Kernel Supported Garbage Collection for Multithreaded Programs Written in C

CS746 Project

Pijush Chakraborty
Roll: 153050015

Problem Description:

With multi core processors in existence and the evolution of processes having multiple threads running, there has been quite a lot of theory on how to utilize them fully. The threads are normally restricted to fiddle with the same data structure or some memory region using some ***synchronization primitives*** which may eventually prove to be inefficient in terms of performance in various cases. To deal with this, many ***lockless mechanisms*** have been developed to allow multiple threads fiddling with them with ease.

The major problem with all the lockless mechanisms is ***reclamation*** or freeing up old unused memory region that should no longer be accessed by any new thread. This is required as even though the same memory region has no reference (pointer pointing to the region), the pages associated to it may be swapped out unnecessarily. The problem with reclamation is that threads have no way of knowing if any other thread still has an access to the old object.

Imagine a scenario where two threads are running in parallel and **thread-A** has access to a **memory-region[M]** created by **thread-B**. Suppose the same memory region is removed by thread-B after some work to free up memory. It will definitely lead to a segmentation fault when thread-A tries to access the memory-region[M] as the same old memory region is no longer a part of the process or the thread. This brings us to the question, when should the old memory-regions be removed? This freeing up of old unused memory region from the process address space is known as garbage collection and is a major issue for memory management in multithreaded programs written in C.

In modern languages such as Java, support for garbage collection is already implemented to save us from the burden of designing our own. This helps in writing lockless mechanisms in Java easily unlike in languages such as C. Though garbage collection should be a particular module of a language, it can be implemented by an operating system with some limitations. This projects aims to show how the Linux Kernel can keep a track of all unused memory regions in a process and can remove them periodically from the process address space. This garbage collection mechanism will help unnecessary swap outs of anonymously mapped pages which are no longer used by the process itself and will reduce the total size of the process virtual address space.

Approach:

The previous section describes the need for garbage collection in multithreaded systems. To handle this problem of un-mapping or removing old unused memory region from the process address space, the approach done is made using a form of **reference counters** for each memory region allocated. The benefit of using reference counters have an significant impact as will be seen later in this part.

Overview:

In an abstract overview the approach can be described as following. Whenever a new memory region is created, an associated **reference counter** is also created for that particular region. This per memory region is incremented whenever a thread has an access to it and similarly it is decremented as soon as a thread removes its access to the particular memory region. The per memory region reference count is tracked by the kernel and it periodically unmap-s all the region with reference count zero from the address space of the process.

Moreover, instead of un-mapping every time a memory region becomes zero, two approaches have been made to support the periodicity as described earlier is this part:

- **Approach-A:**
  The reference count of memory regions of the currently running process is checked only when the process is about to get preempted. Thus, the memory operations of unmapping are deferred to until the process has made some blocking call or is simply preempted.

- **Approach-B:**
  A particular list of unused memory region is also maintained by the kernel to make use of quick reallocation of unused memory region as soon as the process call for another memory allocation. This enables simply reusing old unused memory regions instead of removing them from the address space completely much like the slab layer.

With that said, a process (single-threaded or multi-threaded) doesn't always keep on asking for memory allocation. It can be seen that the second **reuse approach (Approach-B)** is suitable when a process is in need for a lot of memory in a small amount of time. The first **deferred approach (Approach-A)** is suitable when a process is at present happy with whatever memory it has and simply wants to free itself from old unused regions. This brings us to the conclusion to combine both the approaches in order to use the deferred removal of memory regions and also to reuse unused regions when the process is in demand for memory.

The approach described above with the use of reference counters has a significant similarity with **'Hazard-Pointers'** approach to reclaim memory from lockless data structures. In the Hazard pointers, a thread may keep a special pointer (hazard pointer) to any particular memory region to signify and inform the other running threads that since it has a hazard pointer to that memory region, no one should remove that region until the hazard pointer is removed. Similarly, reference counter can be thought as a hazard pointer to that memory region which is simply removed when the reference count becomes zero. Hazard pointers has proved to work well with lockless objects and with the similarity in mind, the reference count approach has proved itself as a worthy competitor as can be seen later in this report.

Design and Implementation:

The above approaches have been coupled in the design of **'kernel supported garbage collector'**. The first step in the design procedure is to make a framework for the userspace library and its functionality. Moving ahead with the userspace library, the following two functions are provided in **'gc.h' header file** which the user process should include in order to user the garbage collector.

The userpace library includes the following few functions:

gc_reg():
This must be called by the user process to register for garbage collection so that the kernel can then keep track of the process and free memory accordingly. This enables the kernel to build and maintain corresponding data structures for the process.

gcalloc(void **ptr,size_t len):
This function must be used to allocate a memory region of size len bytes and assign the pointer *ptr to it. The process can simply call **gcalloc(&ptr_a,1024)** to allocate a size of 1024 bytes and assign ptr_a to point to it. This also changes a few kernel data strucutres that will be discussed in the next part.

gcassign(void **ptr_a, void **ptr_b):
This function allows the pointer *ptr_a to point to the same region as *ptr_b. This also changes some appropriate **kernel data structures** for proper maintaining for the reference counts.

Per Process Data Structures:

The next step in the design procedure is to build the appropriate kernel data strucutres as discussed above, that the kernel will use to maintain the reference count for each process. Since it is a multithreaded processes and multiple threads share the same address sapce, the reference count data structure is built as a per process list with its list head(**list_head mm_list**) inside the mm_struct of each process. Each node of the list has the following structure:

```
struct mm_region{
        unsigned long start;        // start address of memory region
        size_t len;                 // length of memory region
        atomic_t mm_cnt;            // region reference count
        spinclock_t mm_lock;        // used for synchronization
        struct list_head list;      // list head
}
```

The list contains various other information and has various synchronization details so that multiple threads accessing the same per process list will not cause any problem. The list details are examined in detail in the next section. This enables the threads sharing the same mm_struct to refer to the same list for memory region reference count manipulations.

Garbage Collector Module:

The next step in the design is how to enable the actual garbage collection. To make things work, a **module 'gc.ko'** is used which when loaded enables the kernel to support garbage collection and when unloaded removes the feature. If the garbage collector is loaded, the user process using the garbage collector code can work without worrying about reclamation. But if the collector is not enabled, the same userprocess code will run normally as expected.

Interaction with System Calls:

To make the final step work some system calls have been added and the userprocess library simply uses the system calls to make garbage collection possible.

Lets browse through the **system calls** added to the kernel for maintaining the per process reference count data structures.

gc_reg():
Enables garbage collection for the process after initializing the per process reference count list mm_region and sets the per process **enable_gc integer** variable to 1 to allow the kernel to use garbage collector on this process.

gc_inc(void *ptr):
Increases the reference count of the memory region pointed to by 'ptr' in the per process reference count list. The system call implementation is discussed later in this part.

gc_dec(void *ptr):
Dectreases the reference count of the memory region pointed to by 'ptr'.

gc_add(void *ptr):
Enables garbage collection for the memory region pointed to by ptr and adds it to the per process reference count list described above.

Now, the **module 'gc.ko'** has all the code implemented for maintaining the per process reference count data structures. So, to make the module 'gc.ko' enable garbage collection, **a hook has been added to all the system calls** so that if the module is loaded the associated function is called. The above system calls doesn't really implement any code except for simply calling checking if the module is loaded and then calls the associated function. This provides a lot of flexibility as the module code may be changed anytime without worrying about other implementation in the kernel source.

Periodically Calling Garbage Collector:

The final step is simply to call the garbage collector periodically. The garbage collector code is defined in the module 'gc.ko' as gc_sync. To use the collector and finalize the design so that the process is freed from unused memory periodically, the following cases have been put in place.

- **A hook has been placed at 'schedule()'**, so that whenever the current task is going to be preempted it can call the garbage collector and free itself(the process) from memory burden. The hook has been added at the starting of schedule(), before the address space is switched. The collector simply checks for old memory regions with reference count as 0 and then calls **vm_munmap(..)**. Now, since after the process exits, schedule() is also called but the current->mm is no more valid, a check is done to ensure this doesn't cause any problem.

- Inside **do_mmap(..)**, before creating a new vma region for mapping, the memory regions are checked for any zero reference count(old unused regions) so that the same vma region can be reused.

The implementation had a lot of **synchronization mechanism** to have multiple threads working together on the same node. Since the reference count list can be accessed any time, no locks are provided for simple list iterations. But for updating, **fine grained spin_locks** are provided for each node. Moreover to add a new node, another lock has been declared in mm_struct which is to be used while adding a new node. Normal readers of the list are not harmed while adding a new node or updating since the readers view them as atomic operations.

Experiments:

To test the implementation of 'kernel supported garbage collection', **pthread** library is used to create multiple threads that share the same memory region. The user process takes the **no of threads** to create and the **number of operations** it should perform. The ‘number of operations’ is basically the no of read and update operations performed.
For example if the following code is run: **[$ ./a.out 12 1200]**, then 12 threads will be created and they will together perform 1200 operations. The operations are discussed below.

The garbage collector must be loaded with **[$ insmod gc.ko mmap_gc_enable=1 sync_thresh=15]** where **mmap_gc_enable** is by do_mmap(..) to decide if the old regions should be reused and **sync_thresh** is used by gc_syn() garbage collector to defer the collection after 15 successive calls to it.

Suppose a shared memory region is being pointed by a point **'ptr'** and all threads gets to access it. Now, the following operations will be done on **'ptr'** in an alternating manner:

- A particular thread will issue **gcalloc(&ptr, sizeof(struct node))** which will now make the old memory region pointed by ptr inaccesable using ptr.

- A particular thread will issue **gcasign(&local_ptr,ptr)** which will assign the per thread local pointer to point to the shared memory region pointed by **'ptr'**. If this is done, even if 'ptr' is now pointing to some other region, the old memory region must not be freed.

Only after all the threads are no longer pointing to a particular memory region, the region can be freed. Now, if garbage collection in not enabled, the per process address space size must increase linearly. Moreover, if garbage collection is enabled, the memory used up by the process must stay constant.

The following comparison graphs shows the memory allocated to a process under both garbage collections enabled and disabled. The tests have been performed under various situations and can be seen bellow. A shell script is run which periodically after every 2 seconds finds the current address space size of the process (**pmap pid**) and this is used to make the dataset provided along with the report.

The following figure **(Fig 1)** shows the comparison when number of threads is 4 and number of operations performed is 200.

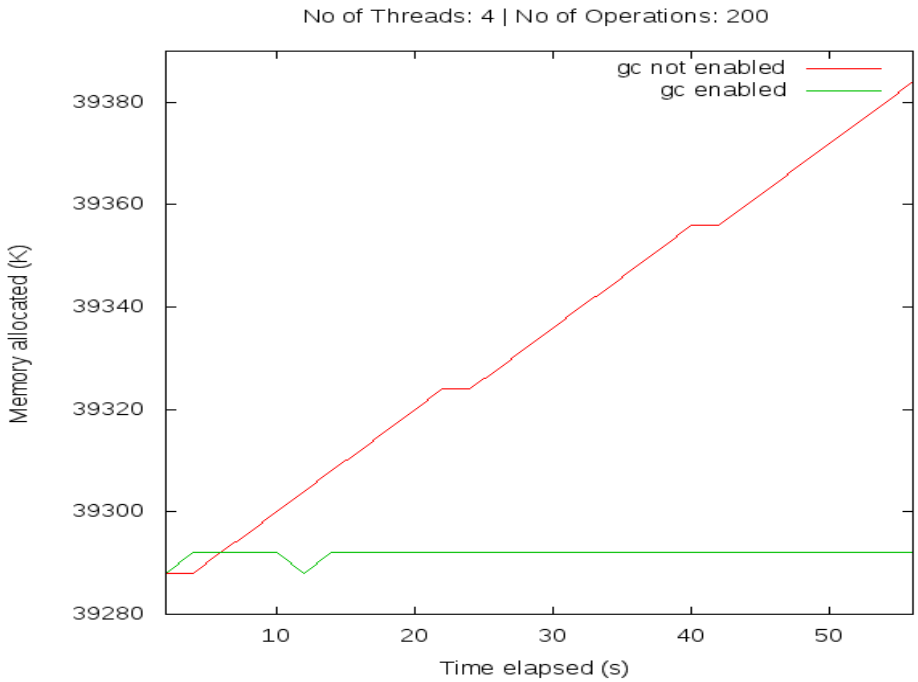

Fig 1: Comparison with no of 4 threads

As can be seen the memory allocated without garbage collection enabled grows linearly while the memory allocated almost remains constant with garbage collection enabled. The rest of the graphs show the same comparison after varying the number of threads and number of operations.

The following figure **(Fig 2)** shows the comparison when number of threads is 8 and number of operations performed is 600.

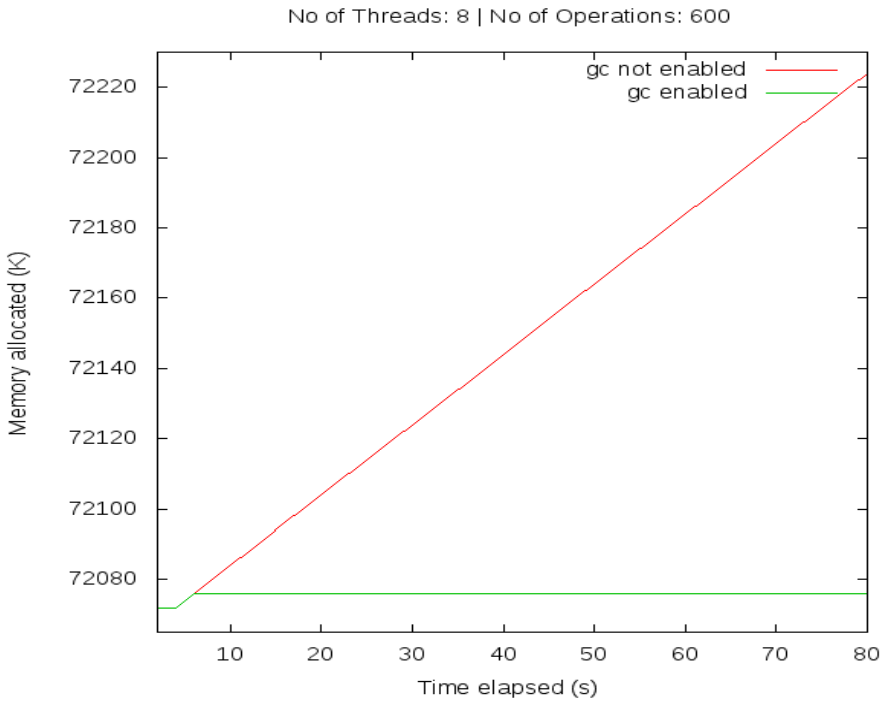

Fig 2: Comparison with no of 8 threads

The following figure **(Fig 3)** shows the comparison when number of threads is 12 and number of operations performed is 1200.

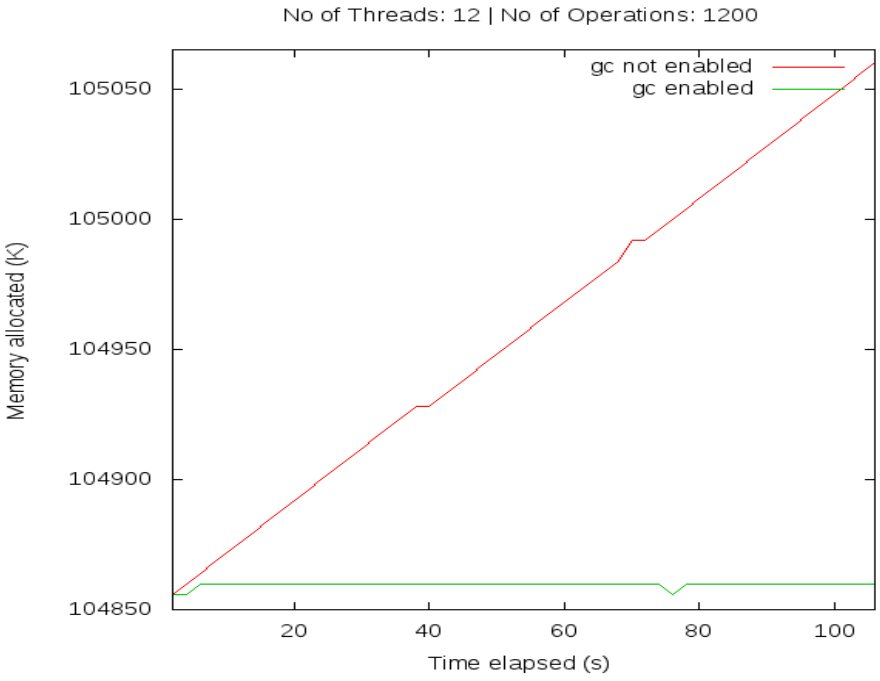

Fig 3: Comparison with no of 12 threads

Thus, from the results it can be seen that the garbage collector thus proves to be an efficient solution for the memory reclamation problem in multithreaded programs.