MCAC403 Advance Operating System

MANIPULATION OF THE PROCESS ADDRESS SPACE

Presented by :
Subham
Roll No. 58
MCA IV Sem

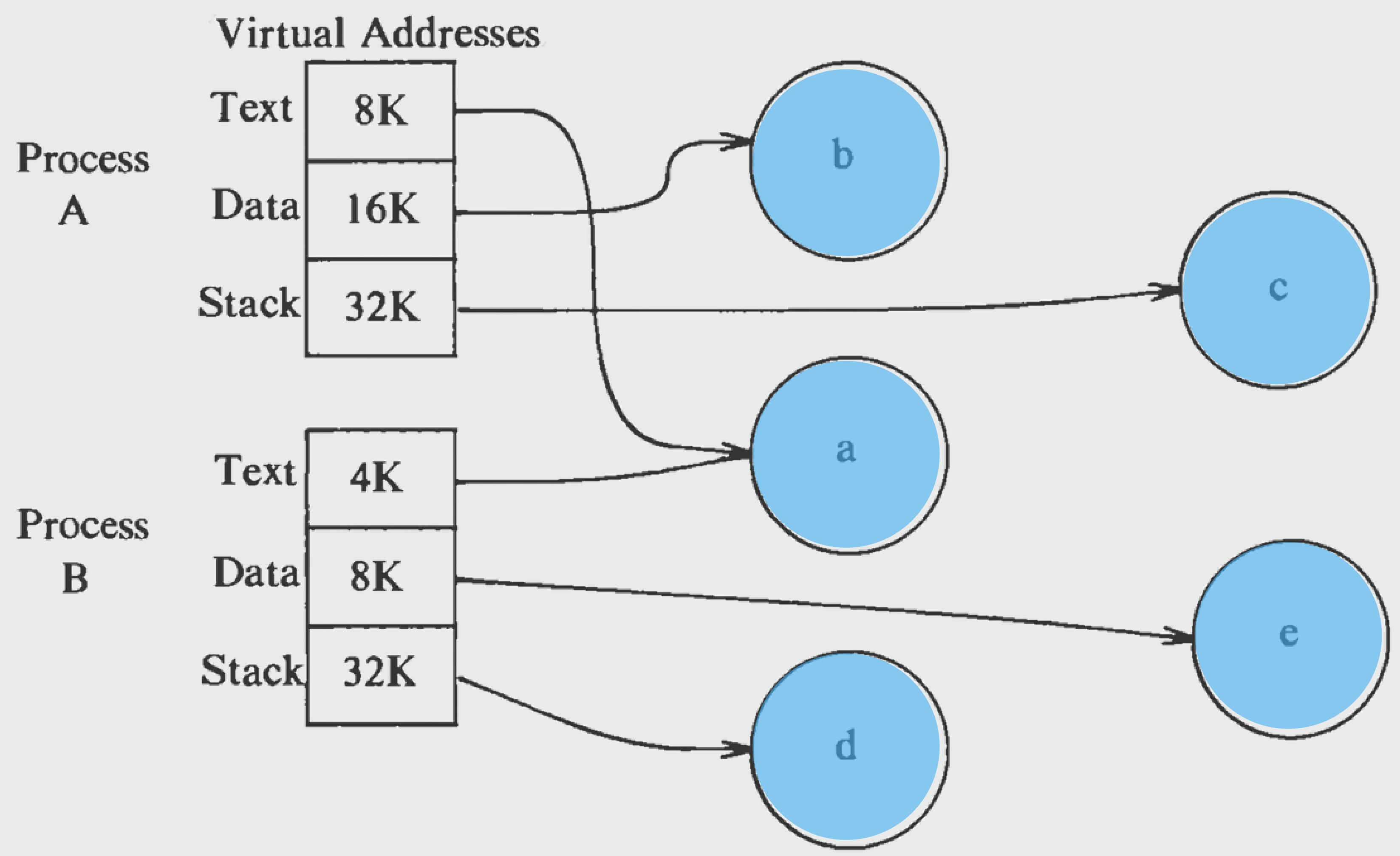
ADDRESS SPACE

Address Space is a space in computer memory and process address pace means a space that is allocated in memory for a process.

- Every process has an address space.
- Address space can be physical and virtual adress space.

The UNIX System divides the virtual address space into logical regions.

Region is an area of the address space of a process that can be treated as a distinct object to be shared or protected.



The Kernel has a Region Table Entry which contains the information necessary to describe a region. A region table entry contains the following information:

- The inode of the file from which the region was initially loaded.
- The type of the region (text, shared memory, private data, or stack).
- The size of the region.
- The location of the region in physical memory.
- The state of the region:
    - locked
    - in demand
    - being loaded into memory
    - valid, loaded into memory
- The reference count, giving the number of processes that reference the region

Now as we know that region is address space of a process, so there are different operations that can be performed on the region. The operations that manipulate regions are:

1. Lock a region
2. Unlock a region
3. Allocate a region
4. Attach a region to the memory space of a process
5. Change the size of a region
6. Load a region from a file into the memory space of a process
7. Free a region
8. Detach a region from the memory space of a process
9. Duplicate the contents of a region

Now let us see these operations in detail one by one ☺

LOCKING AND UNLOCKING A REGION :

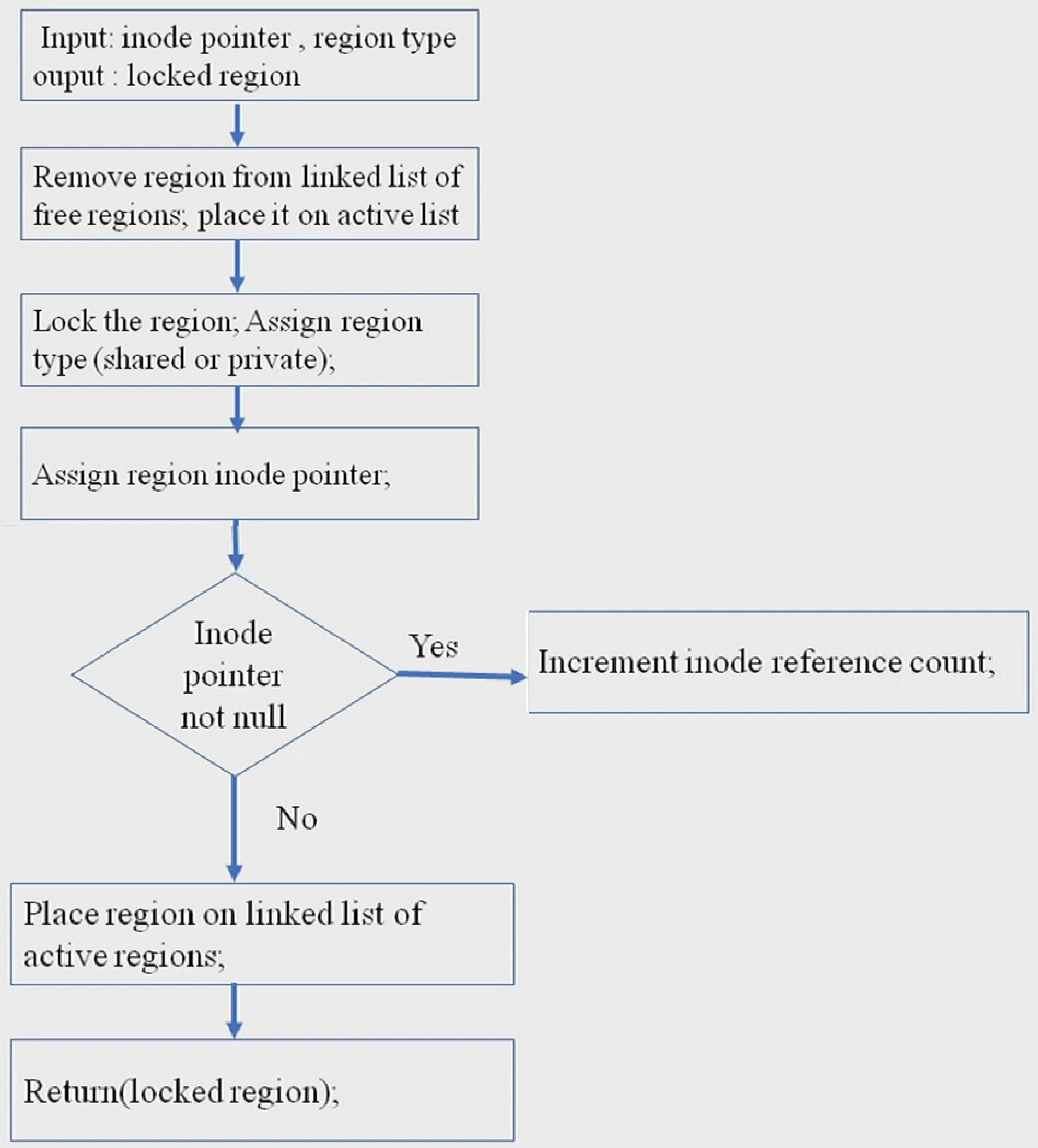
The kernel can lock and unlock a region independent of the operations to allocate and free a region, just like the the locking-unlocking mechanism of inodes is independent of the allocating the releasing (iget and iput) inodes.

ALLOCATING A REGION :

The kernel allocates a region in fork, exec, and shmget (shared memory get) system calls. Just like inodes, the kernel contains a free list of regions, when a region is to be allocated, the kernel picks up the first region from the free list and places it on the active list. The inode is used by the kernel to enable other process to share the region. The kernel increments the inode reference count to prevent other processes from removing its contents when unlinking it.

Algorithm ***allocreg*** to allocate region :

```
/*  Algorithm: allocreg
 *  Input:  indoe pointer
 *              region type
 *  Output: locked region
 */

{
    remove region from linked list of free regions;
    assign region type;
    assign region inode pointer;
    if  (inode pointer not null)
            increment inode reference count;
    place region on linked list of active regions;
    return (locked region);
}
```



Input: inode pointer , region type
ouput : locked region
Remove region from linked list of free regions; place it on active list
Lock the region; Assign region type (shared or private);
Assign region inode pointer;
Inode pointer not null
Yes
Increment inode reference count;
No
Place region on linked list of active regions;
Return(locked region);

ATTACHING A REGION TO A PROCESS

The kernel attaches the region to a processes address space with the attachreg system call. It is used in fork, exec, and shmat. The region being attached might be newly allocated or might be an already allocated region which is to be shared. The algorithm is given on the right :

```
/*  Algorithm: attachreg
 *  Input: pointer to (locked) region being attached
 *        process to which the region is being attached
 *        virtual address in process where region will be attac
 *        region type
 *  Output: pre process region table entry
 */

{
    allocate per process region table entry for process;
    initialize per process region table entry;
            set pointer to region being attached;
            set type field;
            set virtual address field;
   check legality of virtual address, region size;
   increment region reference count;
   increment process size according to attached region;
   initialize new hardware register triple for process;
   return (per process region table entry);
}
```

CHANGING THE SIZE OF A REGION

The stack region of a process grows and shrinks automatically according to the nesting of calls. It uses the growreg algorithm. The data region of a process can be extended with the sbrk system call. It also internally calls growreg. Both of these regions are private to a process. Shared regions cannot be extended, so there are no side-effects of growreg. Text regions also cannot grow.

The algorithm growreg is given on the right:

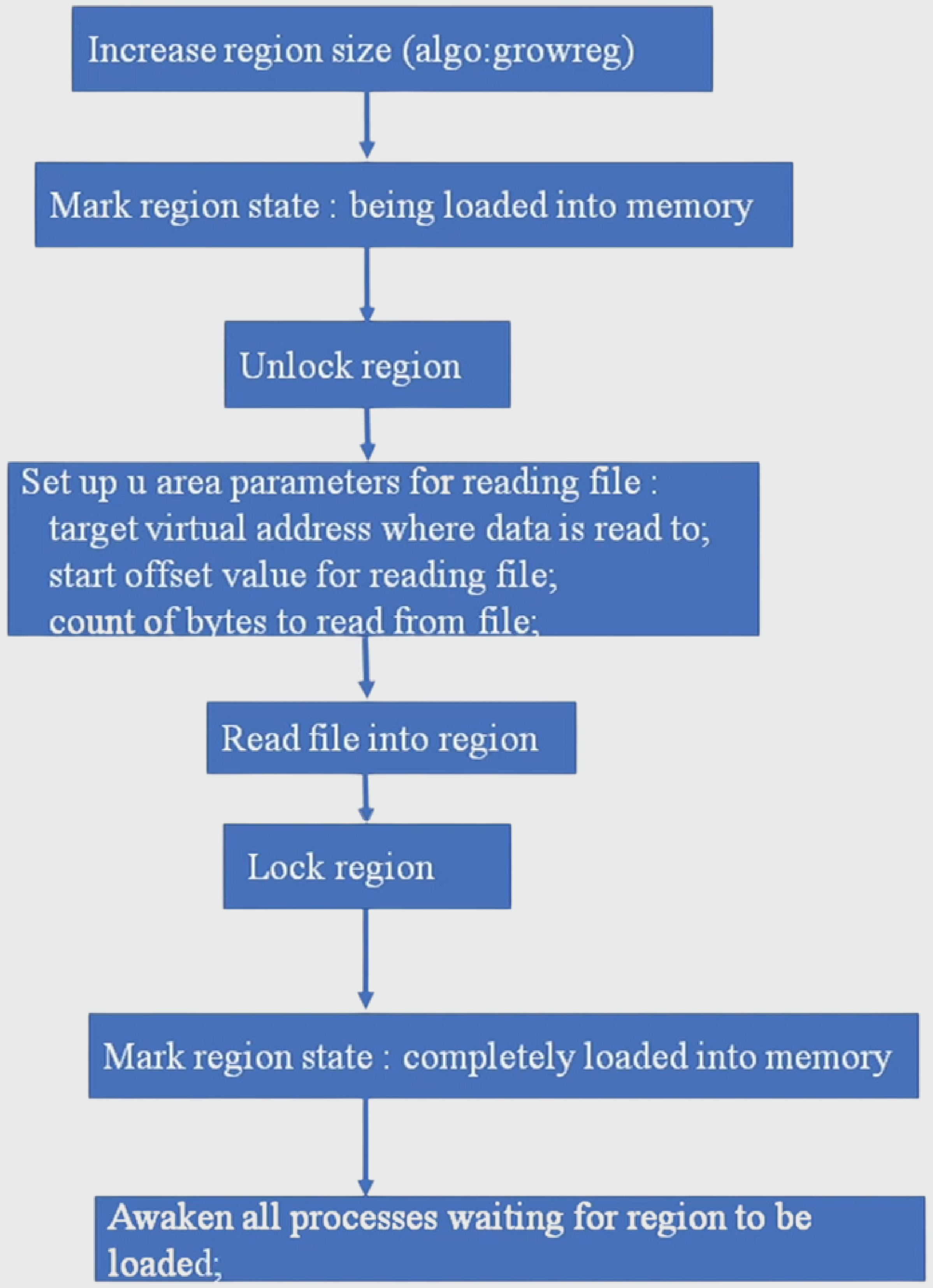
```
/*  Algo :     growreg
 *  Input:     pointer to per process region table entry
 *             change in size of region (positive or negative)
 *  Output:    none
 */

{
    if (region size increasing)
    {
        check legality of new region size;
        allocate auxiliary tables (page tables);
        if (not system supporting demand paging)
         {
            allocate physical memory;
            initialize auxiliary tables, as necessary;
          }
      }
    else  // region size decreasing
      {
          free physical memory, as appropriate;
          free auxiliary tables, as appropriate;
      }
    do (other) initialization of auxiliary tables, as necessary;
    set size field in process table;
}
```

LOADING A REGION

In a system supporting demand paging, the kernel can "map" a file into process address space during the exec system call, arranging to read individual physical pages later on demand (studied later). If the kernel does not support demand paging, it must copy the executable file into memory, loading the process regions at virtual addresses specified in the executable file. It may attach a region at a different virtual address from where it loads the contents of the file, creating a gap in the page table. For example, this feature is used to cause memory faults when user programs access address 0 illegally. Programs with pointer variables sometimes use them erroneously without checking that their value is 0 and, hence, that they are illegal for use as a pointer reference. By protecting the page containing address 0 appropriately, processes that errantly access address 0 incur a fault and abort, allowing programmers to discover such bugs more quickly.

ALGORITHM LOADING A REGION

```
/*  Algo :  loadreg
 *  Input:  pointer to per process region table entry
 *            virtual address to load region
 *            inode pointer of file for loading region
 *            byte offset in file for start of region
 *            byte count for amount of data to load
 *  Oput:  none
 */

{
    increase region size according to eventual size of region (algo: growreg);
    mark region state: being loaded into memory;
    unlock region;
    set up u-area parameters for reading file:
            target virtual address where data is read to,
            start offset value for reading file,
            count of bytes to read from file;
    read file into region (internal variant of read algorithm);
    lock region;
    mark region state: completely loaded into memory;
    awaken all processes waiting for region to be loaded;
}
```


Increase region size (algo:growreg)
Mark region state : being loaded into memory
Unlock region
Set up u area parameters for reading file :
target virtual address where data is read to;
start offset value for reading file;
count of bytes to read from file;
Read file into region
Lock region
Mark region state : completely loaded into memory
Awaken all processes waiting for region to be loaded;

FREEING A REGION

When a kernel no longer needs a region, it frees the region and places it on the free list again. The algorithm freereg is given below:

```
/*  Algo :     freereg
 *  Input:     pointer to a (locked) region
 *  Output:  none
 */

{
    if (region reference count non zero)
    {
            // some process still using region
            release region lock;
            if (region has an associated inode)
                 release inode lock;
            return;
     }
     if (region has associated inode)
           release inode (algorithm: iput);
     free physical memory still associated with region;
     free auxiliary tables associated with region;
     clear region fields;
     place region on region free list;
     unlock region;
}
```

DETACHING A REGION FROM A PROCESS

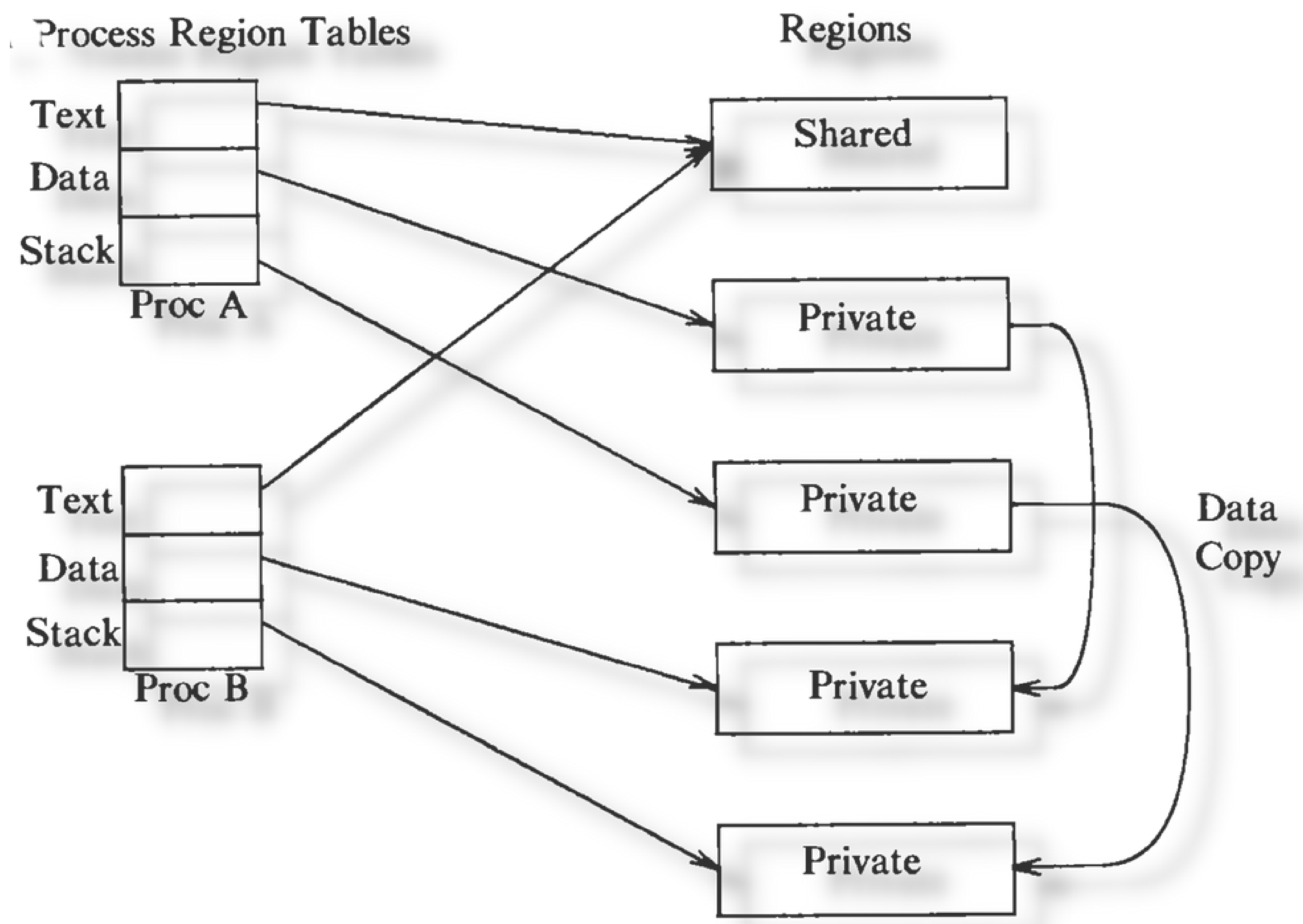
The kernel detaches regions in the exec, exit, and shmdt system calls. It updates the pregion entry and cuts the connection to physical memory by invalidating the associated memory management register triple. The address translation mechanisms thus invalidated apply specifically to the process, not to the region (as in the algorithm freereg). The kernel decrements the region reference count. If the region referenced count drops to 0 and there is no reason to keep the region in memory (studied later), the kernel frees the region with algorithm freereg. Otherwise, it only releases the region and inode locks.
The algorithm detachreg is given on right:

```
/*  Algo :    detachreg
 *  Input:    pointer to per process region table entry
 *  Output: none
 */

{
    get auxiliary memory management tables for process;
    decrement process size;
    decrement region reference count;
    if (region reference count is 0 and region not stick bit)  // studied later
         free region (algorithm: freereg;)
    else  // either reference count non-0 or region sticky bit on
    {
        free inode lock, if applicable (inode associated with region);
        free region lock;
    }
}
```

DUPLICATING A REGION

In the fork system call, the kernel needs to duplicate the regions of a process. If the region is shared, the kernel just increments the reference count of the region. If it is not shared, the kernel has to physically copy it, so it allocates a new region table entry, page table, and physical memory for the region. The algorithm dupreg is given below:

```
/*  Algo :    dupreg
 *  Input:    pointer to region table entry
 *  Output: pointer to a region that looks identical to input region
 */

{
    if (region type shared)
            // caller will increment region reference count with
subsequent attachreg call
          return (input region pointer);
    allocate new region (algorithm: allocreg);
    set up auxiliary memory management structures, as currently exist
in input region;
    allocate physical memory region contents;
    "copy" region contents from input region to newly allocated region;
     return (pointer to allocated region);
}
```



THANK YOU