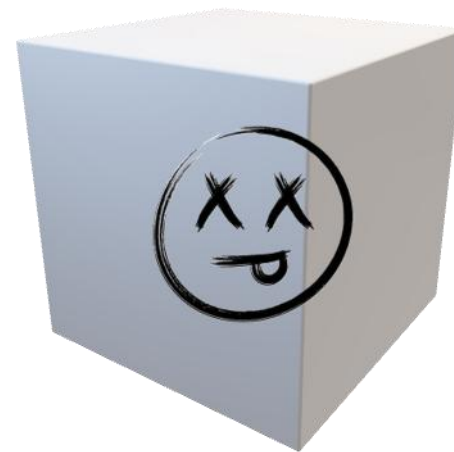
Limits:

memory: 50Mi

Requests:

memory: 50Mi

그리고 죽은 앱은
Pytorch 워크로드 Pod 에요



그렇지만 OOM
으로 죽었어요



Background knowledge

배경 지식 및 개념

1. /dev/shm
2. Pytorch DataLoader 동작 원리
3. 공유 메모리(/dev/shm)에 대한 FD의 표준 처리 순서



1. /dev/shm

```
[root@localhost ~]# df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs        874M   0    874M   0% /dev
tmpfs           893M   0    893M   0% /dev/shm
tmpfs           893M  17M   876M   2% /run
tmpfs           893M   0    893M   0% /sys/fs/cgroup
/dev/mapper/cs-root 47G  2.1G   45G   5% /
/dev/sda1       1014M  214M   801M  22% /boot
tmpfs           179M   0    179M   0% /run/user/0
```

데이터를 아주 빠르게 읽고 쓸 수 있는

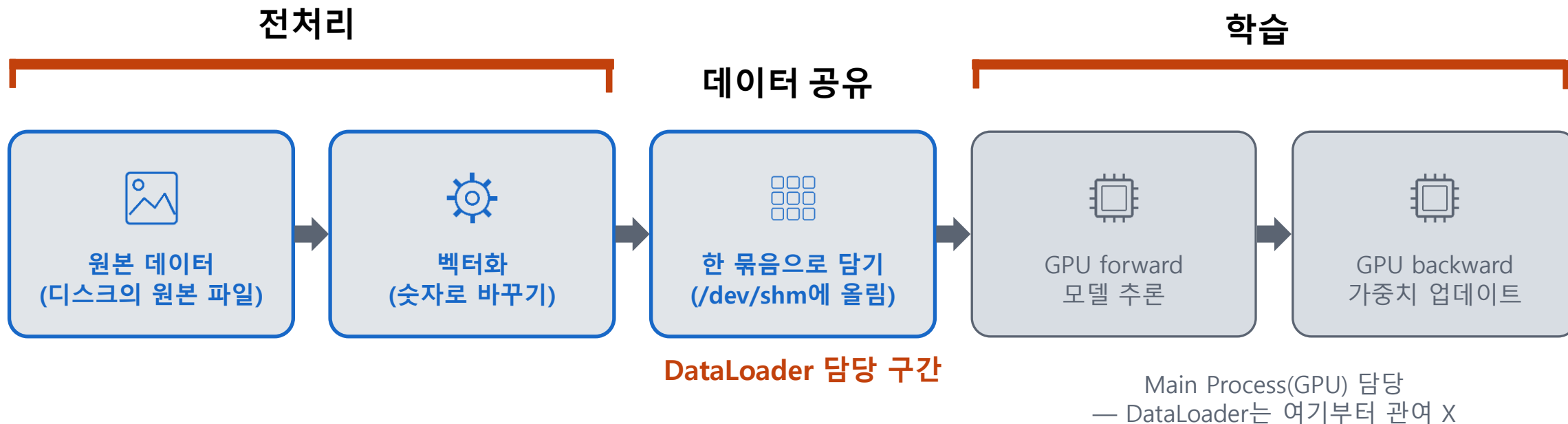
임시 파일(공유 메모리) 공간

(주로 프로세스 간 데이터 공유 시 사용)



2. PyTorch DataLoader 동작 원리

[AI 학습 사이클]

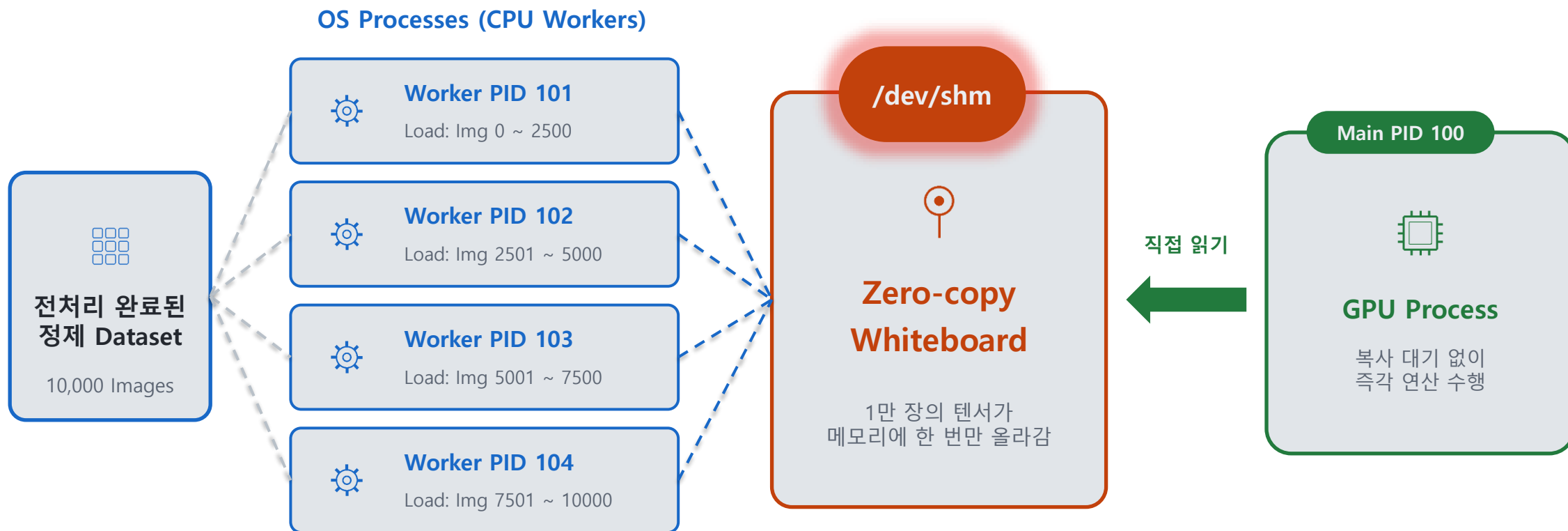


DataLoader 는 전처리 프로세스가 정제한 데이터 -> 학습 프로세스에게 전달하는 마중물.



2. PyTorch DataLoader 동작 원리

병렬 프로세스 처리와 임시 공유 메모리(/dev/shm) 활용



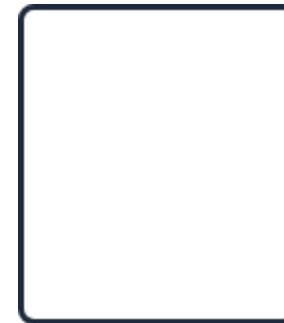
3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



공유 메모리에
임시 데이터 파일
하나 생성해 줘!

프로세스 12

경로: /dev/shm/



3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



공유 메모리에
임시 데이터 파일
하나 생성해 줘!

경로: /dev/shm/**tensor_xyz**

리눅스는 가상파일시스템(VFS)이므로, 모든 경로가 파일.

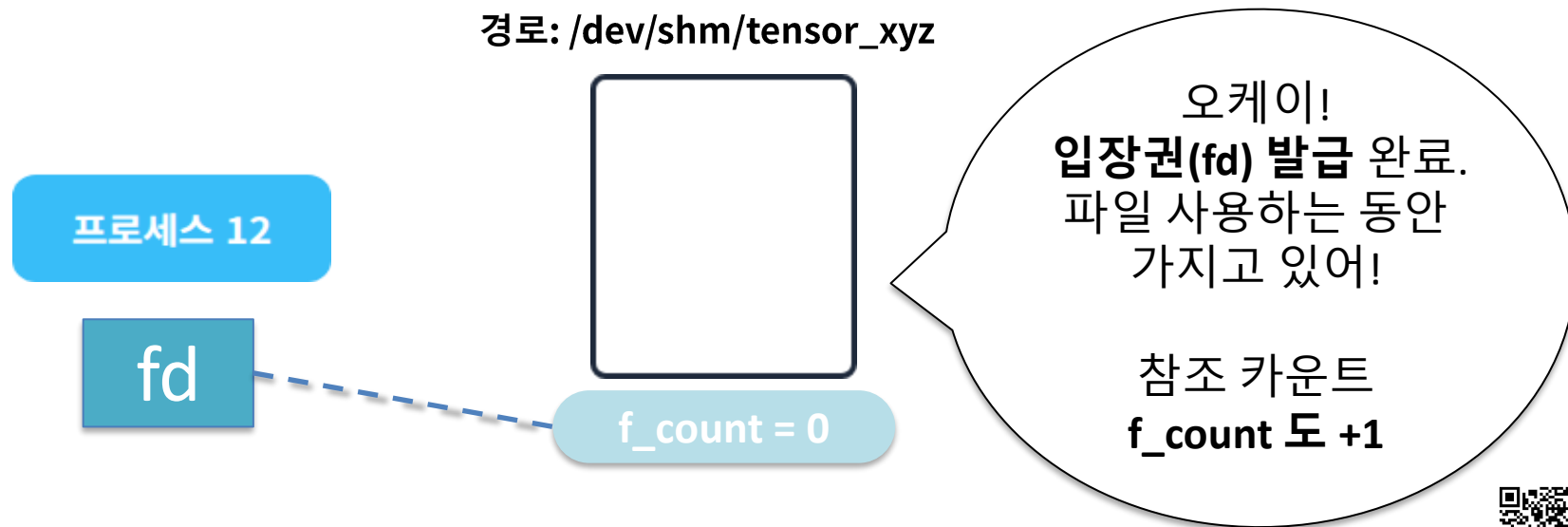
프로세스 12

공유 메모리인 /dev/shm/ 하위 경로도 파일

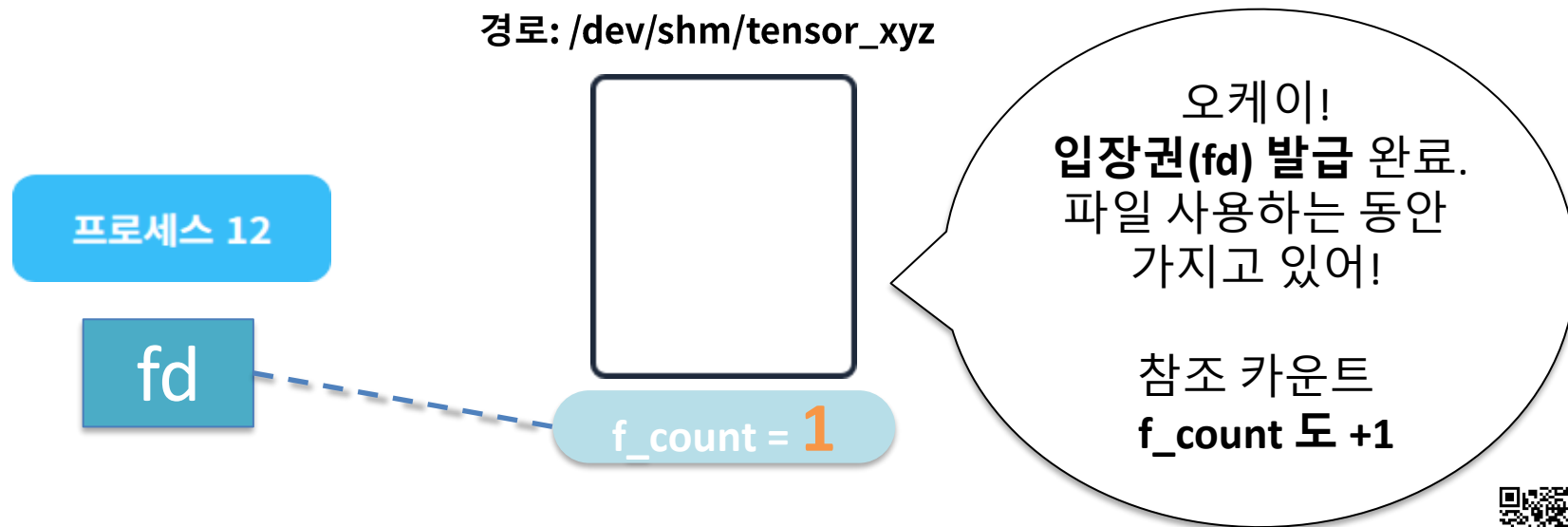
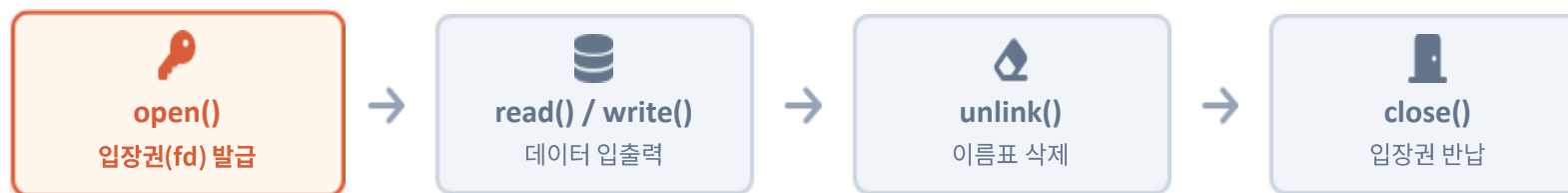
f_count = 0



3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



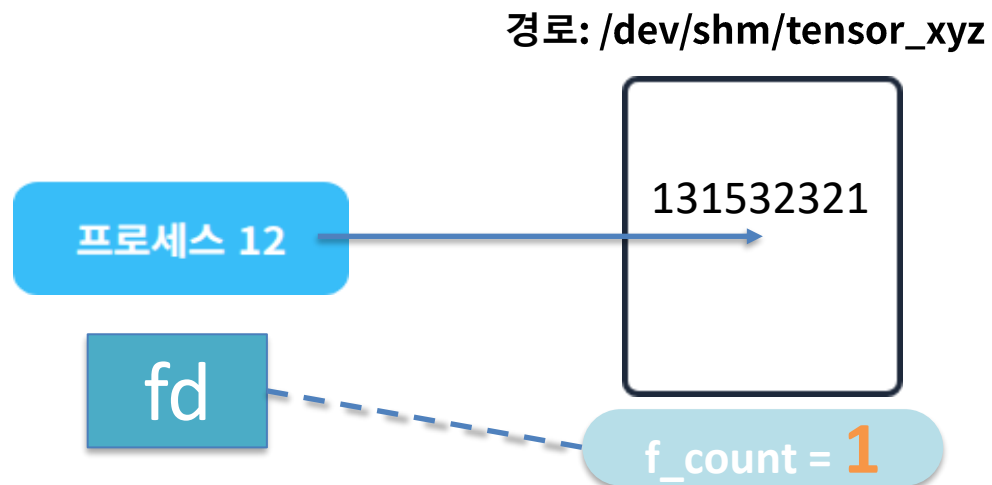
3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



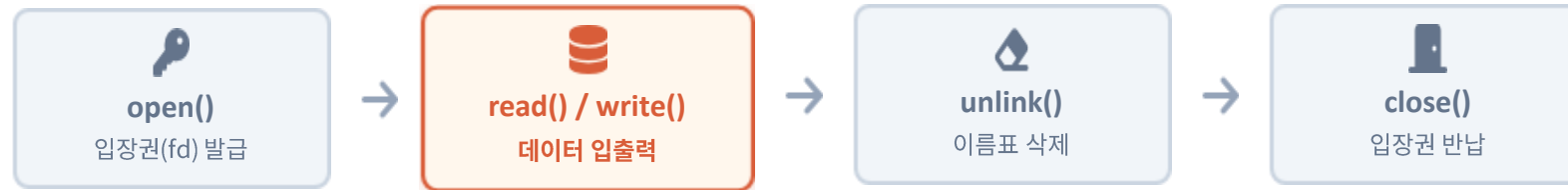
3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



발급받은
입장권(fd)로
데이터 고속 I/O
작업 시작한다!



3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서

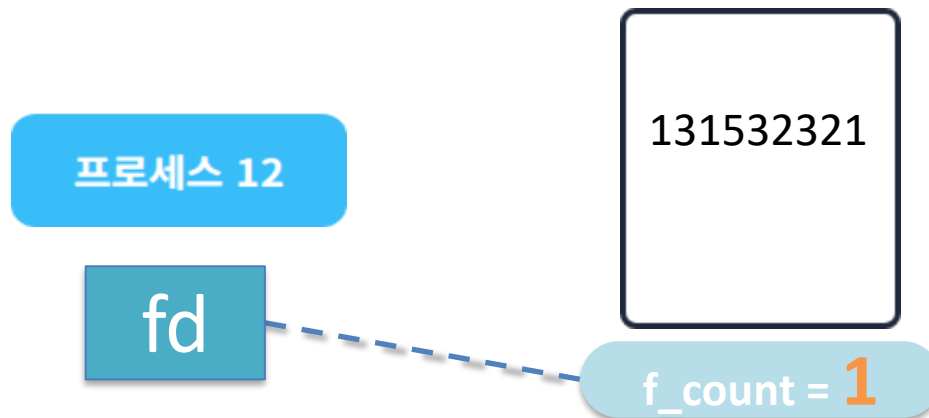


3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서

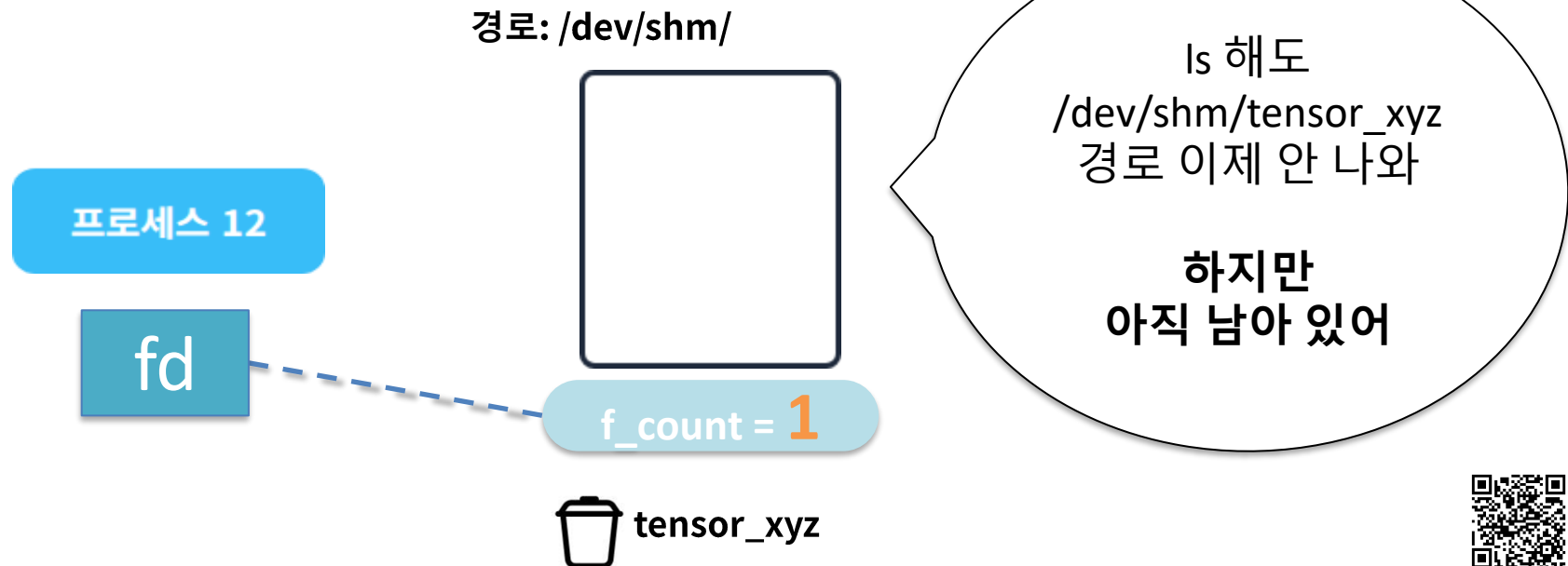


경로: /dev/shm/~~tensor_xyz~~

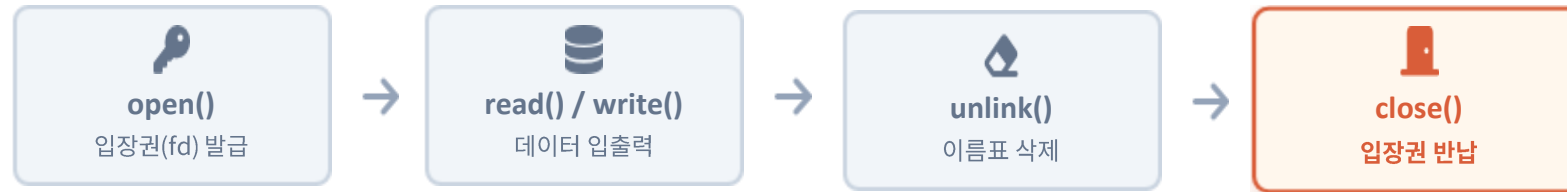
나 이제 작업 다
했으니
저 메모리 공간
필요 없어! 링크 해제!



3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



프로세스 12

경로: /dev/shm/



f_count = 0

tensor_xyz

이름표(unlink)도 없고,
입장권(f_count = 0)도
없네?

🔥 이제 디스크에서
진짜 삭제!



3. 공유 메모리(/dev/shm)에 대한 FD(File Descriptor) 의 표준 처리 순서



경로: /dev/shm/

프로세스 12



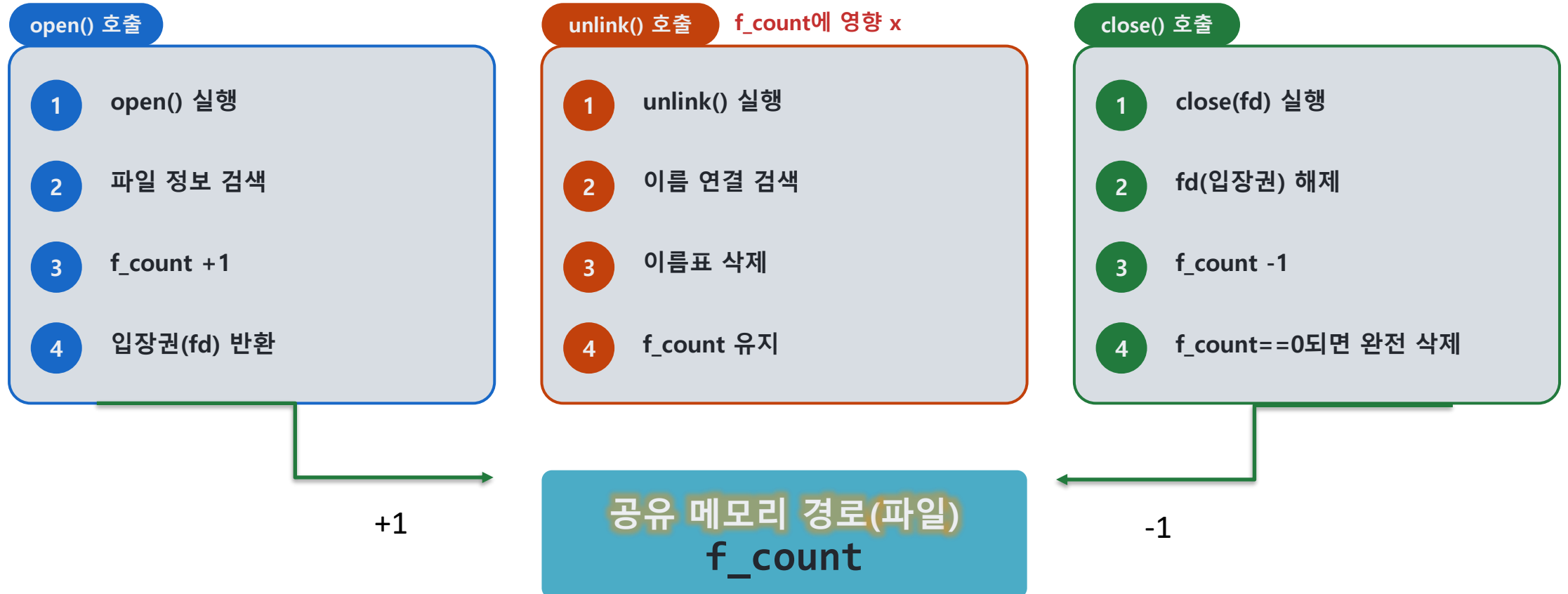
f_count = 0

공유 메모리 리소스
완벽하게
정리(Free) 완료!



임시 공유 메모리 사용 전 `open()` , 사용 후 `unlink()` -> `close()`

리눅스에서 파일은 '이름표가 없고(`unlink`)' + '아무도 안 쓸 때(`f_count=0`)' 디스크에서 실제로 삭제됨.



1. /dev/shm

→ 임시 파일이자 공유 메모리 (목적: 프로세스 간 데이터 공유)

2. Pytorch DataLoader 동작 원리

→ /dev/shm(임시 공유 메모리)에 AI가 사용할 데이터 몰아 넣기

3. 임시 공유 메모리이자 파일(/dev/shm)에 대한 FD의 표준 처리 순서

→ 파일은 이름표(Link)를 가짐

→ fd(입장권)가 있어야 파일 접근 가능, 발급 개수는 f_count에 기록

(1) 파일 사용 시작 : fd(입장권) 발급 및 f_count + 1

(2) 파일 사용 종료 : 이름 삭제(unlink) → fd(입장권) 반납, close()

→ f_count=0(장부 수정) → 최종 삭제



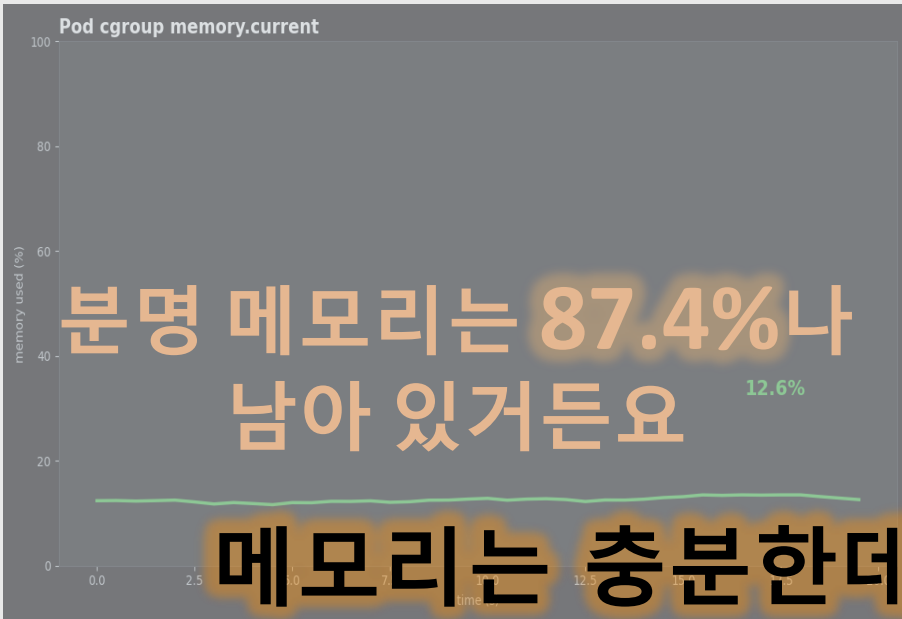
Problem Definition

문제 정의

1. 무엇을 의심해야 할까?
2. 원인은 OO 이다



1. 무엇을 의심해야 할까?



```
kubectl describe pod oom-demo
```

State: Terminated

Reason: OOMKilled

Exit Code: 137

Limits:

memory: 50Mi

Requests:

memory: 50Mi

OOM Killed로 죽은 Pytorch 앱

그리고 죽은 앱은
Pytorch 워크로드 Pod 에요



그렇지만 OOM
으로 죽었어요



1. 무엇을 의심해야 할까?



```
kubectl describe pod oom-demo
```

State:	Terminated
Reason:	OOMKilled
Exit Code:	137

Requests:	
memory:	50Mi

메모리는 충분한데, OOM Killed로 죽은 Pytorch 앱

현재 모니터링이 감지하지 못하는 정보 공백이 있다.

리눅스(Ubuntu) 환경에서 자원 접근 및 정보 조회의 차이가 어디서 날까?



1. 무엇을 의심해야 할까?

유저스페이스 / 모니터링이 보는 층

kubectl top

Prometheus

cgroup 메모리 통계
(/sys/fs/cgroup)

모니터링은 유저스페이스 범위의 정보만 조회



1. 무엇을 의심해야 할까?

유저스페이스 / 모니터링이 보는 층

kubectl top

Prometheus

cgroup 메모리 통계
(/sys/fs/cgroup)

커널 스페이스 / 실제로 자원을 쥐고 있는 층

신원 미상의 무언가



1. 무엇을 의심해야 할까?

커널 스페이스 / 실제로 자원을 쥐고 있는 층

신원 미상의 무언가

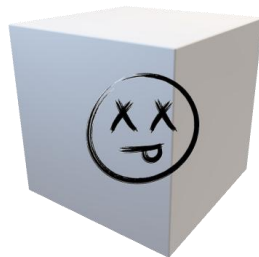
커널 스페이스에서 어떤 정보를 확인해야 할까?

Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.



1. 무엇을 의심해야 할까?

그리고 죽은 앱은
Pytorch 워크로드 Pod 에요



Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

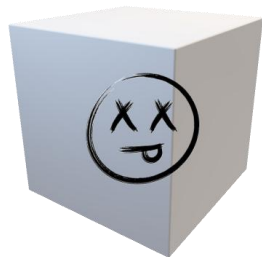
OOM으로 죽은 프로세스는 정확히 어떤 동작을 하고 있었지?

PyTorch DataLoader 워커 프로세스였다



1. 무엇을 의심해야 할까?

그리고 죽은 앱은
Pytorch 워크로드 Pod 에요



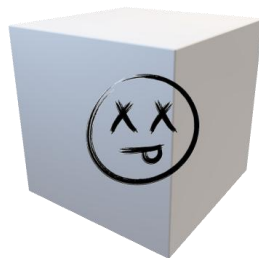
Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

PyTorch DataLoader 워커 프로세스는 어떤 메모리를 쓰지?



1. 무엇을 의심해야 할까?

그리고 죽은 앱은
Pytorch 워크로드 Pod 에요



Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

PyTorch DataLoader 워커 프로세스는 어떤 메모리를 쓰지?

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm



1. 무엇을 의심해야 할까?

Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm



1. 무엇을 의심해야 할까?

Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm

RSS는 cgroup 메트릭에 그대로 기록된다.



1. 무엇을 의심해야 할까?

Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm

RSS는 cgroup 메트릭에 그대로 기록된다.

문제였다면, 진작 대시보드에 떴을 것이다



1. 무엇을 의심해야 할까?

Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm

RSS는 cgroup 메트릭에 그대로 기록된다.

문제였다면, 진작 대시보드에 떴을 것이다

하지만 `kubectl top / Prometheus`에선 계속 메모리가 충분하다고 나왔다.



1. 무엇을 의심해야 할까?

Pod에서 기동이 되는 것도, 메모리를 쥐고 있는 것도 프로세스다.

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm

RSS는 cgroup 메트릭에 그대로 기록된다.

문제였다면, 진작 대시보드에 뺏을 것이다

하지만 kubectl top / Prometheus에선 계속 메모리가 충분하다고 나왔다.

RSS가 문제면 cgroup 메트릭을 보여주는
kubectl top / Prometheus에 메모리 부족 정보가 뺏겠지.



1. 무엇을 의심해야 할까?

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm

RSS가 문제면 cgroup 메트릭을 보여주는
kubectl top / Prometheus에 메모리 부족 정보가 났겠지.

RSS는 범인이 아니다



1. 무엇을 의심해야 할까?

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm

RSS는 범인이 아니다



1. 무엇을 의심해야 할까?

코드/가중치를 올리는 RSS, 그리고 워커끼리 텐서를 주고받는 /dev/shm

RSS는 범인이 아니다

그렇다면 남은 용의자... 진실은 언제나 하나! — /dev/shm



1. 무엇을 의심해야 할까?

그렇다면 남은 용의자... 진실은 언제나 하나! — /dev/shm



커널 통계에서 /dev/shm은 캐시(Shmem)로 잡힌다



1. 무엇을 의심해야 할까?

그렇다면 남은 용의자... 진실은 언제나 하나! — /dev/shm



커널 통계에서 /dev/shm은 캐시(Shmem)로 잡힌다

모니터링 툴은 캐시를 어떻게 취급하지?



1. 무엇을 의심해야 할까?

그렇다면 남은 용의자... 진실은 언제나 하나! — /dev/shm



커널 통계에서 /dev/shm은 캐시(Shmem)로 잡힌다

모니터링 툴은 캐시를 어떻게 취급하지?

"언제든 비울 수 있다"고 보고, 사용량 계산에서 빼거나 후순위로 취급한다



1. 무엇을 의심해야 할까?

그렇다면 남은 용의자... 진실은 언제나 하나! — /dev/shm

커널 통계에서 /dev/shm은 캐시(Shmem)로 잡힌다

모니터링 툴은 캐시를 어떻게 취급하지?

"언제든 비울 수 있다"고 보고, 사용량 계산에서 빼거나 후순위로 취급한다

근데 /dev/shm 은 디스크 원본이 없는 임시 파일(공유 메모리),
그래서 강제로 비울 수 없다



1. 무엇을 의심해야 할까?

그렇다면 남은 용의자... 진실은 언제나 하나! — /dev/shm



1. /dev/shm의 정체: 파일

리눅스의 대원칙: "모든 것은 파일이다."

/dev/shm도 엄연히 **파일**. HDD가 아니라, 오직 '메모리(RAM)'에만 존재하는 파일



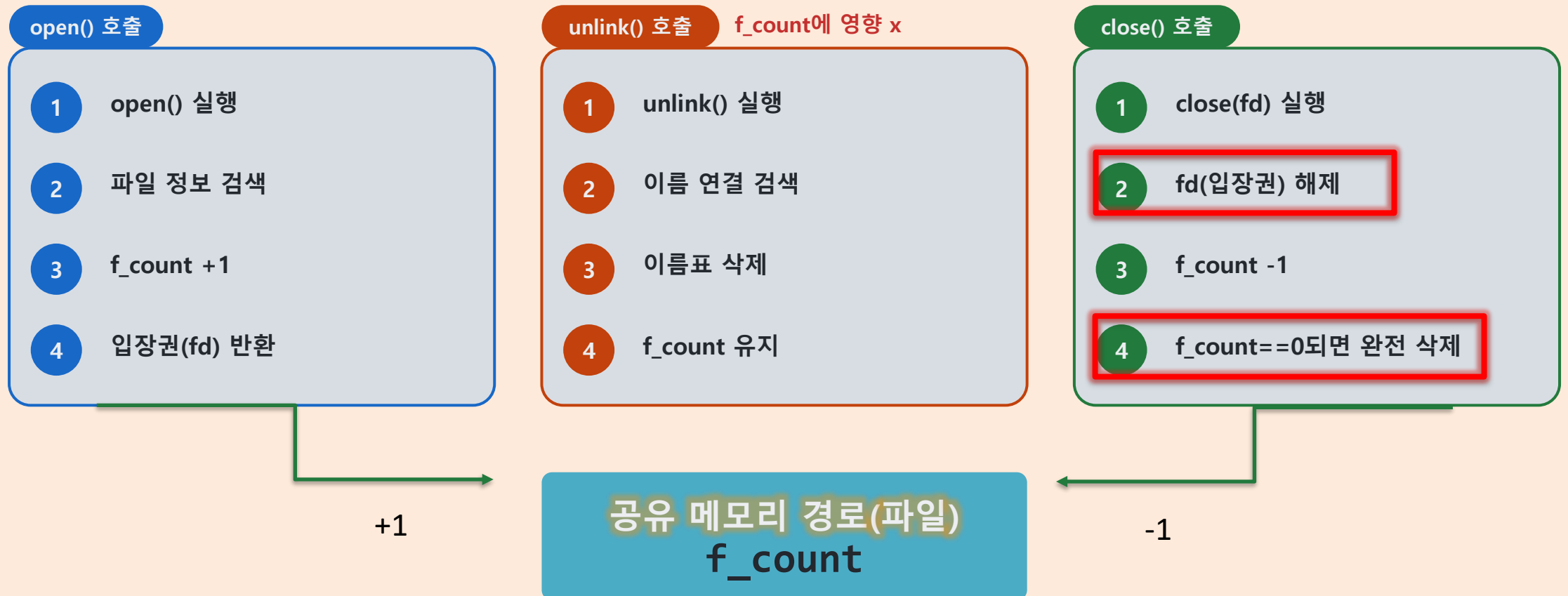
2. 파일 접근의 규칙

프로세스가 '메모리 파일'을 사용하려면,
반드시 커널로부터 '입장권(fd)'를 발급받아야 합니다.



임시 공유 메모리 사용 전 `open()` , 사용 후 `unlink()` -> `close()`

리눅스에서 파일은 '이름표가 없고(unlink)' + '아무도 안 쓸 때(`f_count=0`)' 디스크에서 실제로 삭제됨.



임시 공유 메모리 사용 전 `open()` , 사용 후 `unlink()` -> `close()`

리눅스에서 파일은 '이름표가 없고(unlink)' + '아무도 안 쓸 때(`f_count=0`)' 디스크에서 실제로 삭제됨.

`open()` 호출

- 1 `open()` 실행
- 2 파일 정보 검색
- 3 `f_count + 1`
- 4 입장권(`fd`) 반환

`unlink()` 호출 `f_count`에 영향 x

- 1 `unlink()` 실행
- 2 이름 연결 검색
- 3 이름표 삭제
- 4 `f_count` 유지

`close()` 호출

- 1 `close(fd)` 실행
- 2 `fd`(입장권) 해제
- 3 `f_count - 1`
- 4 `f_count == 0`되면 완전 삭제

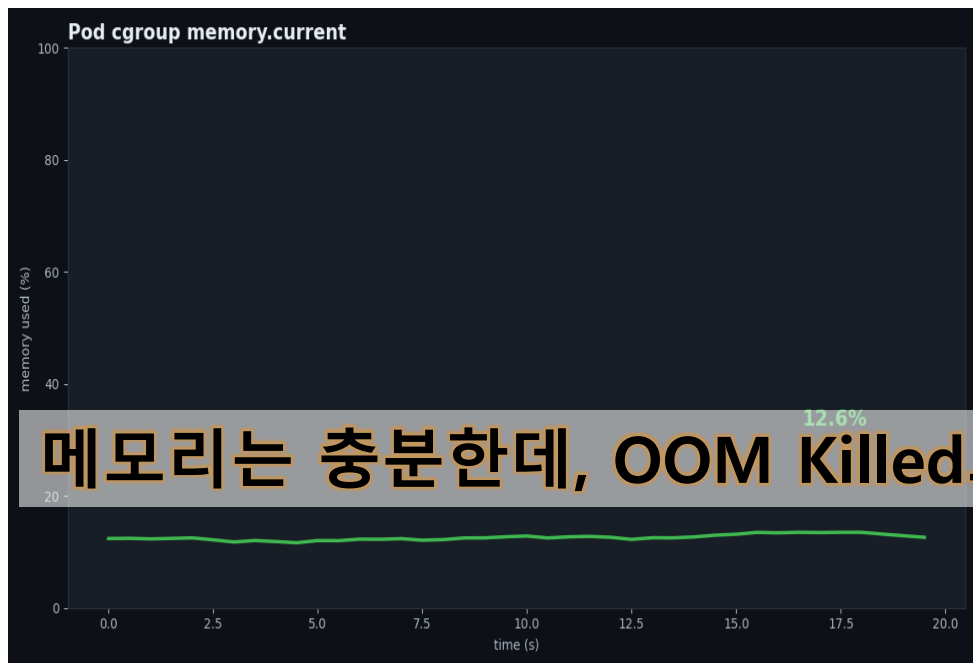


`f d` 전체 회수 & `f_count == 0` 이어만 메모리 파일 진짜 삭제 !

"이 입장권(`fd`)를 쥔 프로세스가 단 하나라도 남아있다면,
커널은 절대 이 메모리를 회수하지 않는다."



1. 무엇을 의심해야 할까?



```
kubectl describe pod oom-demo
```

State: Terminated

Reason: OOMKilled

Exit Code: 137

Limits:

Requests:

memory: 50Mi

[모니터링으로 확인 가능 정보]

- 프로세스(파드)가 죽음
- 프로세스가 점유하는 메모리가 터짐
(/dev/shm으로 유추)



1. 무엇을 의심해야 할까?

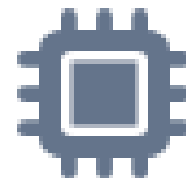
그렇다면 남은 용의자... 진실은 언제나 하나! — /dev/shm
파일



fd 전체 회수 & f_count == 0 이어만 메모리 파일 진짜 삭제!

PyTorch
DataLoader
프로세스

> 죽었다는 프로세스가 정말
정상 종료되었나요?



/dev/shm에 있는 임시 파일(공유 메모리)

> 이 메모리(파일) 정말 삭제됐나요?



1. 무엇을 의심해야 할까?

프로세스와 임시 파일(공유 메모리)을 까보자!

```
$ ps -o pid,ppid,cmd -p 461031
PID  PPID  CMD
461031  1  python3 worker.py
```

PPID 1 — 부모 없이 떠 있다 (고아 프로세스)

```
$ ls -la /proc/461031/fd | grep tensor
lrwx----- 1 root root 64 ... 7 ->
/dev/shm/tensor_461031_..._0 (deleted)
```

(deleted)인데, 왜 아직 열려 있지?

▶ 하드디스크에서 이름표는 뗐지만(unlink 완료), 실제 삭제(close)가 되지 못한 상황

1. 무엇을 의심해야 할까?

OOM 터진 PID 모르면?

detect.bt

```
kprobe:filp_close {  
    $file = (struct file *)arg0;  
    $count = $file->f_count.counter;  
  
    if ($count > 1 && comm == "python3") {  
        $name = $file->f_path.dentry->d_name.name;  
        printf("[ ⚠ ZOMBIE FD] PID: %d | Ref Count: %d  
                | Kernel Name: %s\n", pid, $count, str($name));  
    }  
}
```

닫는 순간(filp_close)에
왜 f_count > 1 인가?

정상이라면 닫을 때 내 몫을 빼서 0이 되어야 함.

하지만 1보다 크다는 것은?

"내가 닫고 나가도, 뒤에 숨어있는 고아
프로세스들이 이 메모리를 여전히 쥐고
안 놔준다"는 증거.

Ref Count(f_count)가 1보다 크면 자식 프로세스들이 계속 /dev/shm 공간을 쥐고 있다는 뜻

1. 무엇을 의심해야 할까?

```
root@bpf-debug:~# kill -9 85
```

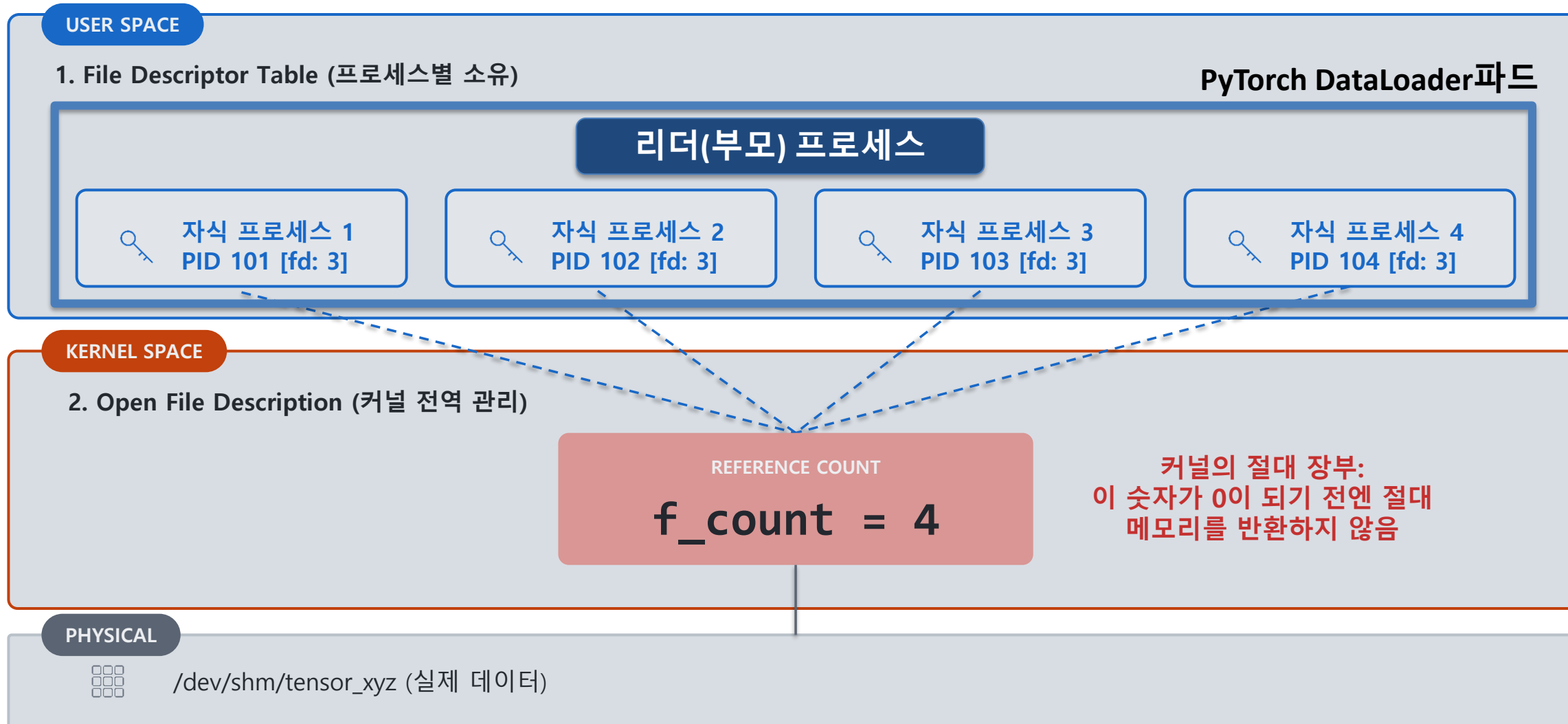
```
[ZOMBIE FD] PID: 368 | Ref Count: 33 | Kernel Name: torch_85_1210951755_0
```

리더/부모 프로세스(PID 85)를 강제 종료(kill -9) 시켰더니?
리더는 죽으면서 파일을 닫았는데, Ref Count가 33이나 남음.

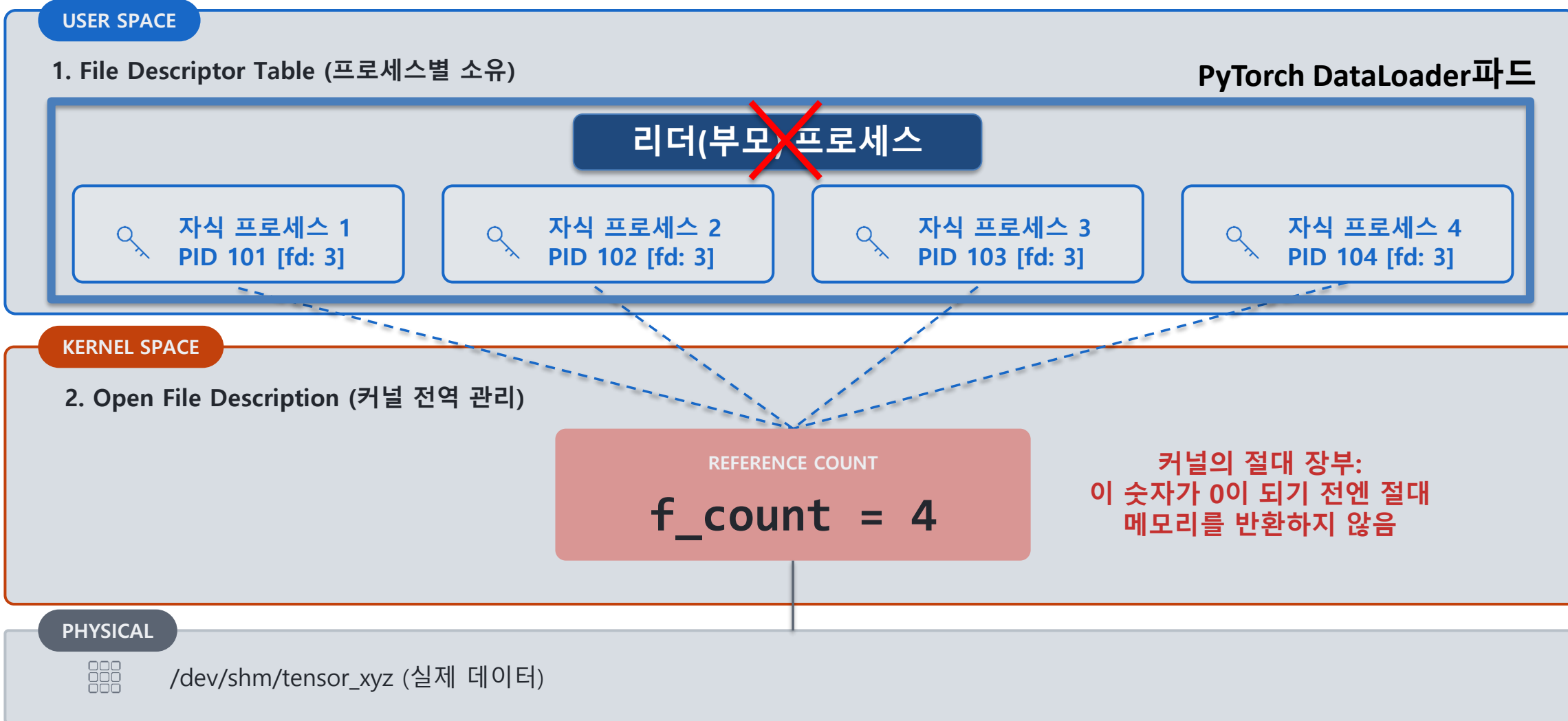
→ 숨어있던 자식 프로세스(워커) 33개가 안 죽고 메모리 (torch_85_...) 를 잡고 있음



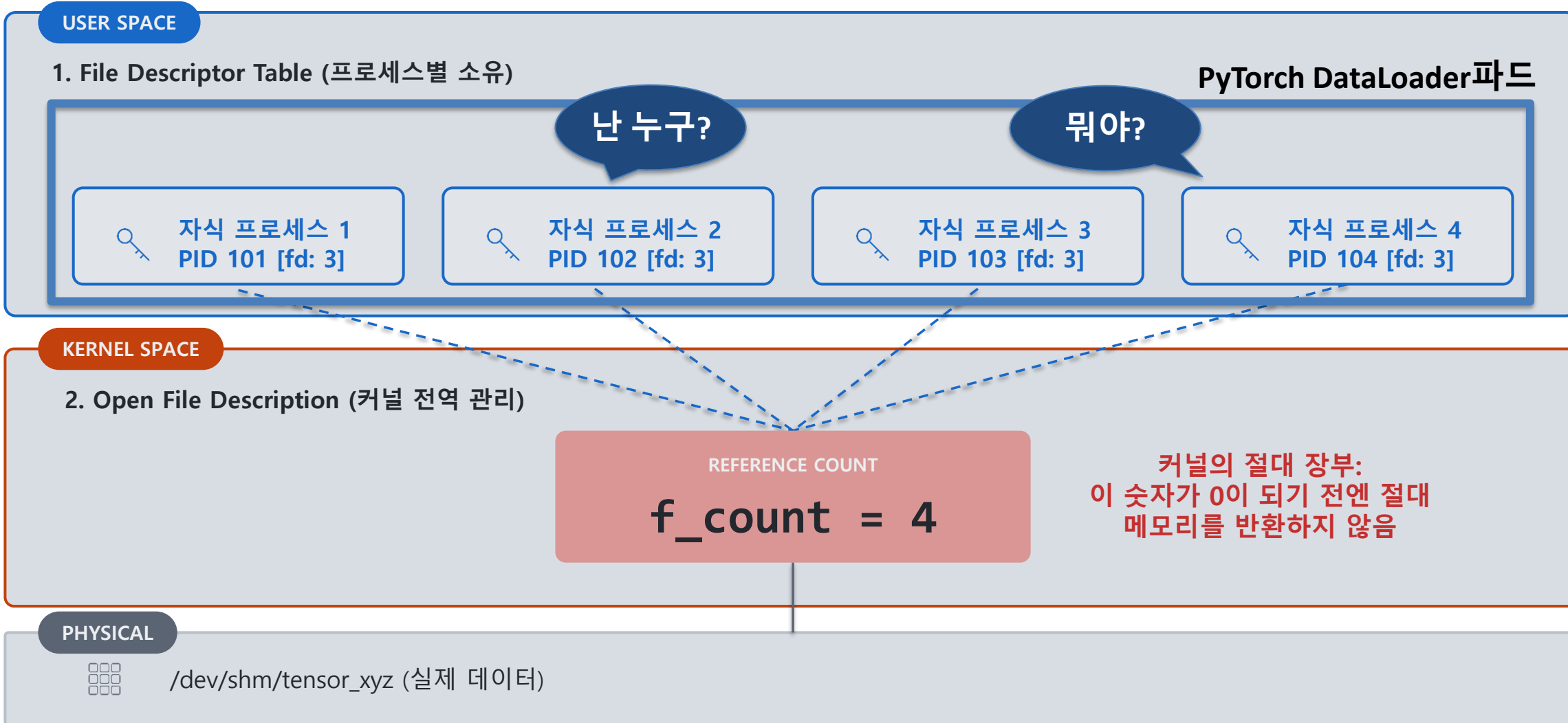
문제 정의



메모리 OOM! 프로세스 kill -9



메모리 OOM! 프로세스 kill -9



문제 정의

메모리 OOM! 프로세스 kill -9

USER SPACE

1. File Descriptor Table (프로세스별 소유)

PyTorch DataLoader파드

난 누구?

뭐야?



자식 프로세스 1
PID 101 [fd: 3]



자식 프로세스 2
PID 102 [fd: 3]



자식 프로세스 3
PID 103 [fd: 3]



자식 프로세스 4
PID 104 [fd: 3]

▶ 메모리를 터트린 부모 프로세스만 죽고, 자식들은 안 죽었다!

KERNEL SPACE

2. Open File Description (커널 전역 관리)

메모리를 터트린 자식 프로세스들이 안 죽었고,
REFERENCE COUNT
f_count = 4
메모리 공간(fd)도 계속 쥐고 있음!

커널의 절대 장부:
이 수가 0이 되기 전엔 절대
메모리를 반환하지 않음

PHYSICAL



/dev/shm/tensor_xyz (실제 데이터)

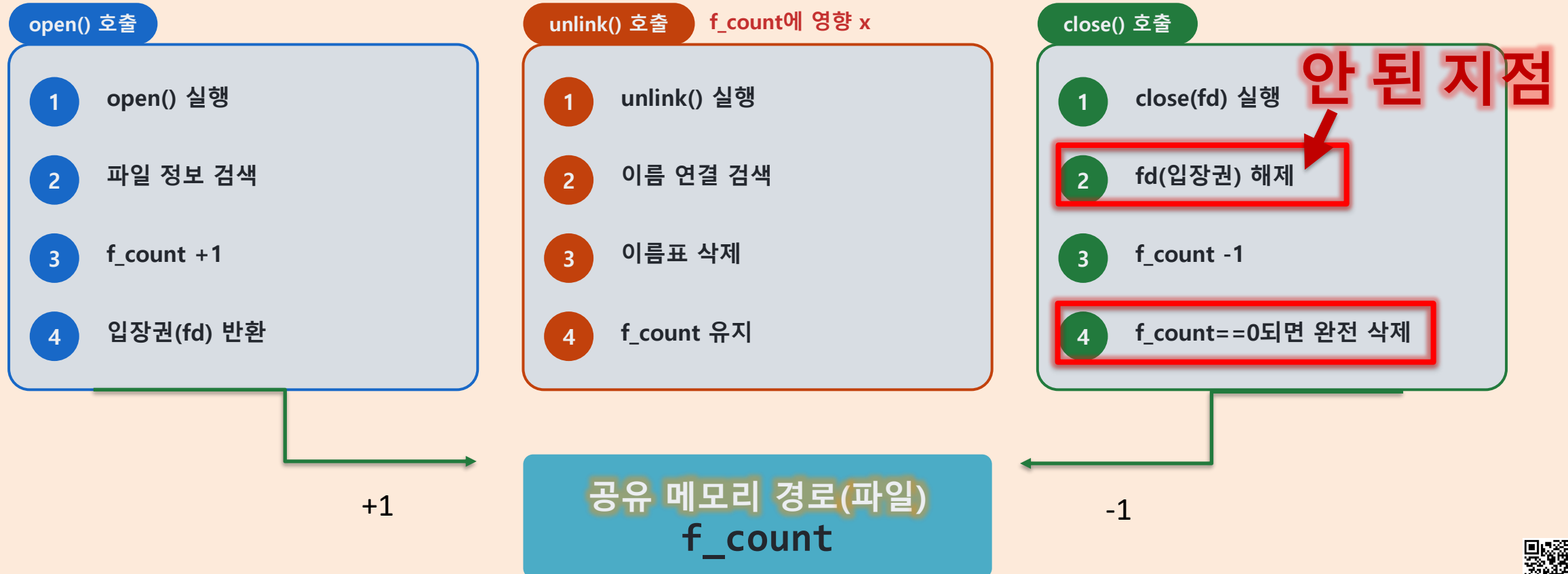
2. 원인은 OO 이다.

원인은 **Zombie FD**이다.

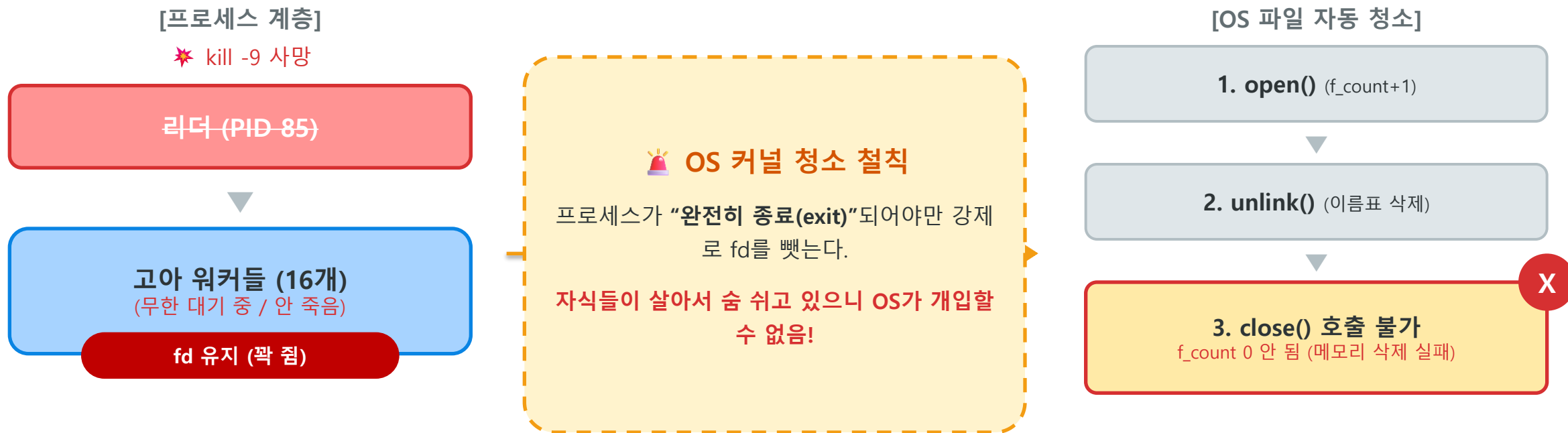


임시 공유 메모리 사용 전 `open()` , 사용 후 `unlink()` -> `close()`

리눅스에서 파일은 '이름표가 없고(unlink)' + '아무도 안 쓸 때(`f_count=0`)' 디스크에서 실제로 삭제됨.



2. 원인은 OO 이다.



프로세스가 죽으면 메모리도 반환되어야 하지만, 살아남은 자식들이 점유권을 놓지 않아 커널의 개입(임시 파일 삭제/청소)이 차단된 상태.

Problem Solution

문제 해결책

1. 해결 방법은 오직 OO뿐
2. 해결하자!



1. 해결 방법은 오직 죽음뿐. (Kill = Reclaim — 죽여야만 회수된다)

1단계 · 원칙

살아있는 프로세스가 쥔 메모리를 우아하게 뺏는 방법은? → 고아 프로세스를 죽이는 것만이 유일한 회수법

참고) 타 프로세스의 fd만 외부에서 강제로 뺏는 리눅스 API는 존재하지 않음
→ 스스로 닫지 못하는 무한 대기(Hang) 상태에서, 외부가 자원을 강제 회수할 유일한 수단은 '프로세스 kill'뿐.

2단계 · eBPF 세이프티넷

부모 프로세스의 사망(exit) 이벤트를 커널에서 실시간으로 낚아챌
/ 남겨진 고아 워커들을 즉시 식별하여 유저스페이스로 전달

3단계 · 선택적 모드?

기본: dry-run
로그만 남기고 실제 kill 없음

--enforce
실제 SIGTERM → SIGKILL 실행

전 과정 audit 로그로 남김

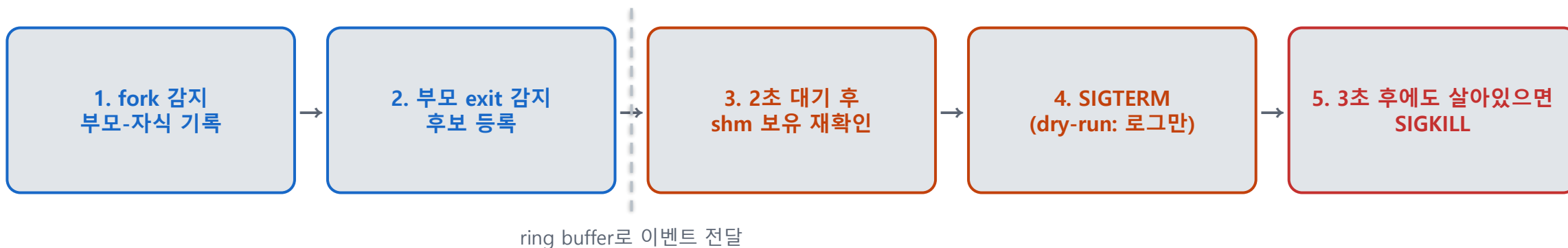
참고) SIGTERM으로 정상 종료(Graceful Shutdown) 기회를 먼저 주고, 응답 없으면 SIGKILL로 강제 처단

1. 해결 방법은 오직 죽음뿐. — with bpftrace

참고) `detect.bt(filp_close)`는 파일이 "닫힐 때"만 걸린다 — 근데 진짜 문제는 영원히 안 닫히는 것.
그래서 죽일 대상(PID)을 미리 짚해두는 `fork/exit` 감시로 변경.

커널 안 (eBPF) · `orphan_guard.bpf.c`

유저스페이스 · `orphan_guard.c`



`detect.bt`가 '누군가 닫는 순간'에만 마주칠 수 있는 스냅샷 진단이었다면,
이 파이프라인은 `fork/exit`부터 상시 감시하는 방어 시스템 코드



2. 해결하자! — 코드는 커널 안/밖, 두 파일로 나뉜다

왜 커널에서 직접 안 죽이고 유저스페이스로 보낼까? — "3초 대기"나 "안전한 Kill 명령" 같은 무거운 제어 로직은 커널 밖(Userspace)에서 처리하는 것이 eBPF의 안전한 설계 원칙이기 때문

커널 안 — 감시 / 커널 스페이스 (eBPF)

```
orphan_guard.bpf.c

SEC("tp_btf/sched_process_exit")
int on_exit(struct task_struct *task) {
    ppid = task->real_parent->tgid;
    // fork/exit을 커널에서 직접 캡처
    emit(EVENT_EXIT, pid, ppid, ...);
}
```

부모가 죽는 순간을 놓치지 않고 잡아서 유저스페이스로 올려보낸다

밖 — 판단과 조치 / 유저 스페이스

```
orphan_guard.c

#define KILL_GRACE_SEC 3
#define MIN_PARENT_AGE_SEC 2

if (!dry_run) kill(pid, SIGTERM);
// 유예 후에도 fd를 쥐고 있으면
if (!dry_run) kill(pid, SIGKILL);
```

일정 시간 기다렸다 SIGTERM → SIGKILL로 격상,
기본은 dry-run



Conclusion

결론

1. 원인은 OO 이다
2. 해결 방법은 오직 OO뿐



Conclusion

결론

1. 원인은 **Zombie FD** 이다

- ▶ 부모 프로세스가 죽으면 메모리도 반환되어야 하지만, 살아남은 자식 프로세스(고아)들이 점유권(fd)을 놓지 않아 임시 파일(/dev/shm)이 안 해제되고, 계속 차 있는 상태



Conclusion

결론

1. 원인은 **Zombie FD** 이다

- ▶ 부모 프로세스가 죽으면 메모리도 반환되어야 하지만, 살아남은 자식 프로세스(고아)들이 점유권(fd)을 놓지 않아 임시 파일(/dev/shm)이 안 해제되고, 계속 차 있는 상태

2. 해결 방법은 오직 **죽음**뿐

- ▶ 자식 프로세스(고아) 프로세스들을 죽여, fd를 뺏는다.
f_count=0이므로 OS가 임시 파일(/dev/shm)을 최종적으로 해제한다.



감사합니다

코드 LINK

<https://github.com/pmj-chosim/orphanmemkiller/tree/main/src>



Q&A 연락 / LinkedIn



UbuCon x
MiniDebConf Korea 2026



“PyTorch 자체엔 부모 죽었을 때 방어 코드 없나요?”

있습니다 — ManagerWatchdog

- torch/utils/data/_utils/worker.py — 워커 루프가 매 틱마다 부모 생존을 직접 확인
- Unix: self.manager_dead = os.getppid() != self.manager_pid
- 부모가 죽은 걸 감지하면 워커는 루프를 빠져나와 스스로 종료

```
watchdog = ManagerWatchdog()
while watchdog.is_alive():
    r = index_queue.get(timeout=MP_STATUS_CHECK_INTERVAL)
```

근데 이 체크가, 블로킹 I/O 중엔 안 돕니다

- 위 watchdog 루프는 index_queue.get(timeout=...) 대기 구간에서만 실행됨
- 워커가 큐에서 작업을 꺼내 dataset.__getitem__ / collate_fn을 실행하는 동안(= S3/NFS 읽기 같은 blocking I/O)은 이 루프 밖
- 즉 부모가 죽어도, I/O 대기(D-state)에 걸린 워커는 자기 죽음을 감지할 기회 자체가 없음

출처: pytorch/pytorch main · torch/utils/data/_utils/worker.py (ManagerWatchdog)

→ PyTorch 자체 방어책은 이미 있고, 블로킹 I/O 도중 체크 안 되는 구멍이 이 세션의 타겟.

"PyTorch DataLoader의 Queue에 timeout을 강제하거나 non-blocking I/O로 고치는 방식은 고려 안 해보셨나요?"

방향은 맞습니다 — 근데 근본적인 벽이 있어요

- D-state(uninterruptible sleep)에 걸린 프로세스는 커널 I/O가 끝날 때까지 유저스페이스 코드 자체를 실행 못 함
- 그 안에 아무리 timeout 체크를 넣어도, 그 코드가 실행될 기회 자체가 없음

설령 PyTorch를 고쳐도, 이 문제 하나만 없어질 뿐입니다

- PR이 머지돼도 PyTorch에만 적용됨 — Ray, joblib, Jupyter 등 같은 패턴을 쓰는 다른 프레임워크는 그대로 남음
- 커널 레벨 문제는 커널 레벨에서 잡아야, 어떤 프레임워크를 쓰든 같은 안전망이 적용됨

출처: orphan-guard 자체 벤치마크(AKS Standard_D4s_v5, 2026-08-02), bench_dataloader.py / fork_stress.sh

→ 애플리케이션 코드를 못 고치는 상황(Ray/Jupyter 등)까지 커버하려면, 결국 커널 레벨 세이프티넷이 필요합니다.

“리더(PID 1)가 죽으면 컨테이너 전체가 같이 안 죽나요?”

PID 네임스페이스 규칙대로면 맞는 지적입니다.

- `pid_namespaces(7)`: PID 1이 종료되면 커널이 같은 네임스페이스의 나머지 프로세스 전부에 SIGKILL
- 학습 스크립트 자신이 컨테이너의 PID 1이라면, 리더가 죽는 순간 워커도 다 같이 죽어야 정상

근데 이 상황에서의 PID 1은 다릅니다.

- `ps -o pid,ppid,cmd` 결과: 고아 워커의 PPID가 1 — 즉 컨테이너의 진짜 init(예: tini/bash)에게 재입양됨
- PID 1이 살아있으니 네임스페이스는 안 죽고, SIGKILL 캐스케이드도 발생 안 함
- 리더(학습 프로세스)는 PID 1의 자식일 뿐 — 리더가 죽어도 컨테이너/네임스페이스는 계속 RUNNING

+ 설령 PID 1이 자식들을 정리하려 해도, I/O 대기(D-state)에 걸린 워커는 SIGKILL을 받아도 당장 안 죽습니다.
— Q1의 ManagerWatchdog 한계와 같은 근본 원인.

"D-state에 깊게 빠진 프로세스는 SIGKILL을 날려도 당장 안 죽지 않나요? 이 경우 orphan_guard는 어떻게 처리하나요?"

맞습니다 — SIGKILL도 D-state엔 당장 안 먹힙니다

- SIGKILL은 그 프로세스가 커널에서 돌아올 때까지 pending 상태로 대기합니다
- 즉 I/O가 안 끝나는 동안은, 신호를 보내도 문제가 그대로 남아있는 시간이 있습니다

```
watchdog = ManagerWatchdog()
while watchdog.is_alive():
    r = index_queue.get(timeout=MP_STATUS_CHECK_INTERVAL)
```

이 경우 orphan-guard도 즉시 죽이지는 못합니다

- I/O가 끝나거나 타임아웃 나는 순간, 대기 중이던 SIGKILL이 바로 전달돼서 그때 정리됩니다
- 근데 스토리지가 영원히 응답을 안 하면, 유저스페이스에서 할 수 있는 게 없습니다 — 이건 스토리지 계층의 문제고, 여기서 한계점입니다

→ orphan-guard는 언젠가 깨어날 좀비를 확실히 잡는 도구지, 모든 무한 행을 뚫는 만능 열쇠는 아닙니다.

"캐시/tmpfs만 나눠서 모니터링하고, 킬 때만 lsof나 /proc로 찾아서 죽이면 될 것 같은데요?"

방향은 맞지만, 이걸로 끝나지 않습니다

- 순수 캐시 vs tmpfs(/dev/shm) 분리 모니터링 → 이상 급증 시 "범인이 /dev/shm이다"까지는 알 수 있음
- 근데 kubectl top / Prometheus는 파드 총 메모리만 보여줄 뿐, shmem만 따로 떼어 보여주지 않음 — 대시보드를 나눠도 "어느 프로세스(PID)" 인지는 별도로 찾아야 함

lsof/proc로 원인 프로세스 찾아 kill? — 무겁고 느립니다

- lsof는 시스템 전체 프로세스의 fd 테이블을 매번 훑는 방식 — 프로세스·파일 수가 많을수록 오버헤드가 큼
- 멀티 클러스터·다수 노드에서 주기적으로 폴링하면 그 자체가 부하, 발생 시점도 예측 불가
- eBPF는 lsof처럼 매번 전체를 훑지 않고, fork/exit 커널 이벤트가 발생할 때만 가볍게 반응

+ 최선의 대응 = 캐시/tmpfs 분리 모니터링(탐지) + eBPF 커널 이벤트로 원인 프로세스 특정 + 유저스페이스에서 안전한 kill
이 발표의 orphan_guard 파이프라인이 해당 조합

"/dev/shm emptyDir에 sizeLimit을 걸어서 Kubelet이 파드만 Evict하게 하면, 굳이 eBPF 킬러를 도입할 이유가 있나요?"

네, sizeLimit + Kubelet Eviction도 유효한 방어선입니다

- 고아 프로세스가 메모리를 부풀려도, 노드 전체가 아니라 이 파드만 먼저 정리되어 다른 파드를 보호합니다

```
watchdog = ManagerWatchdog()
while watchdog.is_alive():
    r = index_queue.get(timeout=MP_STATUS_CHECK_INTERVAL)
```

orphan-guard는 파드가 아니라 좀비만 도려냅니다

- eviction은 파드 전체를 재시작 — 그 안의 멀쩡한 워커들, 학습 상태까지 다 같이 잃습니다
- orphan-guard는 고아가 된 워커만 SIGKILL — 리더와 나머지 파드는 그대로 살아있습니다
- 게다가 eviction은 왜 터졌는지 단서를 안 남기지만, orphan-guard는 어떤 프로세스가 뭘 쥐고 있었는지 로그로 남깁니다

→ sizeLimit은 node를 지키는 방어선, orphan-guard는 pod 안에서 고아프로세스와 좀비 FD만 도려내는 것 — 같이 쓰면 좋습니다.

"수천 개 파드가 뜨고 지는 클러스터에서, 모든 프로세스 exit을 감시하는 오버헤드는 어느 정도인가요?"

실측했습니다 — 사실상 측정 불가능 수준입니다

- PyTorch DataLoader 실제 처리량: orphan-guard 적용 시 오히려 +2.3%(런 간 노이즈 수준, 실질적 차이 없음)
- fork/exit을 초당 2000회 유발하는 인위적 시나리오에서도 +9.1%

이렇게 가벼운 이유는 구조 때문입니다

- 이 혹은 프로세스 생성/종료(fork·exit) 시점에만 딱 한 번 실행됨
- persistent_workers=True로 워커를 재사용하는 실제 학습 중엔, 이 혹은 할 일이 거의 없음
- (솔직하게 말씀드리자면 2~3회 반복 측정이라 엄밀한 통계는 아닙니다..)

출처: orphan-guard 자체 벤치마크(AKS Standard_D4s_v5, 2026-08-02), bench_dataloader.py / fork_stress.sh

→ 실제 학습 워크로드에서 체감 가능한 오버헤드는 없고, 조금 더 벤치마크를 쌓는 게 과제. 진행 예정..

“적용할 때 커널 제약이나 한계점은 없나요?”

커널 5.8 이상 + BTF 필요

tp_btf 계열 혹은 쓰기 때문

높은 권한 필요

보안 정책이 엄격한 클러스터는 eBPF 로더 권한 별도 승인 필요

AWS Fargate, gVisor/Kata 불가

커널을 공유하지 않는 서버리스/샌드박스 환경은 이 방식 자체가 실행 불가

프로덕션 레디 아님

dry-run 기본, --enforce는 신중히

오탐 방지 안전장치

--cgroup으로 컨테이너 스코프 제한 / --exempt PID 허용목록 / k8s 어노테이션 orphan-guard.io/exempt로 파드 단위 제외 (스코프 없으면 경고 로그로 알림)

“얼마나 자주 발생하는 문제인가요?”

정확한 발생 확률(%) 데이터는 없습니다

워커 Hang

PyTorch 이슈 #1579 → #130610
2017~2024, 7년째 반복 리포트

×

OOM-kill

같은 자원 압박 상황에서
흔히 같이 발생

=

Phantom Memory

두 조건이 겹치는 순간
몇 분 ~ 몇 시간 지속

GitHub · pytorch/pytorch #1579 “DataLoader hangs with num_workers>0” (2017), #130610 “dataloader hangs with fork and pin_memory=True” (2024)

터지면, 디버깅이 거의 불가능

- 대시보드엔 흔적이 안 남음 (RSS 기준 지표는 정상으로 보임)
- 매일 터지진 않지만, 한 번 터지면 몇 시간씩 노드를 잠식 — 재현도 어려움
- 커널 레벨 세이프티넷을 미리 깔아두는 쪽이 합리적

“PyTorch DataLoader 말고 다른 데도 적용되나요?”

핵심 패턴:

리더가 죽어도 자식이 안 죽고, 그 자식이 참조 카운트로 관리되는 커널 자원(주로 tmpfs 기반 공유 메모리)을 쥐고 있는 모든 멀티프로세스 구조.

Ray

Plasma object store가 기본적으로 /dev/shm 사용 — 워커/드라이버 비정상 종료 시 같은 f_count 패턴

joblib / scikit-learn (n_jobs=-1)

loky 백엔드에 resource_tracker가 있는 이유가 이 문제 때문 — “leaked semaphore/shared_memory” 경고 흔함 (joblib/loky #369)

Python 표준 라이브러리 자체

multiprocessing.resource_tracker(구 semaphore_tracker) 존재 이유 — CPython bpo-36867, bpo-45209

Jupyter / JupyterHub

공유 클러스터에서 큰 셀 실행 중 커널 OOM — 그 안의 multiprocessing/joblib/Dask 워커가 고아로 남는 조합이 오히려 더 흔함

단, 지금 코드가 저기까지 그대로 되는 건 아님

- orphan_guard의 3번 스텝(“shm 보유 재확인”)은 tmpfs 파일 f_count 기준으로 짜여 있음
- POSIX 세마포어·CUDA IPC 같은 다른 자원까지 커버하려면 재확인 로직만 자원별로 교체하면 되는 구조 — 파이프라인 자체는 범용. 확장 예정.