Recently we have received many GPU out of virtual memory issues from some sepecific using case in Gallery, Camera, Wechat, Browser/Webview based app, or just in general monkey/aging test.
As a result, it may cause the incorrect rendering like black blocks, or native crash then app process exits abnormally.

Classic errors in logs:

Logcat log:

<snippet 1>

08-06 14:42:11.158 7553 9292 W Adreno-GSL: <sharedmem_gpuobj_alloc:1907>: sharedmem_gpumem_alloc: mmap failed errno 12 Out of memory

08-06 14:42:11.195 7553 9292 W Adreno-GSL: <sharedmem_gpuobj_alloc:1907>: sharedmem_gpumem_alloc: mmap failed errno 12 Out of memory

08-06 14:42:11.289 7553 9292 W Adreno-GSL: <sharedmem_gpuobj_alloc:1907>: sharedmem_gpumem_alloc: mmap failed errno 12 Out of memory

<snippet 2>

11-18 12:27:39.485 6414 6444 W Adreno-GSL: <sharedmem_gpuobj_alloc:2021>: sharedmem_gpumem_alloc: mmap failed errno 12 Out of memory

11-18 12:27:39.496 6414 6444 E Adreno-GSL: <gsl_memory_alloc_pure:2125>: GSL MEM ERROR: kgsl_sharedmem_alloc ioctl failed.

11-18 12:27:39.526 6414 6444 W Adreno-GSL: <sharedmem_gpuobj_alloc:2021>: sharedmem_gpumem_alloc: mmap failed errno 12 Out of memory

11-18 12:27:39.536 6414 6444 E Adreno-GSL: <gsl_memory_alloc_pure:2125>: GSL MEM ERROR: kgsl_sharedmem_alloc ioctl failed.

11-18 12:27:39.569 6414 6444 W Adreno-GSL: <sharedmem_gpuobj_alloc:2021>: sharedmem_gpumem_alloc: mmap failed errno 12 Out of memory

11-18 12:27:39.579 6414 6444 E Adreno-GSL: <gsl_memory_alloc_pure:2125>: GSL MEM ERROR: kgsl_sharedmem_alloc ioctl failed.

Kernel log:

<snippet 1>

<3>[1333.877571] (1)[9292:Thread-889]kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 7553 addr 3ab87000 pgoff 188 len 32833536 failed error -12

<3>[1333.915015] (1)[9292:Thread-889]kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 7553 addr 3ab87000 pgoff 188 len 32833536 failed error -12

<3>[1334.008835] (1)[9292:Thread-889]kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 7553 addr 3a6ed000 pgoff 185 len 32833536 failed error -12

<snippet 2>

11-18 12:27:39.632 0 0 I Kernel : <3>[76537.821739] kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 6414 addr 0 pgoff 11 len 8458240 failed error -12

11-18 12:27:39.632 0 0 I Kernel : <3>[76537.821739] kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 6414 addr 0 pgoff 11 len 8458240 failed error -12

11-18 12:27:39.632 0 0 I Kernel : <3>[76537.821739] kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 6414 addr 0 pgoff 11 len 8458240 failed error -12

<snippet 3>

<3>[347733.422879]@6[11-11 15:55:59]kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 18294 addr 0 pgoff 88 len 8454144 failed error -12 SUBSYSTEM=kgsl DEVICE=c239:0

<3>[347733.429543]@6[11-11 15:55:59]kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 18294 addr 0 pgoff 5c len 8454144 failed error -12 SUBSYSTEM=kgsl DEVICE=c239:0

<3>[347733.435406]@5[11-11 15:55:59]kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 18294 addr 0 pgoff 5c len 8454144 failed error -12 SUBSYSTEM=kgsl DEVICE=c239:0

Background knowledge:

1. In recent Gfx/kgsl driver, it enables use_cpu_map feature, so it will use the same virtual mapping on CPU and GPU. When user-mode call kgsl to alloc Gfx/gpu memory, the returned GPU address will be 0,
   then user-mode calls mmap() into kgsl driver again to get the valid GPU address.
2. When user-mode calls mmap(), it will calls to kgsl driver kgsl_get_unmapped_area()->_get_svm_area() to lookup the whole process virtual address space to find one unmapped continuous range of VMA, which can
   meet both CPU's mapping restriction and GPU's mapping restriction.
   In CPU side, the mapping restrictions are the Dalvik VM and Java Heap will occupy a huge range of VMA, also the same with dynamically loadable librarys (.so files), also other restrictions like stack, heap, page guards.
   In GPU side, there are similiar restrictions, GPU kgsl global memstore must be mapped directly, use fixed map; ION memory does not implement use CPU map, so only mapped to GPU; Secure buffer cannot be mapped
   to CPU, only mapped to GPU.
3. In long time running, due to frequently alloc/free memory and map/unmap virtual address, process's virtual address space may be badly fragmented. In such situation, even the total free space are big enough, but they
   are fragmented into small pieces, not continous, but mmap() always need a continous range free VMA in virtual address space to map. so it is very easy to cause mmap() failed for some big size map request.
4. You can read more useful info from Appendix - kgsl mmap vritual address range limitaion.

Root cause analysis:

The root cause is user-mode mmap() is failed due to kernel/kgsl driver cannot find one big enough continous range free VMA in process's virtual address space.
Here are 2 possible reasons:
#1. There is obvious memory leak in APP, which consumes too much range of vma, then cause virtual address space is used out. This is definitely app own issue, need to fix in app side.
#2. There is no obvious memory leak in APP, but the virtual address space is fragmented badly, so cannot find a big enough continuous range VMA for a new request in mmap().

For #2, recently we have a known optimization fix of CR#1049887 in GFX adreno libs, which can improve the virtual address fragmentation introduced by GFX adreno libs.
In kernel log, if you see the addr parameter is not 0 like below, it indicate the known optimization is missed, so you can apply

the CR#1049887 fix at first.

kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 7553 **addr 3ab87000** pgoff 188 len 32833536 failed error -12

Just remind CR#1049887 fix is only optimization in GFX adreno lib, cannot solve all the fragmentation issue.

You may still meet the issue even with the CR#1049887 fix, like below log:

kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 6414 **addr 0** pgoff 11 len 8458240 failed error -12

kgsl kgsl-3d0: |kgsl_get_unmapped_area| _get_svm_area: pid 18294 **addr 0** pgoff 88 len 8454144 failed error -12

How to debug?

To debug the issue for both #1 and #2 reasons, the problem process's proc_maps and kgsl_mem are very valued info to analyze, especially the last time log before process crash.

You can apply below 2 linux shell scripts to get the continuous proc_maps info and kgsl_mem info when reproducing issue.

get_kgsl_mem_info.sh >

```
adb root
adb wait-for-device
adb shell "echo 0>/proc/sys/kernel/kptr_restrict"
while true;
do
adb shell echo "**********/d/kgsl/proc/<pid>/mem start **********">>kgsl_mem_info.txt
adb shell date>>kgsl_mem_info.txt
adb shell "cat /d/kgsl/proc/<pid>/mem">>kgsl_mem_info.txt
adb shell echo "**********/d/kgsl/proc/<pid>/mem completed **********">>kgsl_mem_info.txt
sleep 1
done
```

get_proc_maps_info.sh >

```
adb root
adb wait-for-device
adb shell "echo 0>/proc/sys/kernel/kptr_restrict"
while true;
do
adb shell echo "**********/proc/<pid>/maps start **********">>maps_info.txt
adb shell date>>maps_info.txt
adb shell echo "startaddr-endaddr flags offset major:minor size node">>maps_info.txt
adb shell "cat /proc/<pid>/maps">>maps_info.txt
adb shell echo "**********/proc/<pid>/maps completed **********">>maps_info.txt
sleep 1
done
```

[note] you must replace "<pid>" with the correct value of the problem process pid

When the issue is reproduced, just provide the complete kgsl_mem_info.txt and maps_info.txt to QC GFX team to check.

How to analyze the log?

We have 2 python script to analyze the kgsl_mem info and proc_maps info directly, then get the statistics data in a simple view format.

check_kgsl_map.py - output the kgsl mem info statistics data, like below

```
Num (count) Max (KB) Sum (KB)
--------------------------------------------------------------------------------
any(0) 40 256 1420
arraybuffer 17 4096 9648
command 251 128 5976
egl_surface 6 8220 24684
gl 227 768 4956
renderbuffer 1 1904 1904
texture 1147 4160 119808
--------------------------------------------------------------------------------
Total 168396
```

If we can see some abnormal state of GPU mem usage in the kgsl mem info, then it is APP memory leak issue.
In the above example info, we can see there are too many GL Texture memory, totally 1147 textures, the maximum texture is 4M+, and totally texture memory is 119M~.
We may suspect there is GL Texture leak in APP side, then we can ask for other logs, like GL API log to confirm this.

check_proc_maps.py - output the process maps info statistics data, like below

```
Num (count) Max (KB) Sum (KB)
------------------------------------------------------------------------------------------------
/dev/kgsl-3d0 1687 4164 150488
[anon:libc_malloc] 228 21504 354816
<~snip~>
[anon:v8_heap] 26 1024 11756
[stack:11052] 1 1036 1036
[stack:11109] 1 1036 1036
<~snip~>
[stack:9835] 1 1036 1036
[stack:9990] 1 1036 1036
[stack] 1 8188 8188
anon_inode:dmabuf 8 8220 25076
------------------------------------------------------------------------------------------------
Total 1261092
```

Hole Num (count) Max (KB) Min (KB) Sum (KB)

5545 135124 0 1218236

If we can see too many entries in the proc maps info in specific usage, it may indicate some abnormal state in current process.

In the above example info, we can see there are totally 1687 entries of /dev/kgsl-3d0, the maximum size of the entry is 4M+, the total memory is 150M+, this aligned with the kgsl_mem info log.

We can see 228 entries of libc_malloc, the maximum size of the entry is 21M+, the total memory size is 354M+.

We can also see 211 entries of [stack:xxxx], where xxxx is standing for different number, each has only one entry, with fixed size 1036, it may be related to binder communication with different process.

These too many entries really cause the process virtual address space badly fragmented.

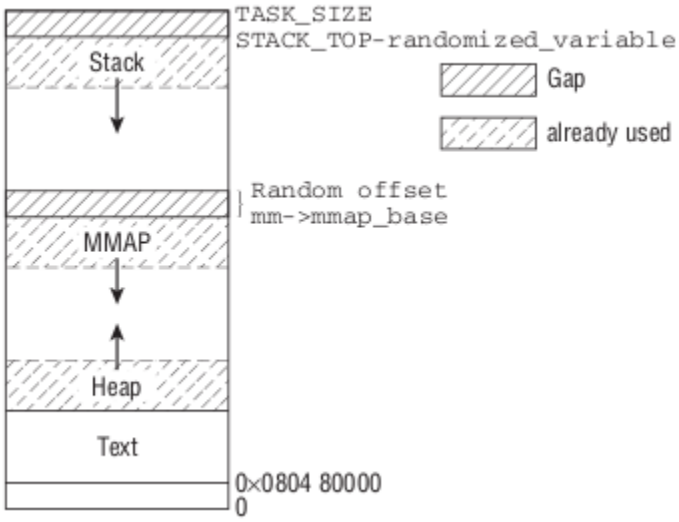
[Note] These 2 python script (check_kgsl_map.py, check_proc_maps.py) are for Qualcomm internal debug only, cannot share to OEM customer.

Appendix - kgsl mmap vritual address range limitaion

1. As you know, the process user mode virtual address is managed by the mm_struct *mm in task_struct, the virtual address layout is defined in mm_sturct *mm as below, the user mode stack is started from STACK_TOP, if PF_RANDOMIZE is set, the start address will be reduced with a random value.

Each architecture must define STACK_TOP, and most of them are defined as TASK_SIZE, it is normally 0xC0000000 in 32bit device. After randomized, the process stack start address is save in mm->start_stack, which can be read by cat /proc/xxx/stat.

As in the below figure, the stack is grown up from Top to Bottom. The range for mmap is starting from mm->mmap_base, which is initialized by calling mmap_base(), there is also one extra gap between stack and mmap range for isolation.



The mmap_base() function is as below:

```
#define MIN_GAP (128*1024*1024)
#define MAX_GAP (TASK_SIZE/6*5)

static inline unsigned long mmap_base(struct mm_struct *mm)
{
unsigned long gap = current->signal->rlim[RLIMIT_STACK].rlim_cur; // rlim_cur the default value is 8388608, which is 8M, can check it with getrlimit(RLIMIT_STACK, &limit)
unsigned long random_factor = 0;
if (current->flags & PF_RANDOMIZE)
random_factor = get_random_int() % (1024*1024);

if (gap < MIN_GAP) // MIN_GAP is used to guarantee the minimum stack size is at least 128M
gap = MIN_GAP;
else if (gap > MAX_GAP) // MAX_GAP the maximum stack size is TASK_SIZE/6*5, which is 2.5G
gap = MAX_GAP;

return PAGE_ALIGN(TASK_SIZE - gap - random_factor); // use the random_factor to avoid the the underflow of stack
}
```

The most recent code for android 8.0 is in /kernel/msm-4.9/arch/arm/mm/mmap.c

```
#ifdef CONFIG_HAVE_ARCH_MMAP_RND_COMPAT_BITS
const int mmap_rnd_compat_bits_min = CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MIN;
const int mmap_rnd_compat_bits_max = CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MAX;
int mmap_rnd_compat_bits __read_mostly = CONFIG_ARCH_MMAP_RND_COMPAT_BITS; // 16 <- 8, the original value is 8bits, the range is 1M, the new value is 16bits, the range is 256M
#endif
/* gap between mmap and stack */
#define MIN_GAP (128*1024*1024UL)
#define MAX_GAP ((TASK_SIZE)/6*5)
static int mmap_is_legacy(void)
{
if (current->personality & ADDR_COMPAT_LAYOUT)
return 1;
if (rlimit(RLIMIT_STACK) == RLIM_INFINITY)
return 1;
return sysctl_legacy_va_layout;
}
static unsigned long mmap_base(unsigned long rnd)
{
unsigned long gap = rlimit(RLIMIT_STACK);
if (gap < MIN_GAP)
gap = MIN_GAP;
```

```
else if (gap > MAX_GAP)
gap = MAX_GAP;
return PAGE_ALIGN(TASK_SIZE - gap - rnd);
}
unsigned long arch_mmap_rnd(void)
{
unsigned long rnd;
rnd = get_random_long() & ((1UL << mmap_rnd_bits) - 1);
return rnd << PAGE_SHIFT;
}
void arch_pick_mmap_layout(struct mm_struct *mm)
{
unsigned long random_factor = 0UL;
if (current->flags & PF_RANDOMIZE)
random_factor = arch_mmap_rnd();
if (mmap_is_legacy()) {
mm->mmap_base = TASK_UNMAPPED_BASE + random_factor;
mm->get_unmapped_area = arch_get_unmapped_area;
} else {
mm->mmap_base = mmap_base(random_factor);
mm->get_unmapped_area = arch_get_unmapped_area_topdown;
}
}
```

2. In GPU kgsl driver, when user mode do a mmap(), it will call into kgsl driver function kgsl_get_unmapped_area() and kgsl_mmap().

In kgsl_get_unmapped_area(), if it is not SECURE buffer, then we support use CPU map for SVM feature support, with this both CPU and GPU are using the exact same virtual address.
In _get_svm_area() called from kgsl_get_unmapped_area(), it will firstly get the svm range for GPU/kgsl side, then clamped the range with CPU's mmap requirements.

```
kernel/msm-4.9/drivers/gpu/msm/kgsl.c
static unsigned long _get_svm_area(struct kgsl_process_private *private,
struct kgsl_mem_entry *entry, unsigned long hint,
unsigned long len, unsigned long flags)
<~snippet~>
/* get the GPU pagetable's SVM range */
if (kgsl_mmu_svm_range(private->pagetable, &start, &end, // start is usually 0x00000000, end is usually 0xBF000000
entry->memdesc.flags))
return -ERANGE;
/* now clamp the range based on the CPU's requirements */
start = max_t(uint64_t, start, mmap_min_addr); // mmap_min_addris usally 0x00008000
```

end = **min_t(uint64_t, end, current->mm->mmap_base**); // current->mm->mmap_base is calculated by the function

if (start >= end)

return -ERANGE;

so we must check the **start** and **end** paramters value here for the final valid virtual address range, which is highly affected by both GPU svm range and CPU mmap limit of current->mm->mmap_base.