

# **쿠버네티스 인 액션**

**14장 파드의 컴퓨팅 리소스 관리**

- 14장에서 다루는 내용
  - 컨테이너의 CPU, 메모리, 그 밖의 컴퓨팅 리소스 요청
  - CPU와 메모리에 대한 엄격한 제한 설정
  - 파드에 대한 서비스 품질 보장 이해
  - 네임스페이스에서 파드의 기본, 최소, 최대 리소스 설정
  - 네임스페이스에서 사용 가능한 리소스의 총량 제한

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.1 리소스 요청을 갖는 파드 생성하기

- `request` : 컨테이너가 필요로 하는 CPU와 메모리 양
- `limit` : 사용할 수 있는 엄격한 제한
- “**파드**”의 `requests` 와 `limits` = 모든 “**컨테이너**”의 리소스 요청과 제한의 합

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.1 리소스 요청을 갖는 파드 생성하기

예제 14.1 리소스 요청을 갖는 파드: requests-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: requests-pod
spec:
  containers:
    - image: busybox
      command: ["dd", "if=/dev/zero", "of=/dev/null"]
      name: main
      resources:
        requests:
          cpu: 200m
          memory: 10Mi
```

주 컨테이너에 리소스 요청을 지정한다.

컨테이너는 200밀리코어를 요청한다  
(하나의 CPU 코어 시간의 1/5이다).

또한 컨테이너는 10Mi의 메모리를 요청한다.<sup>1</sup>

- 컨테이너 실행 시 200m 필요
- $1000m = \text{cpu} : "1"$
- CPU 요청을 지정하지 않으면 다른 프로세스에 밀려서 할당 받지 못할 수도 있음

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.1 리소스 요청을 갖는 파드 생성하기

예제 14.2 컨테이너 내의 CPU와 메모리 사용량 살펴보기

```
$ kubectl exec -it requests-pod top
Mem: 1288116K used, 760368K free, 9196K shrd, 25748K buff, 814840K cached
CPU: 9.1% usr 42.1% sys 0.0% nic 48.4% idle 0.0% io 0.0% irq 0.2% sirq
Load average: 0.79 0.52 0.29 2/481 10
PID  PPID  USER    STAT   VSZ %VSZ  CPU  %CPU  COMMAND
 1      0  root     R    1192  0.0    1  50.2  dd if /dev/zero of /dev/null
 7      0  root     R    1200  0.0    0  0.0  top
```

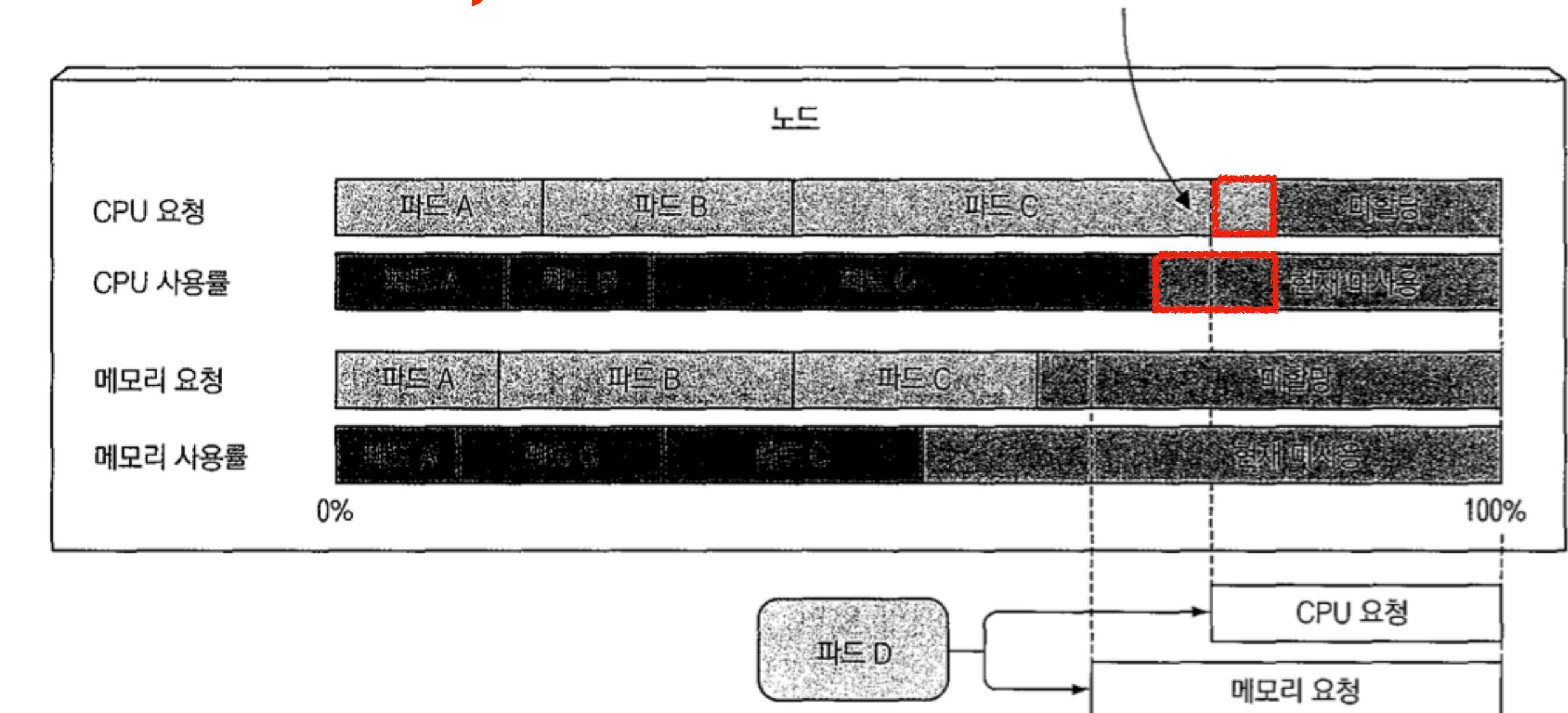
- 예제의 Minikube 가상머신에는 2개의 CPU가 있고
- 200m만을 신청했으나 실제 실행한 프로세스가 사용하고 있는 CPU는 50% (두 코어 중의 하나 사용)
- 이건 컨테이너가 사용할 수 있는 CPU양을 제한하지 않았기 때문

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.2 리소스 요청이 스케줄링에 미치는 영향

- 스케줄러는 노드에 파드 스케줄 시 리소스 요청에 있는 리소스의 최소량을 사용한다.
- 스케줄러는 “요청 사항”을 만족하는 충분한 리소스를 가진 노드만을 고려한다.
- **스케줄러는 각 개별 리소스가 얼마나 사용되는지가 아니라, “리소스 요청량의 전체 합”을 본다.**

파드 D는 스케줄링될 수 없다. 파드의 CPU 요청이 할당되지 않은 CPU 양을 초과한다.



▲ 그림 14.1 스케줄러는 실제 사용량이 아닌 요청에만 관심이 있다.

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.2 리소스 요청이 스케줄링에 미치는 영향

- 두 개의 우선순위 함수가 요청된 리소스의 양에 기반해 노드의 순위를 정함
- LeastRequestedPriority : 요청된 리소스가 낮은 노드를 선호
- MostRequestedPriority : 요청된 리소스가 가장 많은 노드를 선호 (왜?)
- 스케줄러는 이들 함수 중 “하나” 만을 이용
- MostRequestedPriority를 사용하는 이유는, 클라우드 환경에서 사용량을 일부 노드에 몰아서 일부를 비운 뒤 제거하는 식으로 비용을 절감할 수 있어서
  - 퍼블릭 클라우드 환경에서 노드 등이 계속 늘어나면 안되니까(?)

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.2 리소스 요청이 스케줄링에 미치는 영향

- 스케줄러는 각 노드에서 CPU와 메모리가 얼마나 있는지 알아야 하고,
- kubelet이 API서버에 이 데이터를 보고한다.
- 스케줄러는 allocatable 리소스양을 기준으로 결정한다.

예제 14.3 노드의 용량과 할당 가능한 리소스

```
$ kubectl describe nodes
```

```
Name: minikube
```

```
...
```

```
Capacity:
```

```
cpu: 2
```

```
memory: 2048484Ki
```

```
pods: 110
```

노드의 전체 용량

```
Allocatable:
```

```
cpu: 2
```

```
memory: 1946084Ki
```

```
pods: 110
```

파드에 할당 가능한 리소스

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.2 리소스 요청이 스케줄링에 미치는 영향

```
$ kubectl run requests-pod-2 --image=busybox --restart Never  
→ --requests='cpu=800m, memory=20Mi' -- dd if=/dev/zero of=/dev/null  
pod "requests-pod-2" created
```

- 800m을 차지하는 파드를 하나 배포해서 총 1000m을 요청 했으니, 추가 파드에 1000m이 사용가능해야 함

```
$ kubectl get po requests-pod-2  
NAME READY STATUS RESTARTS AGE  
requests-pod-2 1/1 Running 0 3m
```

- 그리고 cpu=1을 써서 1000m만 큐를 더 요청

```
$ kubectl run requests-pod-3 --image=busybox --restart Never  
→ --requests='cpu=1, memory=20Mi' -- dd if=/dev/zero of=/dev/null  
pod "requests-pod-2" created
```

- 그런데 pending 상태로 멈춰 있는 것을 볼 수 있음

```
$ kubectl get po requests-pod-3  
NAME READY STATUS RESTARTS AGE  
requests-pod-3 0/1 Pending 0 4m
```

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.2 리소스 요청이 스케줄링에 미치는 영향

예제 14.4 kubectl describe pod로 파드가 Pending 상태에 머물러 있는 이유 살펴보기

```
$ kubectl describe po requests-pod-3
```

Name: requests-pod-3

Namespace: default

Node: /

...

Conditions:

Type Status

PodScheduled False

...

Events:

```
... Warning FailedScheduling No nodes are available  
that match all of the  
following predicates::  
Insufficient cpu (1).
```

파드와 연관된  
노드가 없다.

파드가 스케줄링되지  
않았다.

CPU가 부족해  
스케줄링이 실패했다.

- CPU가 부족해 파드가 어느 노드에  
도 부합하지 않아 파드가 스케줄링 되  
지 않음
- 2000m을 요청했는데 왜?

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.2 리소스 요청이 스케줄링에 미치는 영향

예제 14.5 kubectl describe node로 노드에 할당된 리소스 검사하기

```
$ kubectl describe node
Name: minikube
...
Non-terminated Pods: (7 in total)
 Namespace     Name           CPU Requ.   CPU Lim.   Mem Req.   Mem Lim.
 -----        ---           ----       ----       ----       ----
 default       requests-pod    200m (10%)  0 (0%)    10Mi (0%)  0 (0%)
 default       requests-pod-2   800m (40%)  0 (0%)    20Mi (1%)  0 (0%)
 kube-system   dflt-http-b...  10m (0%)   10m (0%)  20Mi (1%)  20Mi (1%)
 kube-system   kube-addon-...   5m (0%)   0 (0%)    50Mi (2%)  0 (0%)
 kube-system   kube-dns-26...  260m (13%)  0 (0%)   110Mi (5%) 170Mi (8%)
 kube-system   kubernetes-...  0 (0%)   0 (0%)    0 (0%)   0 (0%)
 kube-system   nginx-ingre...  0 (0%)   0 (0%)    0 (0%)   0 (0%)

```

### Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)

CPU Requests	CPU Limits	Memory Requests	Memory Limits
-----	-----	-----	-----
1275m (63%)	10m (0%)	210Mi (11%)	190Mi (9%)

- 현재 실행 중인 파드가 총 1275m을 요청했음
- 이미 kube-system의 세 파드가 CPU를 요청
- 따라서 1000m의 요청은 노드를 overcommit하게 만들기 때문에 스케줄러는 파드를 스케줄링 할 수 없음

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.2 리소스 요청이 스케줄링에 미치는 영향

예제 14.6 다른 파드를 삭제한 후 파드가 스케줄링된다.

```
$ kubectl delete po requests-pod-2  
pod "requests-pod-2" deleted
```

```
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
requests-pod	1/1	Running	0	2h
requests-pod-2	1/1	Terminating	0	1h
requests-pod-3	0/1	Pending	0	1h

```
$ kubectl get po
```

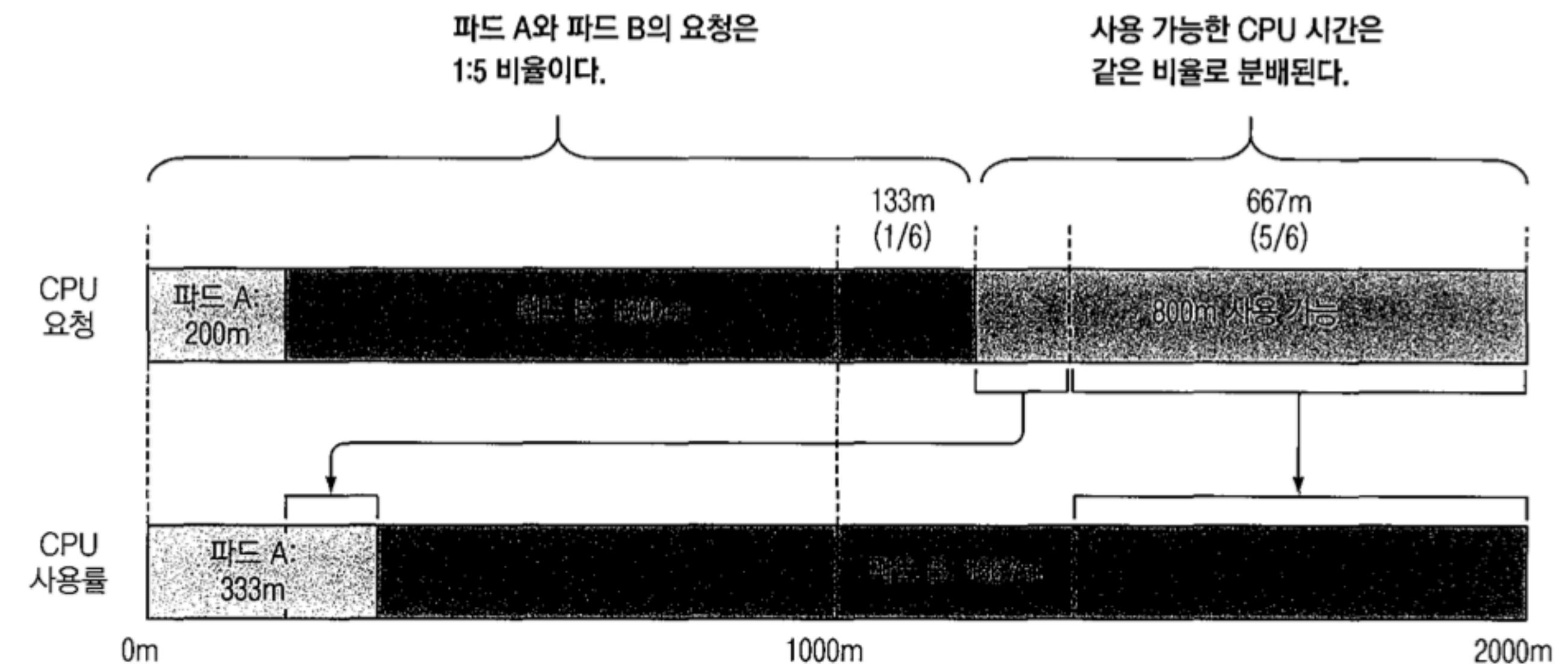
NAME	READY	STATUS	RESTARTS	AGE
requests-pod	1/1	Running	0	2h
requests-pod-3	1/1	Running	0	1h

- 이미 스케줄링 되어 있는 상태이기 때문에 파드 하나를 삭제하면
- 대기하고 있던 세 번째 파드를 스케줄링

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.3 CPU 요청이 CPU 시간 공유에 미치는 영향

- 두 개의 파드가 1:5 비율로 요청을 넣었기 때문에 미사용 CPU도 1:5 비율로 나뉨
- 근데 아무도 쓰고 있지 않다면 한 쪽이 최대로 쓸 수 있음. 그러나 한 쪽이 자기 비율만큼을 필요로 한다면 다시 조절된다.



▲ 그림 14.2 CPU 요청에 기반해 사용되지 않은 CPU 시간이 컨테이너에 분배된다.

# 14.1 파드 컨테이너의 리소스 요청

## 14.1.4 사용자 정의 리소스의 정의와 요청

- 사용자 정의 리소스를 노드에 추가하고 파드에서 사용자 리소스 요청 가능
- 사용자 정의 리소스 이름 Opaque Integer Resources -> Extended Resources (v1.8)
- Capacity 필드에 값을 추가해서 K8S가 사용자 정의 리소스를 인식하게 함
- 수량은 반드시 정수여야 함
- 그런 다음 동일한 리소스 이름과 수량을 resources.requests 필드로 지정하거나 kubectl run –requests 사용

# 14.1 파드 컨테이너의 리소스 요청

\*\* 사용자 정의 리소스로 해보기\*\*

- NVIDIA Triton Inference Server 사용시, 파드 매니페스트에는 명시적으로 GPU를 request 하지 않음 (하지만 파드를 띄우면 실제로 931MB정도를 차지함)
- 먼저 request에 gpu를 1 요청한 flowers 파드를 띄우고 그 다음 Triton 파드를 띄우면 OOM 에러 발생

```
apiVersion: "serving.kubeflow.org/v1alpha2"
kind: "InferenceService"
metadata:
  name: "flowers-sample-gpu"
spec:
  default:
    predictor:
      tensorflow:
        storageUri: "gs://kfserving-samples/models/tensorflow/flowers"
        runtimeVersion: "1.13.0-gpu"
        resources:
          limits:
            nvidia.com/gpu: 1
          requests:
            nvidia.com/gpu: 1
```

```
NAME                                         PF  READY  RESTARTS  STATUS
bert-large-predictor-default-jgd88-deployment-5b97f8d786-7wwqr  ● 2/2   5 CrashLoopBackOff
bert-large-transformer-default-gc2rq-deployment-5044800940ss00z  ● 9/2   0 Running
flowers-sample-gpu-predictor-default-wwrnh-deployment-95b4jndjq  ● 0/2   0 Running

Internal: CUDA runtime implicit initialization on GPU:0 failed. Status: out of memory

+-----+
| NVIDIA-SMI 418.87.00  Driver Version: 418.87.00  CUDA Version: 10.1 |
+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|-----+
| 0  Tesla V100-SXM2... On  | 00000000:00:1E.0 Off |          0 |
| N/A  39C   P0    40W / 300W | 15345MiB / 16130MiB | 0%     Default |
+-----+
+-----+
| Processes:                               GPU Memory |
| GPU  PID  Type  Process name           Usage      |
+-----+
```

# 14.1 파드 컨테이너의 리소스 요청

\*\* 해보기 \*\*

- 반대로 GPU를 명시적으로 요청하지 않는 Triton Inference 파드를 띄우고
- 실제로 GPU를 사용하지만 명시적으로 지정하지 않아서 요청 여력이 있는 GPU request 1이 들어가는 Flowers 파드를 띄우면 잘 실행됨

NAME	PF	READY	RESTARTS	STATUS
bert-large-predictor-default-6pl5x-deployment-6b5cb6fb64-twwh8	●	2/2	0	Running
bert-large-transformer-default-v5n96-deployment-75b558df9cpdbgr	●	2/2	0	Running
flowers-sample-gpu-predictor-default-vsds9s-deployment-7446ncjmz	●	2/2	0	Running

Allocated resources:		
(Total limits may be over 100 percent, i.e., overcommitted.)		
Resource	Requests	Limits
cpu	3185m (40%)	3 (37%)
memory	19Gi (32%)	19Gi (32%)
ephemeral-storage	0 (0%)	0 (0%)
attachable-volumes-aws-ebs	0	0
nvidia.com/gpu	1	1

Allocated resources:		
(Total limits may be over 100 percent, i.e., overcommitted.)		
Resource	Requests	Limits
cpu	2160m (27%)	2 (25%)
memory	17Gi (28%)	17Gi (28%)
ephemeral-storage	0 (0%)	0 (0%)
attachable-volumes-aws-ebs	0	0
nvidia.com/gpu	0	0

## 14.2 컨테이너에 사용 가능한 리소스 제한

### 14.2.1 컨테이너가 사용 가능한 리소스 양을 엄격한 제한으로 설정

- CPU는 압축 가능한 리소스라서 컨테이너에서 실행 중인 프로세스에 부정적인 영향을 주지 않고 컨테이너가 사용하는 CPU양을 조절할 수 있다.
- 메모리는 압축이 불가능 하다.
- 그래서 프로세스에 메모리가 주어지면 프로세스가 메모리를 해제하지 않는 한 가져갈 수 없다.
- 그래서 컨테이너에 할당되는 메모리의 최대량을 제한해야 한다.

# 14.2 컨테이너에 사용 가능한 리소스 제한

## 14.2.1 컨테이너가 사용 가능한 리소스 양을 엄격한 제한으로 설정

예제 14.7 CPU와 메모리의 엄격한 제한을 갖는 파드: limited-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: limited-pod
spec:
  containers:
    - image: busybox
      command: ["dd", "if=/dev/zero", "of=/dev/null"]
      name: main
  resources:
    limits:
      cpu: 1
      memory: 20Mi
```

- 컨테이너 내부에 실행 중인 프로세스는 CPU 1코어와 메모리 20Mi 이상을 사용 할 수 없다.

컨테이너의 리소스 제한을 지정한다.

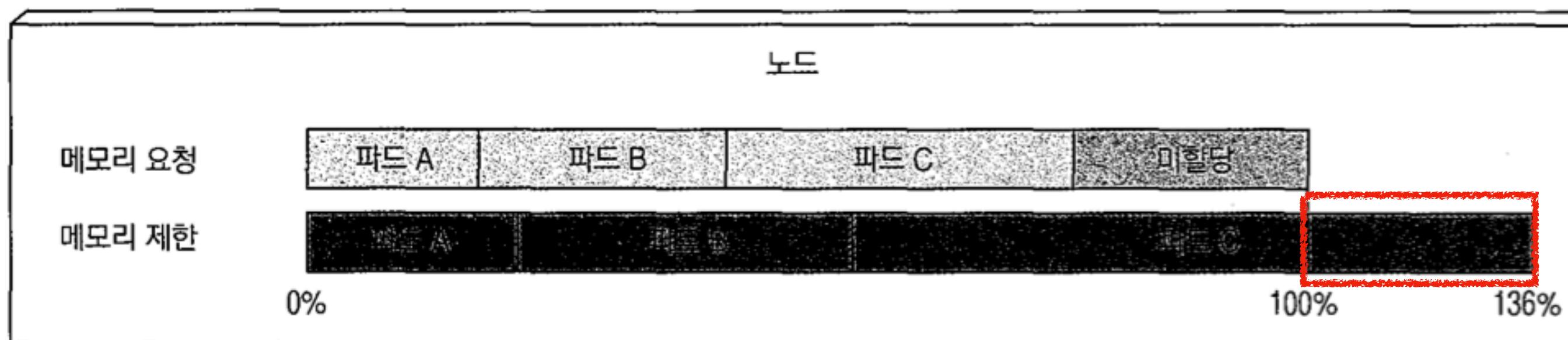
← 이 컨테이너는 최대 CPU 1코어를 사용할 수 있다.

← 컨테이너는 최대 메모리 20Mi를 사용할 수 있다.

## 14.2 컨테이너에 사용 가능한 리소스 제한

### 14.2.1 컨테이너가 사용 가능한 리소스 양을 엄격한 제한으로 설정

- 리소스 제한은 할당 가능한 리소스 양으로 제한되지 않는다.
- 노드에 있는 모든 파드의 리소스 제한 합계는 노드 용량의 100%를 초과할 수 있다.
- 리소스 제한은 overcommitted 될 수 있다.



▲ 그림 14.3 노드에 있는 모든 파드의 리소스 제한 합계는 노드 용량의 100%를 초과할 수 있다.

# 14.2 컨테이너에 사용 가능한 리소스 제한

## 14.2.2 리소스 제한 초과

- 프로세스의 CPU 사용률은 조절되므로 컨테이너에 CPU 제한이 설정돼 있으면 **프로세스는 설정된 제한보다 많은 CPU 시간을 할당받을 수 없음**
- 메모리는 CPU와 달라서 **프로세스가 제한보다 많은 메모리를 할당받으려 시도하면 OOMKilled 됨**
- 파드의 재시작 정책이 Always 또는 OnFailure로 설정된 경우 프로세스는 즉시 다시 시작하는데, 메모리 제한 초과와 종료가 지속되면 결국 CrashLoopBackOff 상태가 됨
- 그래서 발생마다 대기시간이 10초, 20초, 40초, 80초, 160초, 300초로 늘어남

```
$ kubectl get po
NAME      READY   STATUS            RESTARTS   AGE
memoryhog  0/1    CrashLoopBackOff  3          1m
```

# 14.2 컨테이너에 사용 가능한 리소스 제한

## 14.2.2 리소스 제한 초과

예제 14.8 kubectl describe pod로 컨테이너가 종료된 이유 검사하기

```
$ kubectl describe pod  
Name: memoryhog  
...  
Containers:  
main:  
...  
State:    Terminated    현재 컨테이너는 메모리 부족  
Reason:   OOMKilled    (OOM, Out Of Memory)으로 종료됐다.  
Exit Code: 137  
Started:  Tue, 27 Dec 2016 14:55:53 +0100  
Finished: Tue, 27 Dec 2016 14:55:58 +0100  
Last State: Terminated  
Reason:   OOMKilled    이전 컨테이너 또한  
Exit Code: 137    OOM으로 종료됐다.  
Started:  Tue, 27 Dec 2016 14:55:37 +0100
```

- 로그를 확인하면 OOMKilled 상태로 메모리 부족인 것을 알 수 있음
- 그렇다면 메모리 제한을 너무 낮게 설정하지 않는 것이 중요

# 14.2 컨테이너에 사용 가능한 리소스 제한

## 14.2.3 컨테이너의 애플리케이션이 제한을 바라보는 방법

```
$ kubectl create -f limited-pod.yaml  
pod "limited-pod" created
```

예제 14.9 CPU와 메모리 제한을 갖는 컨테이너에서 top 명령 실행하기

```
$ kubectl exec -it limited-pod top  
Mem: 1450980K used, 597504K free, 22012K shrd, 65876K buff, 857552K cached  
CPU: 10.0% usr 40.0% sys 0.0% nic 50.0% idle 0.0% io 0.0% irq 0.0% sirq  
Load average: 0.17 1.19 2.47 4/503 10  
PID PPID USER STAT VSZ %VSZ CPU %CPU COMMAND  
1 0 root R 1192 0.0 1 49.9 dd if /dev/zero of /dev/null  
5 0 root R 1196 0.0 0 0.0 top
```

- top 명령은 “컨테이너”가 아닌 “전체 노드”의 메모리 양을 표시
- 컨테이너 스스로는 제한을 설정해도 이를 인식하지 못함
- 예를 들어 자바 애플리케이션을 실행할 때 노드 전체 메모리를 기준으로 사용을 할 것이기에 OOMKilled가 날 수 있음

## 14.2 컨테이너에 사용 가능한 리소스 제한

### 14.2.3 컨테이너의 애플리케이션이 제한을 바라보는 방법

- CPU도 마찬가지로 노드의 모든 CPU 코어를 봄
- CPU 제한이 하는 일은 단지 컨테이너가 사용할 수 있는 CPU 시간의 양을 제한
- 어떤 애플리케이션은 시스템 CPU 수를 검색해 실행 할 작업 스레드 수를 결정하기에 문제가 생길 수 있음
- CPU 수에 의존하는 대신 Downward API를 사용해 CPU 제한을 전달하고 이를 사용하는 방법이 가능
  - 동적으로 pod에 할당 된 CPU 제한을 환경변수로 받아서 직접 제한을 건다는 뜻(?)

# 14.3 파드 QoS 클래스 이해

## 14.3.1 파드의 QoS 클래스 정의

- 쿠버네티스는 파드를 3가지 서비스 품질 Quality of Service 클래스로 분류한다.
  - BestEffort 최하위 우선순위
  - Burstable
  - Guaranteed 최상위 우선순위

# 14.3 파드 QoS 클래스 이해

## 14.3.1 파드의 QoS 클래스 정의

- BestEffort 클래스 : 우선 순위가 가장 낮은 클래스
- 아무런 리소스 요청과 제한이 없는 파드에 할당
- 이런 파드에 실행 중인 컨테이너는 **리소스 보장을 받지 못한다.**
- 전혀 CPU 시간을 받지 못하고 **다른 파드를 위해 메모리가 해제될 때 가장 먼저 종료 가능**
- 그러나 설정된 **메모리 제한이 없어서 충분할 때 원하는 만큼 메모리 사용 가능**

# 14.3 파드 QoS 클래스 이해

## 14.3.1 파드의 QoS 클래스 정의

- Guaranteed 클래스 : 리소스 요청이 리소스 제한과 동일한 파드에게 주어짐
  - CPU와 메모리에 리소스 요청과 제한이 모두 설정돼야 함
  - 각 컨테이너에 설정돼야 함
  - 리소스 요청과 제한이 동일해야 함
- 리소스 요청을 명시하지 않은 경우 기본적으로 리소스 제한과 동일하게 설정되므로
- 제한을 지정하는 것만으로 파드가 Guaranteed가 되지만 추가 리소스 사용 불가

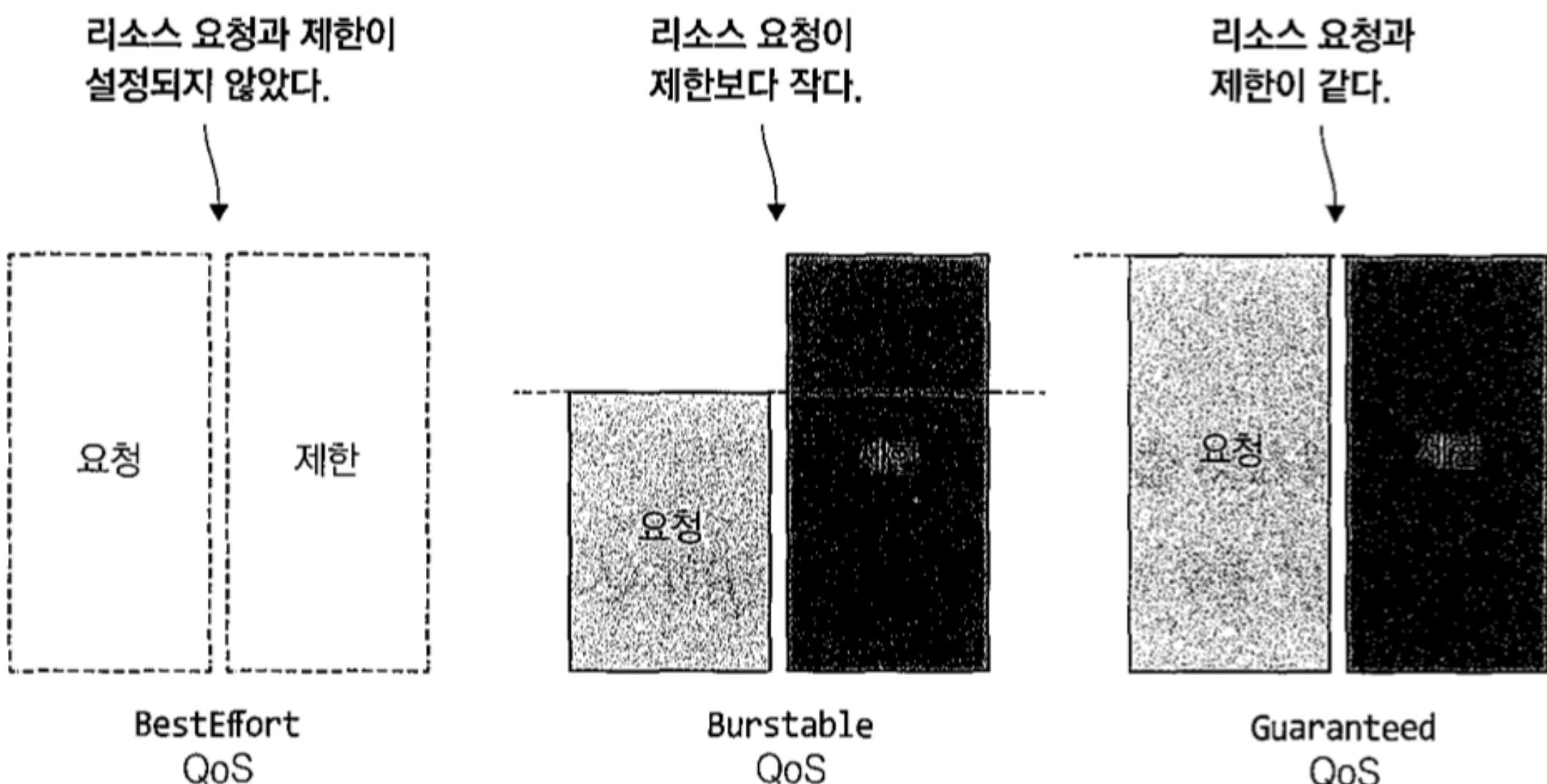
# 14.3 파드 QoS 클래스 이해

## 14.3.1 파드의 QoS 클래스 정의

- **Burstable** 클래스 : 그 밖의 남은 모든 파드
- 리소스 제한이 리소스 요청과 일치하지 않은 단일 컨테이너 파드
- 적어도 한 개의 컨테이너가 리소스 요청을 지정했지만 리소스 제한을 설정하지 않은 파드
- 컨테이너 하나의 리소스 요청과 제한은 일치하지만 다른 컨테이너의 리소스 요청과 제한을 지정하지 않는 파드
- 요청한 만큼의 리소스를 얻지만 추가 리소스를 사용할 수 있음

# 14.2 컨테이너에 사용 가능한 리소스 제한

## 14.3.1 파드의 QoS 클래스 정의



▲ 그림 14.4 리소스 요청, 제한과 QoS 클래스

▼ 표 14.1 리소스 요청과 제한에 기반으로 한 컨테이너가 하나인 파드의 QoS 클래스

CPU 요청 대 제한	메모리 요청 대 제한	컨테이너 QoS 클래스
미설정	미설정	BestEffort
미설정	요청 < 제한	Burstable
미설정	요청 = 제한	Burstable
요청 < 제한	미설정	Burstable
요청 < 제한	요청 < 제한	Burstable
요청 < 제한	요청 = 제한	Burstable
요청 = 제한	요청 = 제한	Guaranteed

# 14.3 파드 QoS 클래스 이해

## 14.3.1 파드의 QoS 클래스 정의

- 다중 컨테이너의 경우 모든 컨테이너가 동일한 QoS 클래스를 가지면 그것이 파드 QoS
- 한 컨테이너라도 다른 클래스를 가지면 Burstable
- kubectl describe pod 명령이나 매니페스트의 status.qosClass에 표시됨

▼ 표 14.2 컨테이너 클래스에서 파생된 파드의 QoS 클래스

컨테이너 1 QoS 클래스	컨테이너 2 QoS 클래스	파드 QoS 클래스
BestEffort	BestEffort	BestEffort
BestEffort	Burstable	Burstable
BestEffort	Guaranteed	Burstable
Burstable	Burstable	Burstable
Burstable	Guaranteed	Burstable
Guaranteed	Guaranteed	Guaranteed

```
QoS Class: Burstable
Node-Selectors: <none>
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
              node.kubernetes.io/unreachable:NoExecute for 300s
```

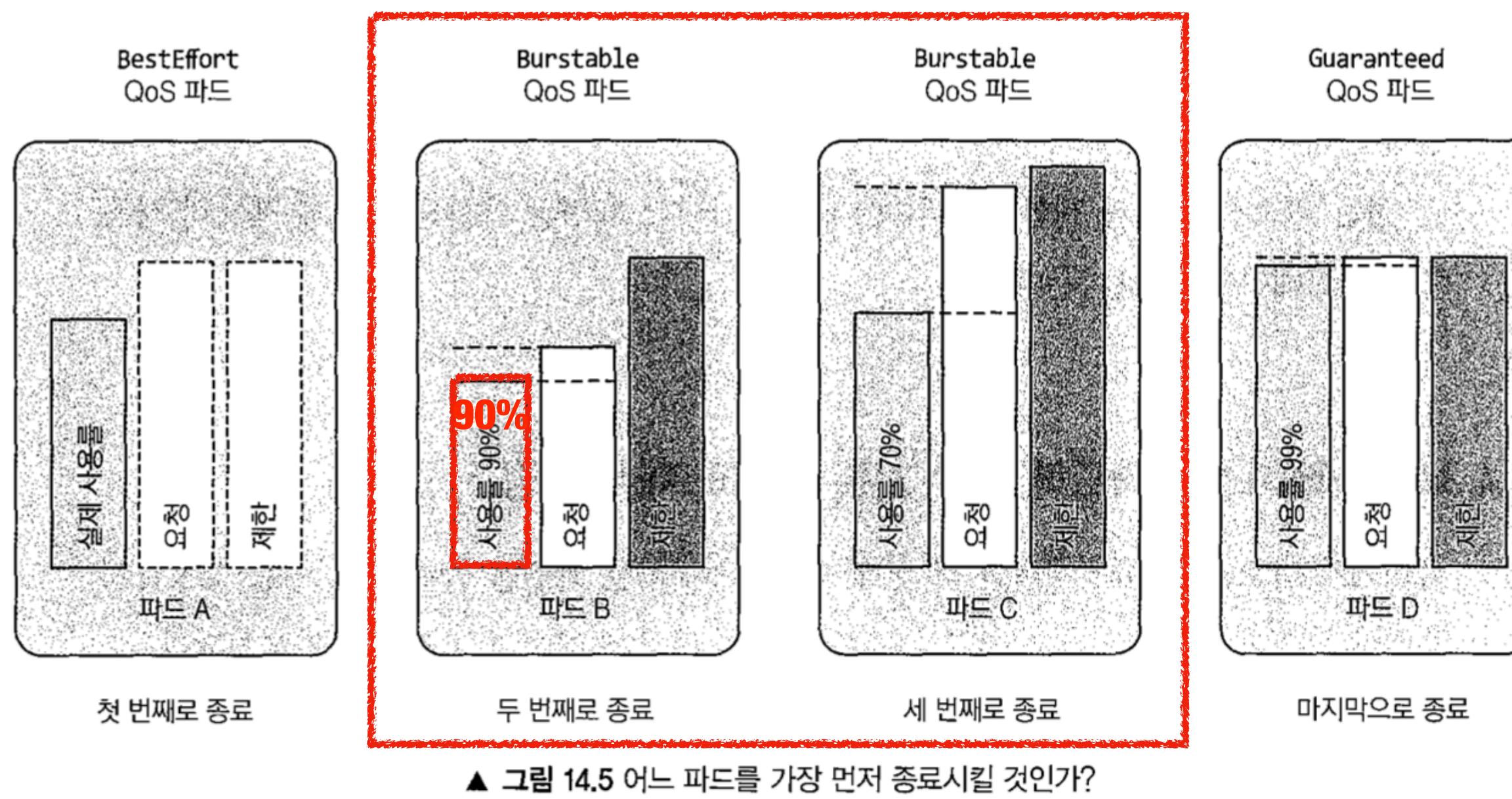
## 14.3 파드 QoS 클래스 이해

### 14.3.2 메모리가 부족할 때 어떤 프로세스가 종료되는지 이해

- 시스템이 오버커밋 되는 경우, BestEffort -> Burstable 순으로 종료
- Guaranteed 파드는 “시스템 프로세스”가 메모리를 필요로 하는 경우에만 종료

# 14.3 파드 QoS 클래스 이해

## 14.3.2 메모리가 부족할 때 어떤 프로세스가 종료되는지 이해



- 동일하게 Burstable 인 경우에는 OOM 점수가 높은 걸 종료시킴
- OOM 점수 계산 기준
  - 프로세스가 소비하는 가용 메모리 비율
- 컨테이너의 요청된 메모리와 파드의 QoS 클래스를 기반으로한 고정된 OOM 점수 조정

## 14.4 네임스페이스별 파드에 대한 기본 요청과 제한 설정

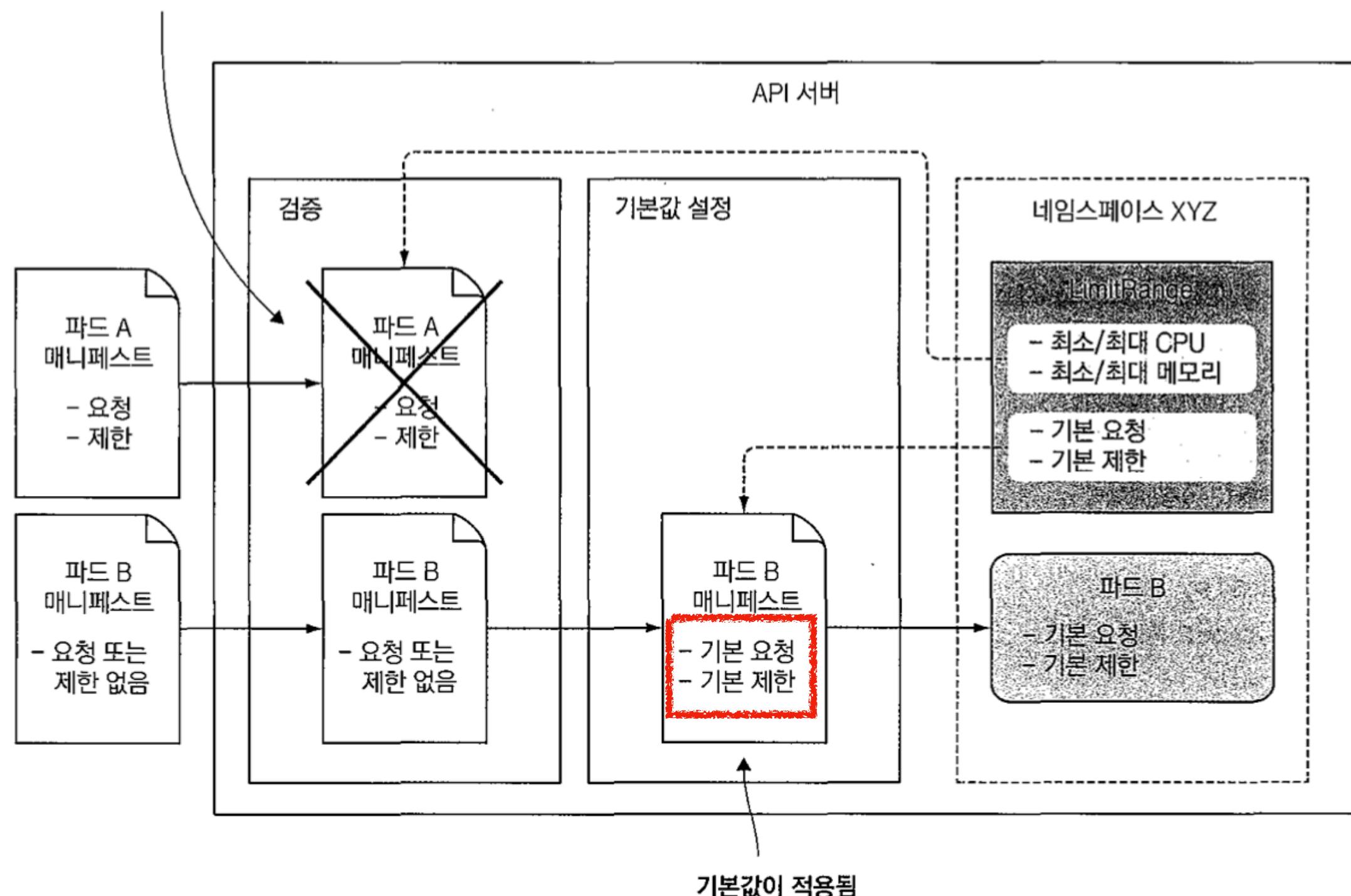
### 14.4.1 LimitRange 리소스 소개

- LimitRange 리소스는 모든 컨테이너에 리소스 요청과 제한을 설정
- 컨테이너의 각 리소스에 최소/최대 제한을 지정
- 리소스 요청을 명시적으로 지정하지 않은 컨테이너의 기본 리소스 요청을 지정

# 14.4 네임스페이스별 파드에 대한 기본 요청과 제한 설정

## 14.4.1 LimitRange 리소스 소개

리소스 요청과 제한이 최소/최댓값을 벗어났으므로 거부된다.



▲ 그림 14.6 LimitRange는 파드 검증과 기본값 설정에 사용된다.

- LimitRange 리소스는 **LimitRanger** 어드미션 컨트롤 플러그인에서 사용
- 파드 매니페스트가 API 서버에 게시되면 **LimitRanger** 플러그인이 파드 스펙 검증
- 검증이 실패하면 매니페스트는 거부
- 클러스터의 어느 노드보다 큰 파드를 생성하려는 사용자를 막을 수 있음

# 14.4 네임스페이스별 파드에 대한 기본 요청과 제한 설정

## 14.4.2 LimitRange 오브젝트 생성하기

```
예제 14.10 LimitRange 리소스: limit.yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: example
spec:
  limits:
    - type: Pod
      min:
        cpu: 50m
        memory: 5Mi
      max:
        cpu: 1
        memory: 1Gi
    - type: Container
      defaultRequest:
        cpu: 100m
        memory: 10Mi
      default:
        cpu: 200m
        memory: 100Mi
      min:
        cpu: 50m
        memory: 5Mi
      max:
        cpu: 1
        memory: 1Gi
    maxLimitRequestRatio:
      cpu: 4
      memory: 10
  - type: PersistentVolumeClaim
    min:
      storage: 1Gi
    max:
      storage: 10Gi
  
```

The diagram shows the structure of the limit.yaml file with annotations:

- Pod Level:** A callout points to the first Pod limit entry. It says "파드 전체에 리소스 제한을 지정한다." (Specifies resource limits for the entire pod).
- Container Level:** A callout points to the Container section. It says "모든 파드의 컨테이너가 전체적으로 요청(및 제한)하는 최대 CPU 및 메모리" (All containers in the pod request and limit the maximum CPU and memory).
- Request vs Limit:** A callout points to the defaultRequest section under Container. It says "컨테이너 제한은 이 줄의 아래에 지정된다." (Container limits are specified below this line).
- Default Requests:** A callout points to the default section under Container. It says "명시적으로 요청을 지정하지 않은 컨테이너에 적용되는 CPU 및 메모리의 기본 요청" (Default CPU and memory requests for containers not explicitly specified).
- No Requests:** A callout points to the min section under Container. It says "리소스 제한을 지정하지 않은 컨테이너의 기본 제한" (Default limits for containers with no explicit resource requests).
- Ratio Annotation:** A callout points to the maxLimitRequestRatio section. It says "각 리소스의 제한과 요청 간의 최대 비율" (Maximum ratio of limit to request for each resource).
- PVC Annotation:** A callout points to the PersistentVolumeClaim section. It says "LimitRange는 PVC가 요청할 수 있는 스토리지의 최소 및 최대량을 설정할 수 있다." (LimitRange can set the minimum and maximum storage amounts for PVCs).

- 전체 파드의 최소 및 최대 제한
- 리소스 요청과 제한을 명시적으로 지정하지 않은 각 컨테이너에 적용될 기본 요청과 기본 제한
- 제한 대 요청의 비율
- maxLimitRequestRatio가 4이면 200밀리코어를 요청하는 컨테이너에 801 밀리코어 이상의 제한 설정 불가
- 단일 PVC에서 요청 가능한 스토리지 양 제한 가능
- 오브젝트 유형에 따라 분할 가능
- LimitRange 오브젝트 제한을 수정시 신규 파드 및 PVC에 적용

# 14.4 네임스페이스별 파드에 대한 기본 요청과 제한 설정

## 14.4.3 강제 리소스 제한

예제 14.11 CPU 요청이 제한보다 많은 파드: limits-pod-too-big.yaml

```
resources:  
  requests:  
    cpu: 2
```

```
$ kubectl create -f limits-pod-too-big.yaml  
Error from server (Forbidden): error when creating "limits-pod-too-big.yaml":  
pods "too-big" is forbidden: [  
  maximum cpu usage per Pod is 1, but request is 2.,  
  maximum cpu usage per Container is 1, but request is 2.]
```

- LimitRange의 최대보다 큰 두 개의 CPU를 요청한 경우 에러가 뜨고
- 파드가 거부된 이유를 다 보여줌
  - 컨테이너 최대 CPU 제한은 한 개
  - 파드가 최대 CPU 제한도 한 개
  - 컨테이너 요청 합 = 파드 제한

# 14.4 네임스페이스별 파드에 대한 기본 요청과 제한 설정

## 14.4.4 기본 리소스 요청과 제한 적용

```
$ kubectl create -f ./Chapter03/kubia-manual.yaml
pod "kubia-manual" created
```

예제 14.12 파드에 자동으로 적용된 리소스 제한 검사하기

```
$ kubectl describe po kubia-manual
Name: kubia-manual
...
Containers:
  kubia:
    Limits:
      cpu: 200m
      memory: 100Mi
    Requests:
      cpu: 100m
      memory: 10Mi
```

- LimitRange 오브젝트를 설정하기 전에는 모든 파드가 리소스 요청과 제한이 없이 생성되었지만
- 이제 기본값 자동 적용
- **네임스페이스당** 파드의 기본 최소, 최대 리소스 설정 가능

# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.1 리소스쿼터 오브젝트 소개

- 리소스쿼터 ResourceQuota 오브젝트는 사용가능한 총 리소스양을 제한
- 리소스쿼터 어드미션 컨트롤 플러그인은 생성 중인 파드가 설정된 리소스쿼터 초과 확인
- 초과시 파드 생성은 거부
- 리소스쿼터 오브젝트 생성 후 신규 파드에만 영향을 미치고 기존 파드는 아님
- 네임스페이스에서 파드가 사용할 수 있는 컴퓨팅 리소스 양과 PVC가 사용하는 스토리지 양 제한
- 네임스페이스에서 만들 수 있는 파드, 클레임, 기타 API 오브젝트 수도 제한

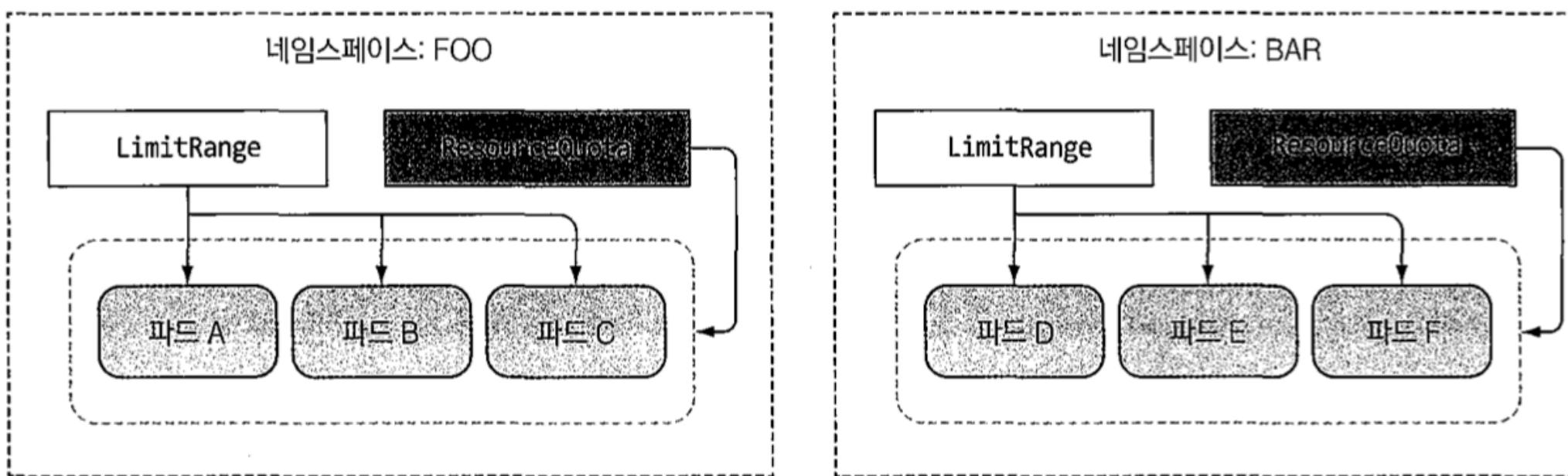
# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.1 리소스쿼터 오브젝트 소개

예제 14.13 CPU와 메모리에 관한 리소스쿼터: quota-cpu-memory.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-and-mem
spec:
  hard:
    requests.cpu: 400m
    requests.memory: 200Mi
    limits.cpu: 600m
    limits.memory: 500Mi
```

- 각 리소스 합계 대신 CPU 및 메모리 요청과 제한에 대한 별도의 합계
- 네임스페이스에서 파드 요청 최대 CPU 400m
- 네임스페이스 최대 총 CPU 600m
- 메모리는 200Mi, 500Mi
- 네임스페이스에 적용



▲ 그림 14.7 LimitRange는 개별 파드에 적용되고, 리소스쿼터는 네임스페이스의 모든 파드에 적용된다.

# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.1 리소스쿼터 오브젝트 소개

예제 14.14 kubectl describe quota로 리소스쿼터 검사하기

```
$ kubectl describe quota
Name:          cpu-and-mem
Namespace:     default
Resource       Used   Hard
-----
limits.cpu     200m   600m
limits.memory  100Mi  500Mi
requests.cpu   100m   400m
requests.memory 10Mi   200Mi
```

- 추가 파드 실행 시 요청과 제한이 Used에 추가

# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.1 리소스쿼터 오브젝트 소개

```
$ kubectl create -f ./Chapter03/kubia-manual.yaml  
Error from server (Forbidden): error when creating ".../Chapter03/kubiamanual.  
yaml": pods "kubia-manual" is forbidden: failed quota: cpu-andmem:  
must specify limits.cpu,limits.memory,requests.cpu,requests.memory
```

- 리소스쿼터는 LimitRange 오브젝트와 함께 생성되어야 함
- LimitRange가 없으면 에러 발생
- CPU 또는 메모리에 대한 쿼터가 설정된 경우, 파드에는 동일한 리소스에 대한 요청 또는 제한이 각각 설정되어야 함
- 그렇지 않으면 API 서버가 파드를 허용하지 않음
- 그래서 기본값이 있는 LimitRange가 쉬움

# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.2 퍼시스턴트 스토리지에 관한 큐터 지정하기

예제 14.15 스토리지에 대한 리소스쿼터: quota-storage.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage
spec:
  hard:
    requests.storage: 500Gi
    ssd.storageclass.storage.k8s.io/requests.storage: 300Gi
    standard.storageclass.storage.k8s.io/requests.storage: 1Ti
```

요청 가능한 스토리지의 전체 용량

ssd 스토리지 클래스에서 요청 가능한 스토리지 용량

- 네임스페이스의 모든 PVC가 요청할 수 있는 스토리지양을 500Gi로 제한
- PVC는 특정 스토리지 클래스에 동적 프로비저닝된 PV를 요청할 수 있으므로
  - 예) AWS를 쓰는 경우 EBS의 급증 방지 (?)
- 각 스토리지 클래스에 개별적으로 스토리지 큐터를 정의할 수 있게 함
- 예제에서는 SSD 와 HDD를 별도 스토리지 클래스로 구분해서 제한

# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.3 생성 가능한 오브젝트 수 제한

예제 14.16 리소스 최대 수에 관한 리소스쿼터

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: objects
spec:
  hard:
    pods: 10
    replicationcontrollers: 5
    secrets: 10
    configmaps: 10
    persistentvolumeclaims: 4
    services: 5
    services.loadbalancers: 1
    services.nodeports: 2
    ssd.storageclass.storage.k8s.io/persistentvolumeclaims: 2
```

네임스페이스에는 파드 10개, 레플리케이션 컨트롤러 5개, 시크릿 10개, 컨피그맵 10개, PVC 4개를 생성할 수 있다.

서비스 5개를 생성할 수 있으며 최대 1개의 로드밸런서 서비스와 최대 2개의 노드포트 서비스가 될 수 있다.

ssd 스토리지 클래스를 사용해 2개의 PVC가 스토리지를 요청할 수 있다.

- 네임스페이스 내 파드, 레플리케이션컨트롤러, 서비스 수 등도 제한 가능
- 예시에서는 어떤 파드인지 상관없이 최대 10개
- 레플리케이션컨트롤러는 5개
- 서비스는 5개
- 로드밸런서는 최대 1개
- 노드포트는 최대 2개
- SSD 스토리지 클래스에 한해 PVC 2개

# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.3 생성 가능한 오브젝트 수 제한

- **현재** 오브젝트 수 큐터 가능한 오브젝트
  - 파드
  - 레플리케이션컨트롤러
  - 시크릿
  - 컨피그맵
  - PVC
  - 서비스 중 로드밸런서와 노드포트

# 14.5 네임스페이스의 사용 가능한 총 리소스 제한하기

## 14.5.4 특정 파드 상태나 QoS 클래스에 대한 큐터 지정

- 지금까지 본 큐터는 파드의 현재 상태나 QoS 클래스에 관계없이 적용되었지만,
- 큐터는 큐터 범위 quota scope 로도 제한 할 수 있으며
- **BestEffort, Not Best Effort** : BestEffort 클래스 또는 나머지 두 클래스가 있는 파드에 적용 되는지
- **Terminating, Not Terminating** : 전자는 activeDeadlineSeconds 필드가 설정된 파드에 적용 후자는 아닌 경우
  - activeDeadlineSeconds 파드가 실패로 표시된 후 노드에서 활성화하도록 허용하는 시간

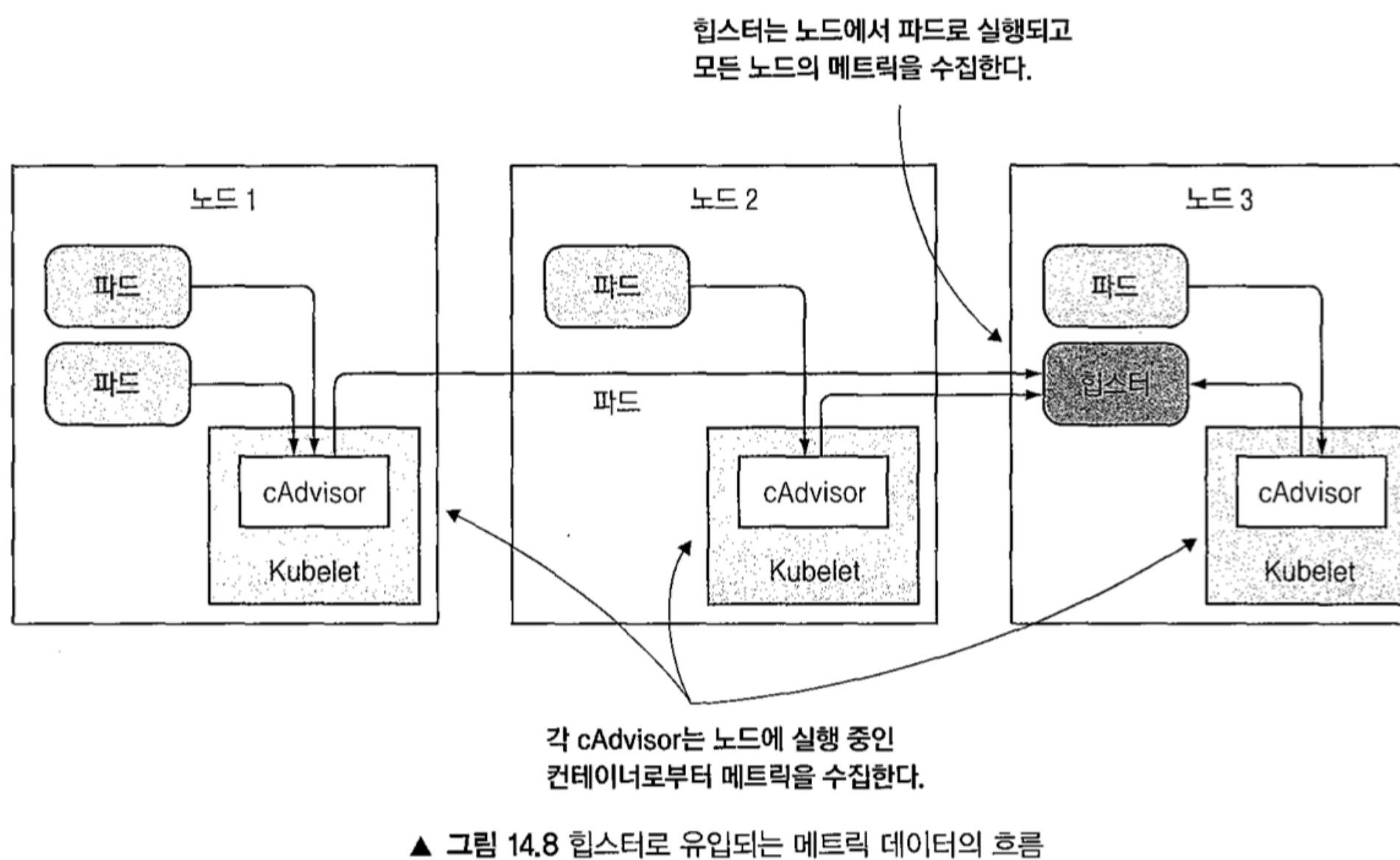
# 14.6 파드 리소스 사용량 모니터링

## 14.6.1 실제 리소스 사용량 수집과 검색

- Kubelet에는 cAdvisor라는 에이전트가 포함돼 있고,
- 이 에이전트가 노드에서 실행되는 개별 컨테이너와 노드 전체의 리소스 사용 데이터 수집
- 중앙에서 이 데이터를 수집하려면 힙스터라는 추가 구성요소를 실행해야 한다  
(최근에는 메트릭 서버 metric-server)
- 힙스터는 파드로 실행되고, K8S 서비스를 통해 노출돼 안정된 IP 주소 접속 가능
- 그러면 클러스터의 모든 cAdvisor로부터 데이터를 수집해 한 곳에서 노출

# 14.6 파드 리소스 사용량 모니터링

## 14.6.1 실제 리소스 사용량 수집과 검색



- 메트릭 데이터의 흐름
- 파드는 cAdvisor를 전혀 모름
- cAdvisor는 힙스터를 전혀 모름
- 힙스터가 모든 cAdvisor에 연결
- cAdvisor는 파드 내 프로세스와 통신하지 않고 데이터 수집

# 14.6 파드 리소스 사용량 모니터링

## 14.6.1 실제 리소스 사용량 수집과 검색

```
$ minikube addons enable heapster  
heapster was successfully enabled
```

예제 14.18 노드의 실제 CPU 메모리 사용률

```
$ kubectl top node  
NAME      CPU(cores)  CPU%  MEMORY(bytes)  MEMORY%  
minikube  170m        8%    556Mi          27%
```

예제 14.19 파드의 실제 CPU 메모리 사용률

```
$ kubectl top pod --all-namespaces  
NAMESPACE  NAME          CPU(cores)  MEMORY(bytes)  
kube-system  influxdb-grafana-2r2w9  1m          32Mi  
kube-system  heapster-40j6d        0m          18Mi  
default      kubia-3773182134-63bmb  0m          9Mi  
kube-system  kube-dns-v20-z0hq6   1m          11Mi  
kube-system  kubernetes-dashboard-r53mc  0m          14Mi  
kube-system  kube-addon-manager-minikube  7m          33Mi
```

- GKE는 기본으로 힙스터가 사용
- Minikube 사용시 추가 기능으로 사용 가능
- 그 후 kubectl top 사용 가능
- 개별 파드는 kubectl top pod
- 때때로 오류가 뜨는 경우는 메트릭을 집계하고 노출하기 때문

▶ kubectl top pod

```
Error from server (NotFound): the server could not find the requested resource (get services http:heapster:)
```

# 14.6 파드 리소스 사용량 모니터링

## 14.6.1 실제 리소스 사용량 수집과 검색

```
(base) mokpolar ➤
▶ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.6/components.yaml

clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
serviceaccount/metrics-server created
deployment.apps/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
(base) mokpolar ➤
▶ kubectl get deployment metrics-server -n kube-system

NAME           READY   UP-TO-DATE   AVAILABLE   AGE
metrics-server 1/1     1            1           7s
```

```
▶ kubectl top nodes

CPU(cores)  CPU%  MEMORY(bytes)  MEMORY%
110m        5%    917Mi          12%
82m         2%    1071Mi         7%
501m        6%    9545Mi         15%
88m         4%    561Mi          8%
```

# 14.6 파드 리소스 사용량 모니터링

## 14.6.2 기간별 리소스 사용량 통계 저장 및 분석

- top은 현재 리소스 사용량만 표시
- 오랜 기간에 대해서는 추가 도구가 필요
- GKE는 구글 클라우드 모니터링
- 다른 경우 통계 데이터 저장에는 인플럭스 DB InfluxDB
- 시각화와 분석에는 그라파나 Grafana

# 14.6 파드 리소스 사용량 모니터링

## 14.6.2 기간별 리소스 사용량 통계 저장 및 분석

