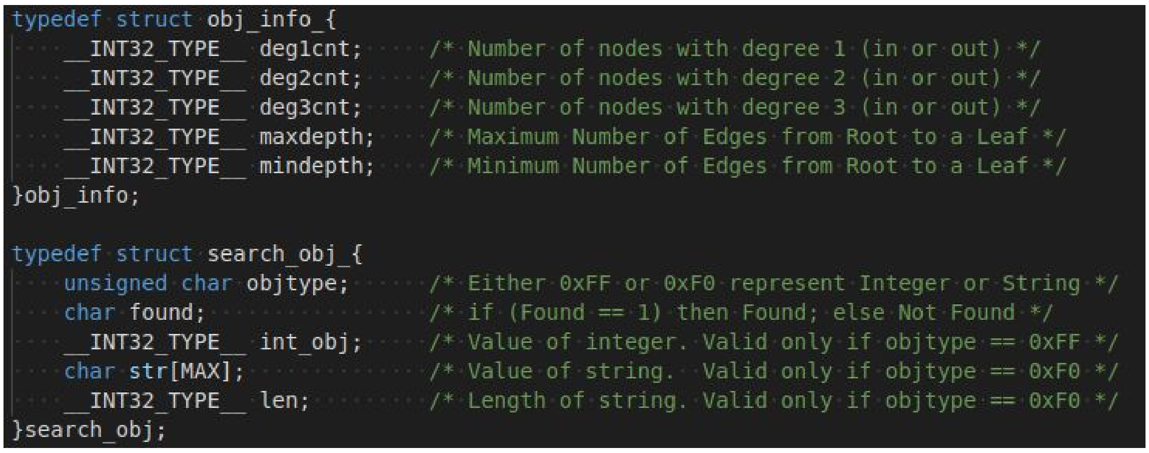
LOADABLE KERNEL MODULE FOR SORTED STORAGE IN BST

The LKM builds a binary search tree inside the kernel space for each userspace process.
The LKM also support various ioctl() call to set and retrieve various configurations and internal information respectively. Here, upon insertion of this LKM, it will create a file at the path **/proc/bst_insert**. This path will be world-readable and writable.
A userspace program will interact with the LKM through this file. A userspace program can fire ioctl() system any time it wants.

The definition of structures are as follows:



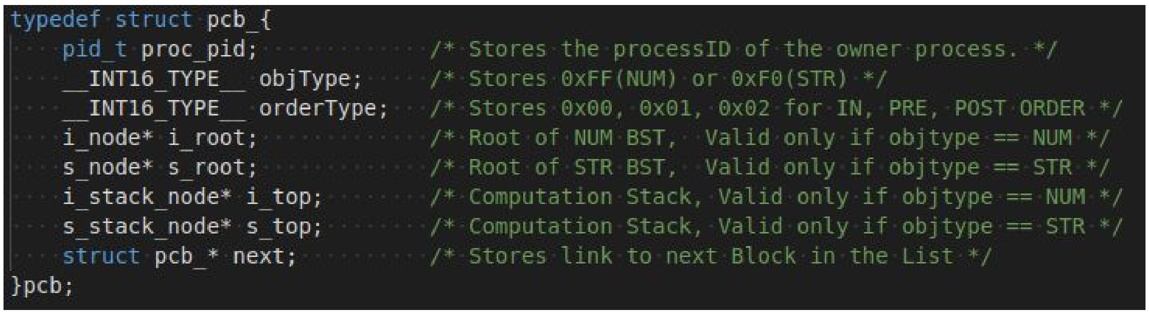
```c
typedef struct obj_info_{
    __INT32_TYPE__  deg1cnt;     /* Number of nodes with degree 1 (in or out) */
    __INT32_TYPE__  deg2cnt;     /* Number of nodes with degree 2 (in or out) */
    __INT32_TYPE__  deg3cnt;     /* Number of nodes with degree 3 (in or out) */
    __INT32_TYPE__  maxdepth;    /* Maximum Number of Edges from Root to a Leaf */
    __INT32_TYPE__  mindepth;    /* Minimum Number of Edges from Root to a Leaf */
}obj_info;

typedef struct search_obj_{
    unsigned char objtype;       /* Either 0xFF or 0xF0 represent Integer or String */
    char found;                  /* if (Found == 1) then Found; else Not Found */
    __INT32_TYPE__  int_obj;     /* Value of integer. Valid only if objtype == 0xFF */
    char str[MAX];               /* Value of string.  Valid only if objtype == 0xF0 */
    __INT32_TYPE__  len;         /* Length of string. Valid only if objtype == 0xF0 */
}search_obj;
```

A userspace process interacts with the LKM in the following manner only.

1. It will open the file (**/proc/bst_store**) in read-write mode and obtain the file descriptor.
2. Userspace process calls the ioctl() system call with command PB2_SET_TYPE and appropriate value to set the expected object type in the kernel.
3. Insert object using the write() system call on the file descriptor as many time as want. However, there is no bound on the number of times the user program can call the write() system call. LKM verifies the arguments in write() and adds a new node in the BST.
4. To get the object stored in Kernel Space BST, the userspace program needs to perform a read() system call on the file descriptor. Every read() system call returns a single object from the BST. Consecutive read system calls return consecutive nodes as per the order set by ioctl() system call with command PB2_SET_ORDER. For example, if a userspace program wants to sort the object then it has to insert the objects in the LKM, then calls the ioctl() system call with command PB2_SET_ORDER and value 0x00. After that, it performs consecutive read system call to get the sorted list.
    - The first read() system after a write() system call or an ioctl() with command PB2_SET_ORDER, **returns the first node in the list** (i.e. the root in preorder, smallest element in inorder and postorder).
    - The read system call **returns 1** in case of successful execution. However, it **returns 0** after it finishes the traversal of the tree.
    - A read system call **returns 0** if there is no node.
5. When all the operations are done, the userspace process closes() the file and LKM releases() all the resources allocated for the process.

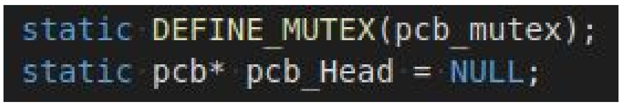
LKM should be able to handle concurrency and separate data from multiple processes.

We handle this by using a process_control_block - esque structure for each process and maintaining a linked list of pcb, with each pcb holding the required data for that particular process. This takes care of the data separation.

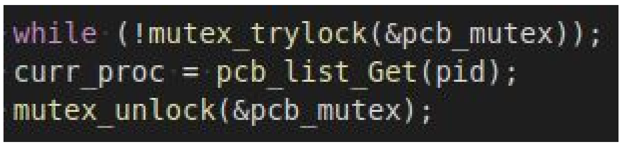


```c
typedef struct pcb_{
    pid_t proc_pid;             /* Stores the processID of the owner process. */
    __INT16_TYPE__ objType;     /* Stores 0xFF(NUM) or 0xF0(STR) */
    __INT16_TYPE__ orderType;   /* Stores 0x00, 0x01, 0x02 for IN, PRE, POST ORDER */
    i_node* i_root;             /* Root of NUM BST,  Valid only if objtype == NUM */
    s_node* s_root;             /* Root of STR BST,  Valid only if objtype == STR */
    i_stack_node* i_top;        /* Computation Stack, Valid only if objtype == NUM */
    s_stack_node* s_top;        /* Computation Stack, Valid only if objtype == STR */
    struct pcb_* next;          /* Stores link to next Block in the List */
}pcb;
```

Multiple processes access the process control block list simultaneously and hence there is an inherent need for mutual exclusion in this shared data, hence we use MUTEX_LOCKS to take care of this issue.

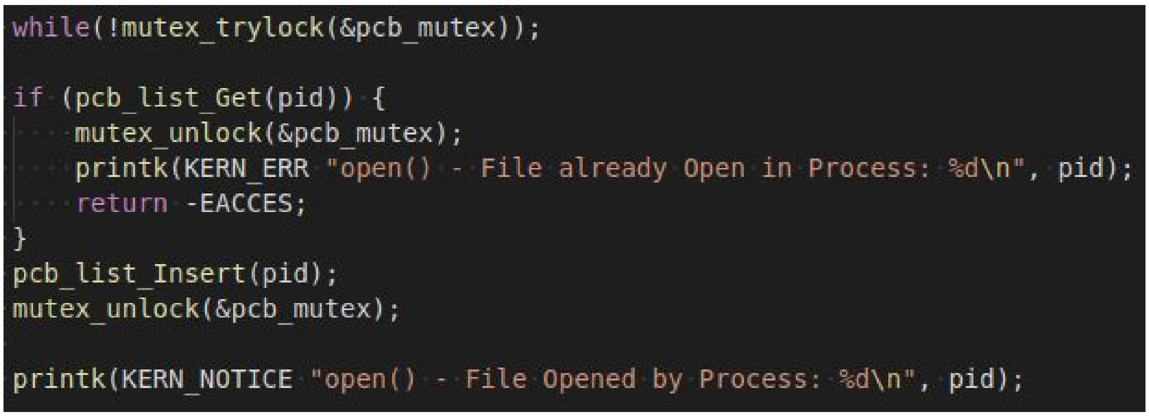


```c
static DEFINE_MUTEX(pcb_mutex);
static pcb* pcb_Head = NULL;
```

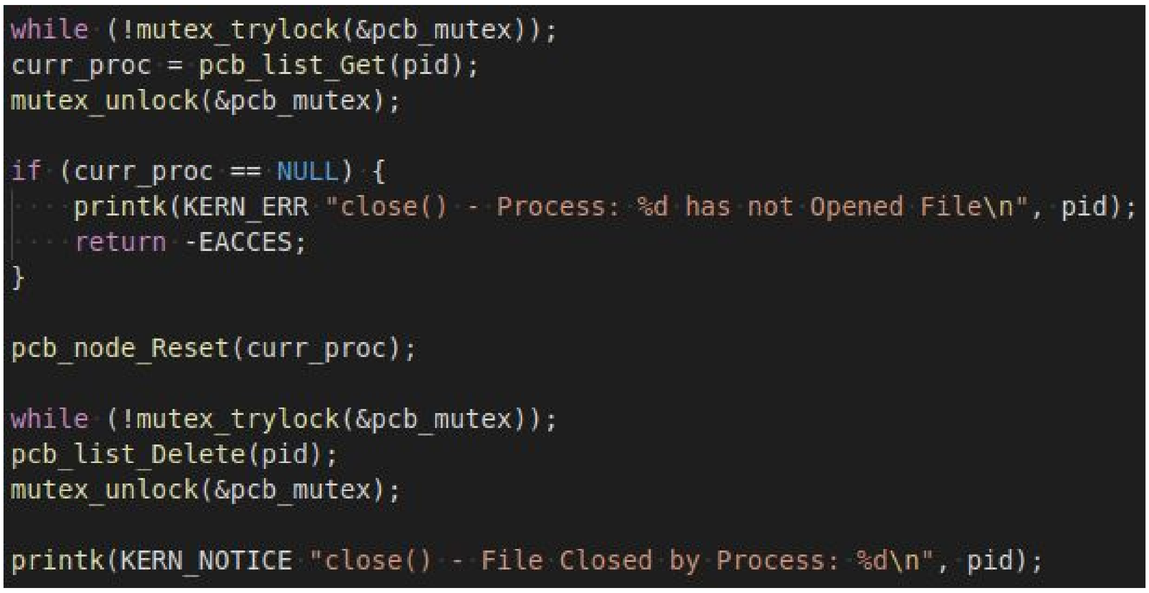


```c
while (!mutex_trylock(&pcb_mutex));
curr_proc = pcb_list_Get(pid);
mutex_unlock(&pcb_mutex);
```

Also, no userspace program can open the file more than once simultaneously. However, the userspace process should be able to reset the LKM (for the said process) by reopening the file.



```c
while(!mutex_trylock(&pcb_mutex));

if (pcb_list_Get(pid)) {
    mutex_unlock(&pcb_mutex);
    printk(KERN_ERR "open() - File already Open in Process: %d\n", pid);
    return -EACCES;
}
pcb_list_Insert(pid);
mutex_unlock(&pcb_mutex);

printk(KERN_NOTICE "open() - File Opened by Process: %d\n", pid);
```



```c
while (!mutex_trylock(&pcb_mutex));
curr_proc = pcb_list_Get(pid);
mutex_unlock(&pcb_mutex);

if (curr_proc == NULL) {
    printk(KERN_ERR "close() - Process: %d has not Opened File\n", pid);
    return -EACCES;
}

pcb_node_Reset(curr_proc);

while (!mutex_trylock(&pcb_mutex));
pcb_list_Delete(pid);
mutex_unlock(&pcb_mutex);

printk(KERN_NOTICE "close() - File Closed by Process: %d\n", pid);
```

IOCTL() Command descriptions

Command	Description
PB2_SET_TYPE	This command **initializes the LKM** with the expected object type. If LKM is already initialized for the current process, this **command resets the LKM and reinitializes again**. It **expects a pointer to char as value** where the pointer points to a single character which stores value either **0xFF** or **0xF0**. Here 0xFF means objects are 32bit integers and 0xF0 means null-terminated string with a maximum length of 100 bytes. Invalid value or inaccessible pointer causes *error* and sets error number to **EINVAL**. On *success*, it **returns 0**. No other operation can be performed (including read or write) before performing this system call. Every other system call on the LKM returns error with code **EACCES**.
PB2_SET_ORDER	This command **set the order** (i.e. inorder (*default*), preorder or postorder) of output and **resets the output cursor to the root** of the BST. It **expects a pointer to char as the value** like PB2_SET_TYPE. However, the value of the character can be 0x00, 0x01 or 0x02 which represents inorder, preorder or postorder respectively. Invalid value causes an *error* in LKM and sets error number to **EINVAL**. On *success*, this command **returns 0**.
PB2_GET_INFO	This command returns the **information about the nodes of the tree**. PB2_GET_INFO command **expects a pointer to a structure object** of type struct obj_info as the value. On *success*, LKM **returns the size of the BST** and fills the various fields of the pointer passed by the process. On *error*, LKM sets **appropriate error no**.
PB2_GET_OBJ	This command **searches an object in the BST**. It **expects a pointer to an object** of type search_obj as value. This function **always returns 0**. However, it fills the field **found** in the **search_obj** accordingly.

The definition of ioctl() commands are as follows:

```
#define  PB2_SET_TYPE  _IOW(0x10,  0x31,  int32_t*)
#define  PB2_SET_ORDER  _IOW(0x10,  0x32,  int32_t*)
#define  PB2_GET_INFO  _IOR(0x10,  0x33,  int32_t*)
#define PB2_GET_OBJ _IOR(0x10, 0x34, int32_t*)
```