**LINUX TROUBLESHOOTING**

**1. dmesg — Kernel Message Log**

**Significance:**  
Shows messages from the Linux kernel.  
Mainly useful for hardware issues, driver problems, or low-level system events.  
It’s live, meaning it shows what the kernel is seeing right now since boot.  
  
**Why it’s useful in OpenShift:**  
Worker and master nodes in OCP are still Linux machines underneath.  
If a pod is failing because of low memory, disk errors, or network driver issues, dmesg can show you the cause.  
It’s especially useful when the problem is not inside the pod, but in the node OS or hardware.

**Examples as an OCP Engineer:**

* Node Not Ready → Check dmesg for:
* Out of Memory killer messages (OOM-killer)
* Disk I/O errors
* NIC (network interface) resets

Pod CrashLoopBackOff → If the pod is using a lot of memory or causing a kernel panic, dmesg might tell you.  
  
**Typical Commands:**  
  
dmesg | tail -n 50 # Last 50 kernel messages  
dmesg -T # Human-readable timestamps  
dmesg | grep -i oom # Search for OOM killer events  
dmesg | grep -i error # Search for errors

**Real-Life Example Using dmesg in OpenShift**  
**Scenario:**  
  
One of your OpenShift worker nodes suddenly shows NotReady when you run:  
  
oc get nodes  
Pods running on that node are crashing or getting evicted.  
  
**Step-by-step  
First check in OpenShift:**  
  
oc describe node worker-1

**You might see something like:  
Conditions:**

MemoryPressure True  
DiskPressure False  
This tells you the node is running out of memory.  
  
**SSH into the node:**  
  
ssh core@worker-1

**Run dmesg to see kernel events:**  
dmesg -T | grep -i oom

**You see:**  
  
[Mon Aug 12 15:20:11 2025] Out of memory: Kill process 27654 (java) score 991 or sacrifice child  
[Mon Aug 12 15:20:11 2025] Killed process 27654 (java) total-vm: 2058424kB, anon-rss: 1024000kB

**What this means in plain words:**  
The kernel saw that the node was running out of memory.  
It used the OOM-killer to kill a process (in this case, a Java process inside a pod) to free memory.  
  
This is why your pod crashed — not because of application bugs, but because the node ran out of RAM.  
  
**Your next action as OCP Engineer:**

* Add memory limits to that pod in the Deployment so it can’t take too much.
* Move some workloads off that node (oc adm drain worker-1).
* Increase node memory if consistently happening.

**Why dmesg was crucial here**:  
If you only looked at oc logs, you would think “the pod just died” with no clear reason.  
dmesg tells you the exact cause from the kernel’s perspective: memory exhaustion.

**What is OOM-killer?**  
OOM = Out Of Memory.

* The Linux kernel constantly monitors available RAM.
* If memory gets critically low and there’s no swap space left, the kernel must kill a process to free RAM, or the whole system will freeze.
* This is done by the OOM-killer (Out Of Memory Killer).
* The kernel picks the “most expensive” process (highest memory usage and priority score) and kills it.

**In OpenShift/Kubernetes, this often means:**  
  
A pod’s container process gets killed → pod restarts → you see CrashLoopBackOff.  
  
**Why it matters in OCP troubleshooting:**  
If pods are dying due to OOM-killer, you know the problem is node-level resource pressure, not necessarily an application bug.

**Run dmesg to see kernel events:**  
dmesg -T | grep -i oom

**Explanation of this command:**

**-T in dmesg**  
By default, dmesg shows timestamps like [12345.6789] (seconds since boot).  
  
-T = human-readable timestamps (date and time), making it easier to read.  
  
**Example:**

* Without -T:

[ 456.123] Out of memory: Kill process 27654 (java)

* With -T:

[Mon Aug 12 15:20:11 2025] Out of memory: Kill process 27654 (java)

**grep -i oom**

* grep = search for lines containing a specific word or pattern.
* -i = case-insensitive search (matches OOM, oom, Oom, etc.).
* oom = the keyword we’re looking for in the log.

**So:  
dmesg -T | grep -i oom**

**Means:**  
Get kernel messages with human-readable timestamps.  
Search for any line containing "oom", ignoring letter case.  
  
**Example in Plain Words:  
You’re basically telling Linux:**  
  
“Show me all the kernel’s human-readable messages since boot, but only the ones about running out of memory.”

**2. journalctl — Systemd Logs**

**Significance:**

* Displays logs collected by systemd on a Linux machine.
* Includes service logs, system events, and even some kernel logs.
* You can filter logs by time, service, priority, or boot session.

**Why it’s useful in OpenShift:**

* Lets you see node-level service failures (CRI-O, kubelet, NetworkManager, etc.).
* If OpenShift components on a node (like kubelet or SDN) fail, you can check why.
* It’s like oc logs but for system services running on the node OS.

**Why it matters for OpenShift Engineers**  
In OpenShift, nodes are Linux servers with system services that run the cluster components.

**If a service like kubelet or crio is down:**

* Your pods won’t start or schedule
* Your node might go into NotReady
* Networking might break

With journalctl, you can see exactly what the service reported before failing.

**Practical Example  
Scenario:**  
  
**You run:**  
oc get nodes

**and see:**  
worker-2 NotReady ...

**Step 1: SSH into the node**  
  
ssh core@worker-2

**Step 2: Check the kubelet logs**  
journalctl -u kubelet -n 50

**You might see:**  
Aug 12 16:05:42 worker-2 kubelet[2134]: Failed to start ContainerManager: failed to initialise Topology Manager  
Aug 12 16:05:42 worker-2 kubelet[2134]: kubelet cgroup driver does not match container runtime

**What happened here in simple words:**  
  
The kubelet service (which manages pods on the node) failed because its configuration didn’t match CRI-O’s configuration.  
  
That’s why OpenShift marked the node as NotReady.  
  
**Step 3: Fix the issue**  
  
Update kubelet configuration or CRI-O settings to match cgroup driver.

**First — what is a Cgroup driver?**  
Cgroups = Control Groups — a Linux kernel feature that:

* Limits CPU usage
* Limits RAM usage
* Tracks resource usage for processes

Cgroup driver = the method/tool that manages these cgroups.  
  
**Two common ones:**  
  
**systemd**

* The modern Linux init system also manages cgroups directly.
* Everything is neatly integrated with system services.
* This is the default in OpenShift.

**cgroupfs**

* Older method where cgroups are managed via a virtual filesystem (/sys/fs/cgroup).
* Still works, but less integrated with systemd.

**How this works in OpenShift**

**On a node:**

* CRI-O creates and runs containers.
* Kubelet talks to CRI-O, telling it when to start/stop containers.
* Both need to know where and how to place containers into cgroups for resource limits.

**If both use the same driver:**  
  
CRI-O and kubelet talk the same “resource management language”.  
  
**Example:** both agree “this container goes in /sys/fs/cgroup/system.slice”.  
  
**What happens if they mismatch?**  
Imagine kubelet and CRI-O are two people trying to put the same container into a “box” that controls memory:  
  
**Kubelet (systemd driver) says:**  
“This container belongs in /system.slice/container-xyz.scope”  
  
**CRI-O (cgroupfs driver) says:**  
“Nope, I’m putting it in /kubepods/burstable/pod-xyz/container-xyz”  
  
**Now:**

* CRI-O thinks it set memory limits in its cgroup.
* Kubelet looks in its own cgroup location, sees no limits, and reports errors.
* They can overwrite each other’s settings or lose track of container usage.

**The result:**  
  
Kubelet logs errors like:

* failed to create cgroup for pod: cgroup driver does not match the runtime
* Kubelet refuses to start or stops managing pods properly.
* OpenShift marks the node NotReady because kubelet isn’t functioning.

**Why OpenShift prefers systemd**  
systemd runs everything on a node — network services, kubelet, CRI-O.  
  
If both kubelet and CRI-O use systemd as cgroup driver:

* Resource management is consistent.
* It integrates better with the OS.
* Monitoring tools see accurate usage.

Red Hat even enforces this in OCP 4.x — MachineConfigs default to systemd.

**What you just did:**

* You didn’t check pod logs (oc logs) — because this is a node-level issue.
* You looked at systemd-managed service logs for kubelet, using journalctl.
* You found the real reason — cgroup driver mismatch.

**How you’d fix it in OpenShift 4.\***  
**In managed OCP:**  
  
This would usually mean updating the MachineConfig for that node pool to ensure CRI-O uses systemd.  
  
Then you’d reboot the node so the config applies.  
  
**Example to check CRI-O logs:**  
journalctl -u crio --since "30 min ago"

**==================Highlight===================**

**Key point in simple words:**

* In OpenShift, you use journalctl when you suspect the node’s operating system service is failing — like kubelet, crio, NetworkManager, ovn-kubernetes.
* **oc logs** = See logs for a pod (application level)
* **journalctl** = See logs for the node’s services (infrastructure level)
* **dmesg** = Kernel/hardware-level events on the node

**dmesg — OpenShift Usages**  
  
**Check Out of Memory (OOM) kills**  
dmesg -T | grep -i oom  
  
**Look for disk I/O errors**  
dmesg -T | grep -i "i/o error"  
  
**Find network interface resets**  
dmesg -T | grep -i eth  
  
**Recent kernel messages after a pod/node crash**  
dmesg -T | tail -n 50  
  
**journalctl — OpenShift Usages**  
  
**View kubelet service logs (pod scheduling issues)**  
journalctl -u kubelet -n 50  
  
**Check CRI-O runtime errors (container start failures)**  
journalctl -u crio --since "1 hour ago"  
  
**Follow network service logs (SDN/OVN issues)**  
journalctl -u NetworkManager -u ovnkube-node -f  
  
**See only kernel logs (node crash investigation)**  
journalctl -k -n 100  
  
**Inspect last boot logs (after node reboot)**  
journalctl -b -1

**=====================END====================**

**1. tcpdump – Network Debugging**

* Captures and shows network traffic on a node or pod.
* Useful for debugging **network issues** in OCP (e.g., pod-to-pod communication, DNS queries, API calls).

**Examples in OCP:**

1. **Debug Pod-to-Pod Communication**

tcpdump -i eth0 host <pod-ip>

→ Check if packets are reaching the pod’s network interface.

1. **Verify DNS Resolution in a Pod**

tcpdump -i eth0 port 53

→ See if DNS queries are sent to the cluster DNS service.

1. **Check API Server Connectivity from a Node**

tcpdump -i eth0 host <api-server-ip> and port 6443

→ Verify if kubelet is talking to the API server.

**2. iostat – Disk I/O Statistics - Storage Performance**

* Shows how busy your **disks** are and how fast they read/write data.
* Useful in OCP when diagnosing **storage slowness** (e.g., PVs backed by slow disks).

**Examples in OCP:**

1. **Check if a PV-backed disk is overloaded**

iostat -xz 1

→ Look for high %util (>80%) meaning disk is maxed out.

1. **Compare Read/Write Speed Between Nodes**  
    → Detect storage bottlenecks in specific worker nodes.
2. **Debug Slow Database Pod**  
    → If iostat shows slow I/O, the PV might be on slow storage.

**3. top – Real-Time Resource Usage - Live Resource Monitor**

* Shows **CPU, memory, process usage** in real time.
* In OCP, used on nodes to check if kubelet, CRI-O, or a container is hogging resources.

**Examples in OCP:**

1. **Find High CPU Processes on a Node**  
   → Spot if kubelet or cri-o is consuming abnormal CPU.
2. **Check Memory Usage During a Pod CrashLoop**  
   → See if node memory is exhausted, triggering the OOM killer.
3. **Monitor a Node in Real-Time While Scaling Pods**  
   → Confirm CPU/memory load increases as expected.

**4. ps – Process Status - Process Inspection**

* Shows running processes and their details.
* Useful in OCP for checking if specific daemons are running (e.g., ps -ef | grep kubelet).

**Examples in OCP:**

1. **Check if kubelet is Running**

ps -ef | grep kubelet

1. **Find CRI-O Container Runtime Process**

ps -ef | grep crio

1. **See All Processes for a Pod** (on node hosting it)  
   → Helps in deep troubleshooting.

**5. /proc & stat Files – System Info from Kernel - Kernel Insight**

* Linux stores live system info in /proc (a virtual filesystem).
* Example files:
  + /proc/cpuinfo → CPU details
  + /proc/meminfo → Memory usage
  + /proc/stat → CPU usage counters
  + /proc/[PID]/stat → Process stats
* In OCP, useful when debugging **performance issues** without extra tools.

**Examples in OCP:**

1. **Check CPU Info for Worker Node Scheduling Decisions**

cat /proc/cpuinfo

1. **Verify Memory Availability Without top**

cat /proc/meminfo

1. **Check Kernel Logs for OOM Events**

cat /proc/<pid>/status

→ See memory usage of a specific process

**Quick Memory Hook:**

* **tcpdump** → "See the network packets"
* **iostat** → "See disk speed & load"
* **top** → "See resource hogs live"
* **ps** → "See what’s running"
* **stat files** → "See what the kernel knows"

**Interview Tip:**  
If they ask “How would you troubleshoot a slow pod in OpenShift?” you can reply:

“First, I’d check application logs using oc logs. If nothing obvious, I’d look at node-level issues using journalctl for kubelet/CRI-O errors, top for CPU/memory pressure, iostat for storage bottlenecks, and tcpdump if I suspect network issues. If needed, I’d inspect /proc files for kernel-level stats.”

**OCP Linux Command Cheat Sheet**

*(Application → Node → Kernel)*

**1. oc logs – Application Logs in Pods**

oc logs <pod-name>

* See logs from a container inside a pod.
* First step in app-level troubleshooting.

**2. journalctl – Node Service Logs**

journalctl -u kubelet

journalctl -u crio

* View logs from **systemd services** (e.g., kubelet, CRI-O).
* Use for **node-level issues** like container runtime failures.

**3. dmesg – Kernel/Hardware Logs**

dmesg | tail

* Shows **low-level kernel events**.
* Good for disk errors, OOM kills, network card issues.

**4. tcpdump – Network Traffic Capture**

tcpdump -i eth0 port 53 # DNS traffic

tcpdump -i eth0 host <IP> # Traffic to/from IP

* Check pod-to-pod/API server/DNS communication.
* Think: “What’s actually flowing in/out?”

**5. iostat – Disk Performance**

iostat -xz 1

* %util high? Disk is maxed out.
* Helps debug **slow PVs / storage bottlenecks**.

**6. top – Live CPU/Memory Monitor**

top

* See real-time CPU, memory usage.
* Shift + P = sort by CPU, Shift + M = sort by memory.

**7. ps – Process Status**

ps -ef | grep kubelet

ps aux | grep crio

* Check if key services/processes are running.

**8. /proc & stat files – Kernel Insights**

cat /proc/cpuinfo

cat /proc/meminfo

cat /proc/<PID>/status

* /proc is a **live system info store**.
* Use to see CPU, memory, process stats directly from the kernel.

**Troubleshooting Flow in OCP:**

1. **Pod Issue?** → oc logs
2. **Service Issue?** → journalctl
3. **Hardware/Kernel Issue?** → dmesg & /proc
4. **Network Issue?** → tcpdump
5. **Storage Issue?** → iostat
6. **Resource Issue?** → top & ps

**/var/log – Where Logs Live:**

* Contains **log files** generated by the OS, services, and apps.
* Useful when journalctl isn’t enough or for historical logs (older than journal size).

**In OpenShift**

* Most **pod logs** you see via oc logs are actually stored under /var/log/pods/ on the **node** running the pod.
* **System service logs** (kubelet, CRI-O, etc.) are either in /var/log or available through journalctl.

**Note:**

* /var/log = “Log warehouse” on a Linux node.
* journalctl = “Live + archived systemd-managed logs.”
* In OCP, if you want raw logs **without oc**, you SSH to a node and check /var/log/pods or /var/log/containers.

**dmesg vs /var/log**

* **dmesg**
  + Shows **kernel ring buffer** → live, in-memory log of kernel messages since boot.
  + Focused on **low-level events**: hardware detection, driver issues, I/O errors, OOM kills.
  + Temporary — logs can disappear after reboot or when overwritten.
* **/var/log/messages or /var/log/kern.log**
  + Persistent log files that **store dmesg output + other system logs** for long-term review.
  + Managed by rsyslog or another logging daemon.
  + Good for checking **old kernel events** after a reboot.

**OpenShift Examples**

* **Pod won’t start** → oc logs
* **Kubelet keeps restarting** → journalctl -u kubelet
* **PV mount fails** → dmesg | grep sd (disk attach errors)
* **Node crashed yesterday** → /var/log/messages (check kernel panic)