Shared Memory

- `shmget()` creates new region of shared memory or returns one
- `shmat()` logically attaches all regions to the virtual address space of a process
- `shmdt()` detaches a region from the virtual address space of a process
- `shmctl()` manipulates various parameters associated with shared memory

Processes read and write shared memory using the same instructions they use to read and write regular memory.

After attaching, no system calls are needed access data in shared memory.

shmget

```
shmid = shmget(key, size, flag);
```

`size` is the number of bytes in the region.

The kernel searches the shared memory table for the given key

- If it finds an entry and the permission modes are acceptable, it returns the descriptor for the entry
- If it does not find an entry, and the user has set `IPC_CREAT` flag, the kernel verifies that the size is between the system wide minimum and maximum values and then allocates a new region data structure using algorithm `allocreg`
- The kernel saves the permission modes, size, and a pointer to the region table table entry in the shared memory table and sets a flag there to indicate that no memory is associated with the region.
- It only allocates memory when a process attaches a region to its address space
- The kernel also sets a flag on the region table entry to indicate that the region should not be freed when the last processes that attaches to it exits.

shmat

A process attaches a shared memory region to its virtual address space

```
virtaddr = shmat(id, addr, flags);
```

`addr` is the virtual address where the user wants to attach the shared memory region
`flags` indicate whether the region is readonly and whether the kernel should round off the user specified address.
`virtaddr` is where the kernel attached the region

- When executing `shmat`, the kernel verifies that the process has the necessary permissions to access the region
- It examines the user specified address, if its 0, the kernel chooses a convenient virtual address.
- The shared memory region must not overlap other regions in the virtual address space of the process.
- If the kernel attaches the region next to data region, and the process execute `brk()` , then the new data region is virtually contiguous to to the previous one.
- Similarly, it should not attach close to the stack region
- **Best place would be immediately before the start of the stack region**.
- The kernel checks that the shared memory region fits into the process address space using algo `attachreg`
- If the calling process if the first to attach the region, the kernel allocates necessary tables, using algo `growreg`.

shmdt

A process detaches a shared memory region from its virtual address space by

```
shmdt(addr);
```

`addr` is the address returned by `shmat()`

- Detaches the shared memory region using `detachreg` algo.
- Since the region table entry has no back pointers to the shared memory table, the kernel searches the shared memory table for the entry that points to the region and adjusts it for time the region was last detached.

shmctl

To get query status and set parameters for the shared memory region

```
shmctl(id, cmd, shmstatbuf);
```

`id` identifies the shared memory table entries

When removing a shared memory region, the kernel frees the entry and looks at the region table entry:

- If no process has the region attached to its virtual address space, it frees the region table entry and all its resources, using algorithm `freereg`.
- If the region is still attached to some processes (its reference count > 0), the kernel just clears the flag that indicates whether the region should not be freed when the last process detached the region.
- Processes that are using the shared memory may continue doing so, but no new processes can attach it.

Semaphores

Dekker Algorithm

- Integer valued objects
- Two atomic operations: P and V
- P decrements the value of semaphore if > 0
- V increments the value
- The operations are atomic
  - Only one P or V operation can succeed.

Contents of Semaphores in UNIX System V

- Value of semaphore
- process ID of the last process to manipulate it
- The number of processes waiting for the semaphore value to increase
- The number of processes waiting for the semaphore value to equal 0.

System calls

- `semget` to create and gain access to a set of semaphores
- `semctl` to do various control operations on the set
- `semop` to manipulate the value of semaphore

semget

```
semid = semget(key_t key, int nsems, int flag);
```

- if `key == IPC_PRIVATE`, new semaphore is created
- If no semaphore corresponding to this key exists, `IPC_CREAT` is asserted
- otherwise, integer id for existing semaphore is returned
- `nsems` specifies the number of semaphores to be created, i.e. the kernel allocates an entry that points an array
- 0 if existing semaphore is used

semop

```
int semop(int semid, struct sembuf* ops, size_t nops);

struct sembuf{
        ushort sem_num; // semaphore number in the array
        short sem_op;
        short sem_flg; // specifies IPC_NOWAIT, etc
}
```

`sem_op > 0` add this value to semval (unlocking / returning resources)
`sem_op < 0` subtract this value from semval (obtaining resource)

- IPC_NOWAIT $\Rightarrow$ return -1 (error)
- otherwise block until `semval >= abs(sem_op)` (has resources to satisfy this request)

`sem_op == 0` wait until `semval == 0`

semctl

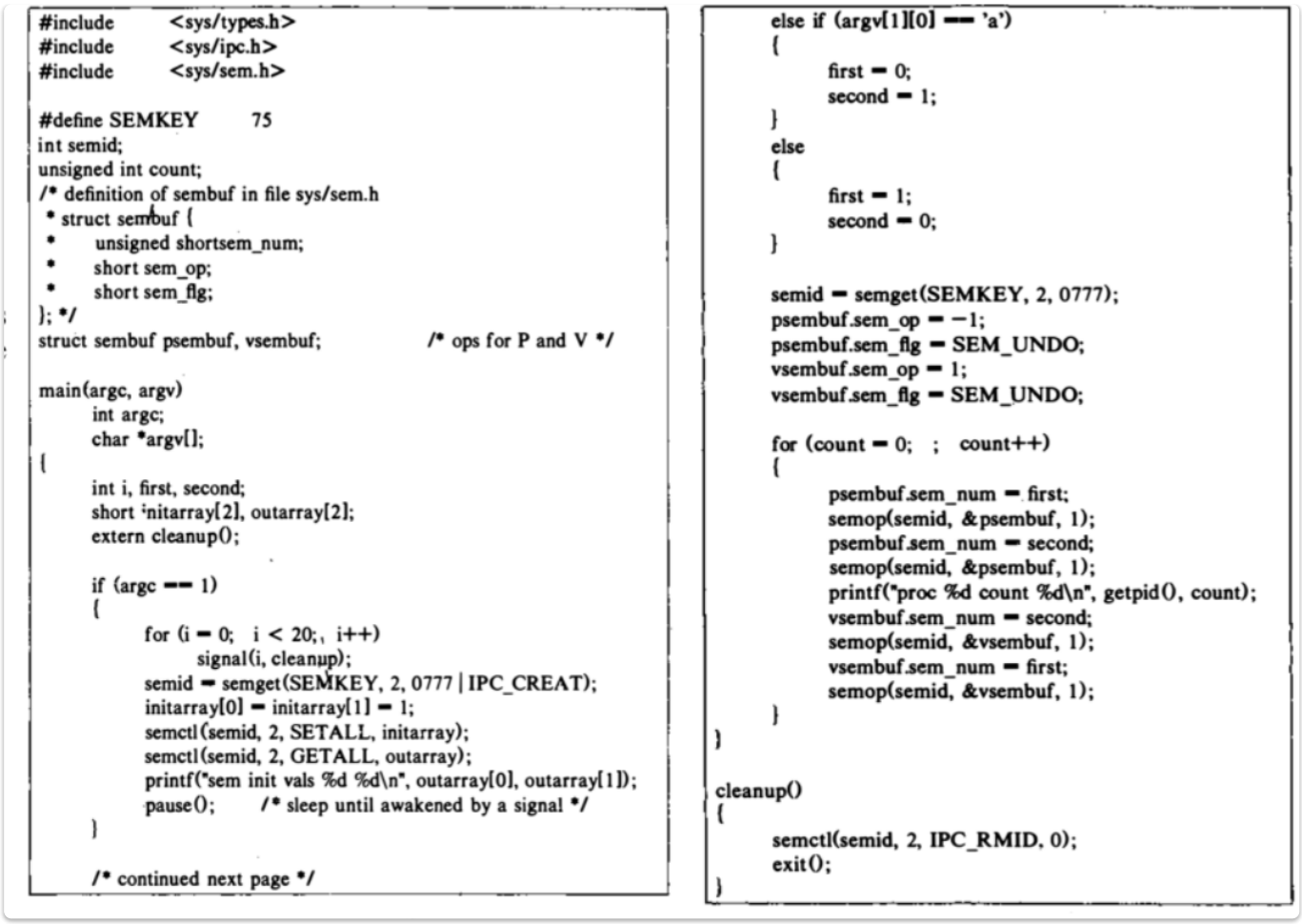
```
int semctl(int semid, int semnum, int cmd, union semun arg);

union semun{
int val;
ushort *array;
struct semid ds *buf;
}
```

`semnum` spciefies which semaphore is being operated on

`cmd == GETVAL` gets the value of `base[sumnum].semval`
`cmd == SETVAL` sets the value of `base[sumnum].semval` to `arg.val`

ALGORITHM SEMOP EXPLANATION

- The kernel reads the array of semaphore operations from the user address space and verifies that **semaphore numbers are legal** and that the **process has necessary permissions** to read or change the semaphores.
- If permission is not allowed, sys call fails
- The kernel changes the value of semaphore according to the operation.
- If positive, it increments the value of the semaphore and awakens all processes that are waiting for the value of the semaphore to increase
- If the semaphore operation is 0, the kernel checks the semaphore value: if 0, continues with other operations in the array; otherwise, it increments the number of processes asleep, waiting for the semaphore value to become 0, and goes to sleep.
- If the semaphore value is negative, and its value is less than or equal to the value of the semaphore, the kernel adds the operation value (negative) to the semaphore value.
- If the result is 0, awakens all processes asleep waiting for the semaphore value to become 0.
- If the value of the semaphore is less than the absolute value of semaphore value, it puts the process to sleep on the event that value of the semaphore increases.

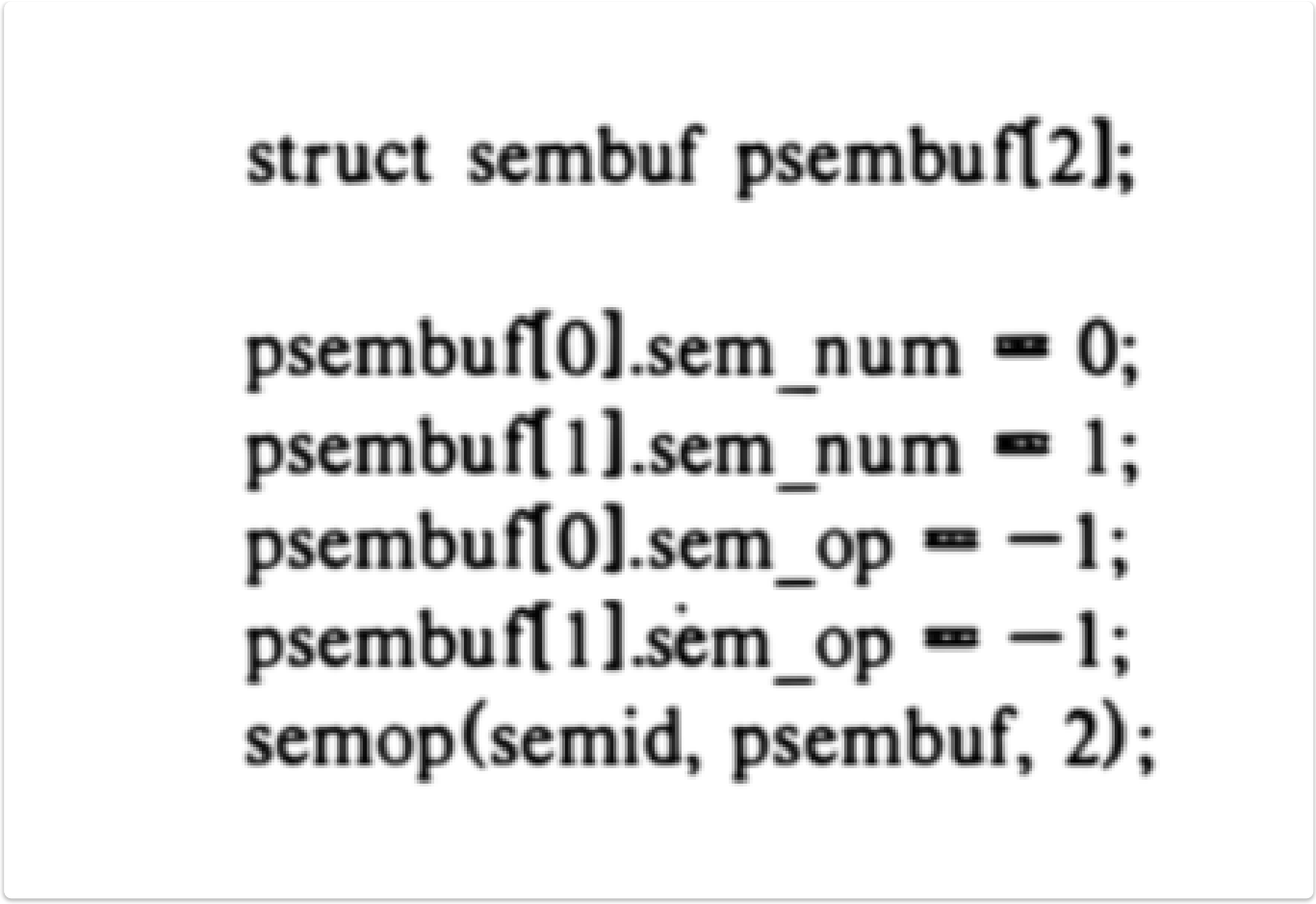


```
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>

#define SEMKEY        75
int semid;
unsigned int count;
/* definition of sembuf in file sys/sem.h
 * struct sembuf {
 *      unsigned short sem_num;
 *      short sem_op;
 *      short sem_flg;
 }; */
struct sembuf psembuf, vsembuf;                 /* ops for P and V */

main(argc, argv)
        int argc;
        char *argv[];
{
        int i, first, second;
        short initarray[2], outarray[2];
        extern cleanup();

        if (argc == 1)
        {
                for (i = 0;  i < 20;, i++)
                        signal(i, cleanup);
                semid = semget(SEMKEY, 2, 0777 | IPC_CREAT);
                initarray[0] = initarray[1] = 1;
                semctl(semid, 2, SETALL, initarray);
                semctl(semid, 2, GETALL, outarray);
                printf("sem init vals %d %d\n", outarray[0], outarray[1]);
                pause();        /* sleep until awakened by a signal */
        }

        /* continued next page */
```

```
        else if (argv[1][0] == 'a')
        {
                first = 0;
                second = 1;
        }
        else
        {
                first = 1;
                second = 0;
        }

        semid = semget(SEMKEY, 2, 0777);
        psembuf.sem_op = -1;
        psembuf.sem_flg = SEM_UNDO;
        vsembuf.sem_op = 1;
        vsembuf.sem_flg = SEM_UNDO;

        for (count = 0;  ;  count++)
        {
                psembuf.sem_num = first;
                semop(semid, &psembuf, 1);
                psembuf.sem_num = second;
                semop(semid, &psembuf, 1);
                printf("proc %d count %d\n", getpid(), count);
                vsembuf.sem_num = second;
                semop(semid, &vsembuf, 1);
                vsembuf.sem_num = first;
                semop(semid, &vsembuf, 1);
        }
}

cleanup()
{
        semctl(semid, 2, IPC_RMID, 0);
        exit();
}
```

In the above, process A has locked semaphore 0 and process B has locked semaphore 1, so when process A wants to lock semaphore 1 but it is already locked, so it goes to sleep, and process B wants to lock semaphore 0, and process B goes to sleep. SO effectively, they are deadlocked.

SOLUTION: use operations array



In the above, the kernel will only decrement the values of the semaphores until they both can be decremented, otherwise they are left intact. Meaning, if a process can lock both semaphore, then only it will lock them

SCENARIO

If a process locks a semaphore, but when it exits, releasing the resources, does not reset the semaphore value. Other process would it locked, even though the process that locked it exited.

SOLUTION: A process can set `SEM_UNDO` flag, it reverses the effect of every semaphore operation

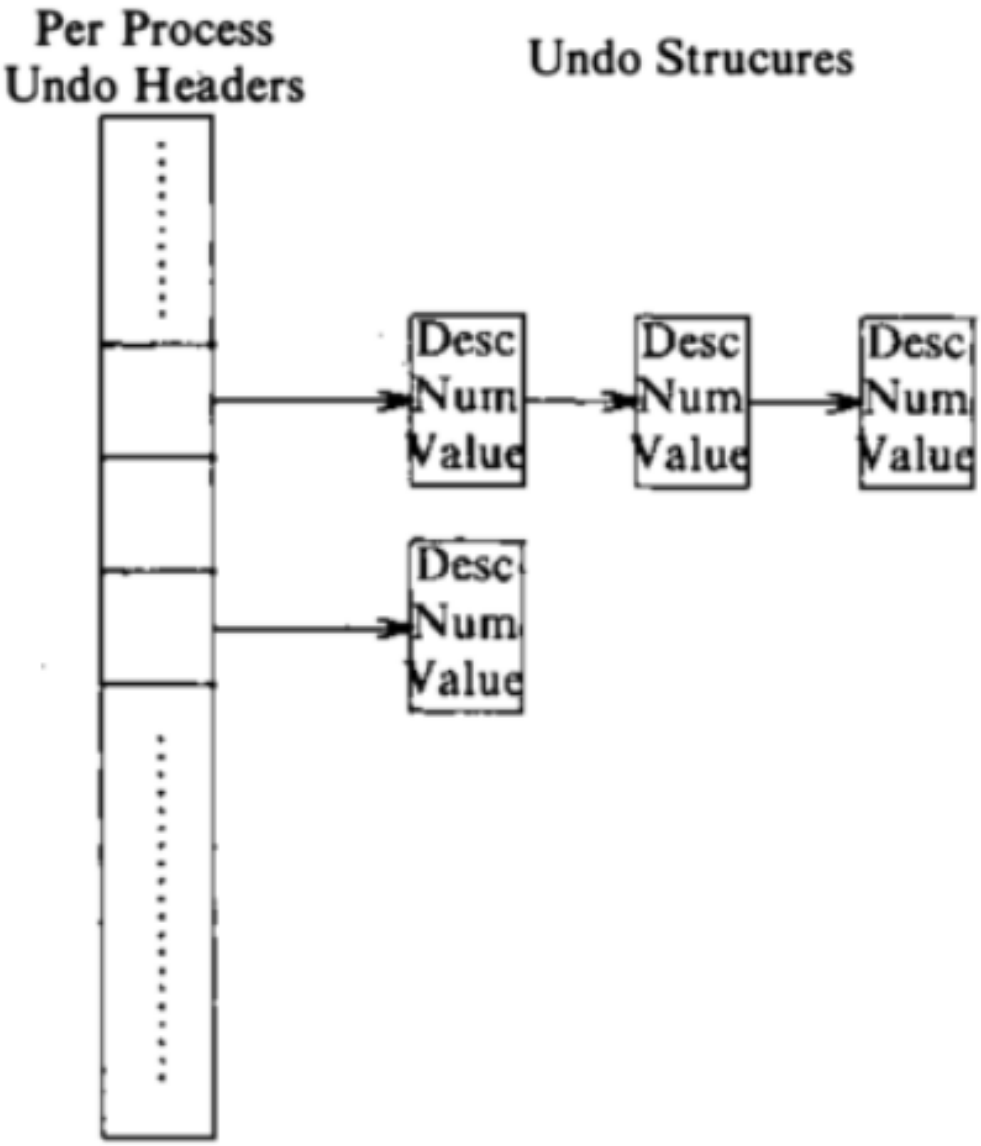


Figure 11.16. Undo Structures for Semaphores

Undo structure contains:

- semaphore ID
- semaphore number in the set identified by the ID
- An adjustment value = adjustment value - sem_op

The undo structure contains the negated sum of all semaphore operations the process had done on the semaphore. The kernel calls a special routine, when a program exits, that goes

through the undo structure.

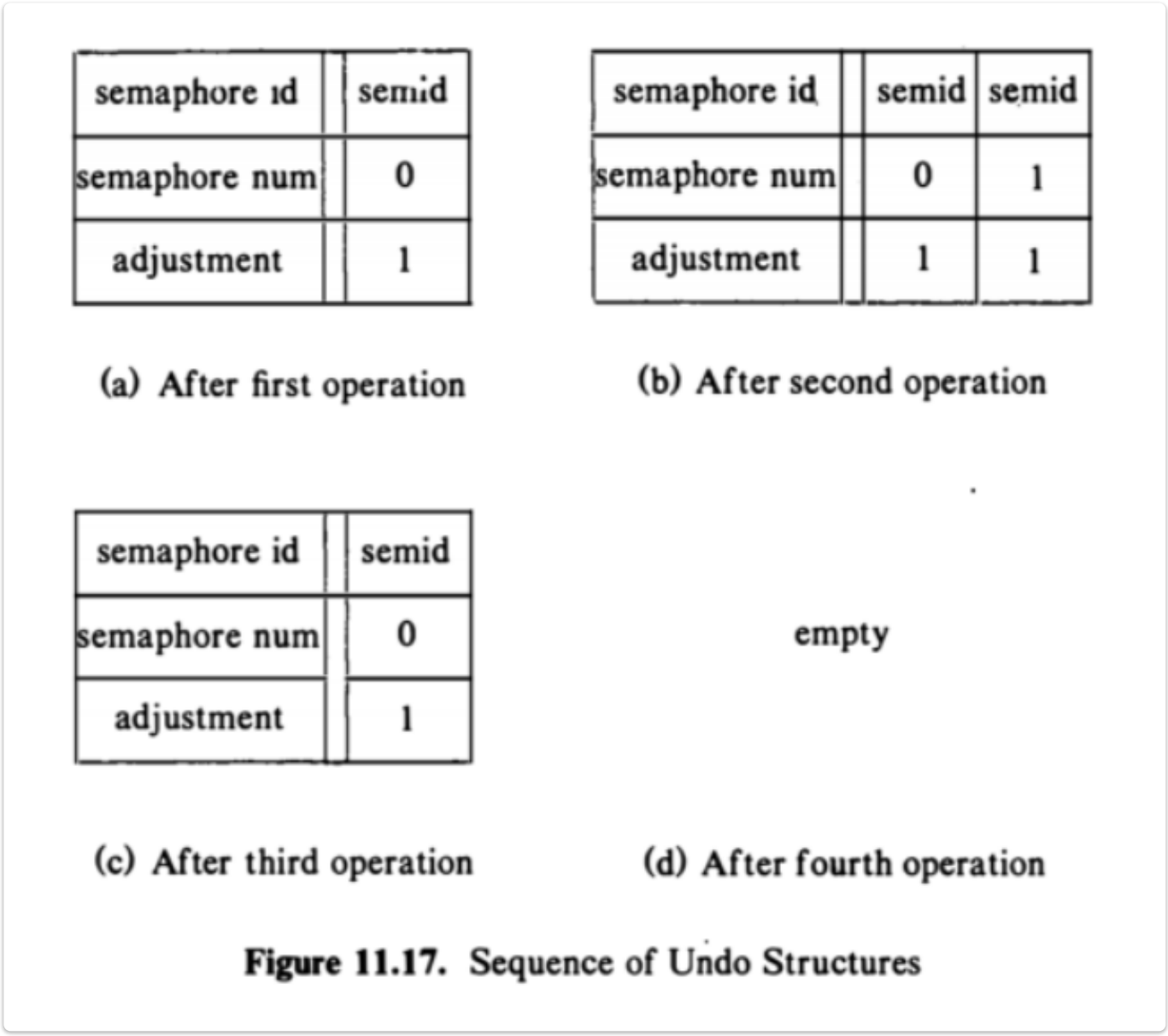


Figure 11.17. Sequence of Undo Structures

If the process were to exit after the second operation, the kernel will go through the structure and add value 1 to both semaphores restoring their values to 0.