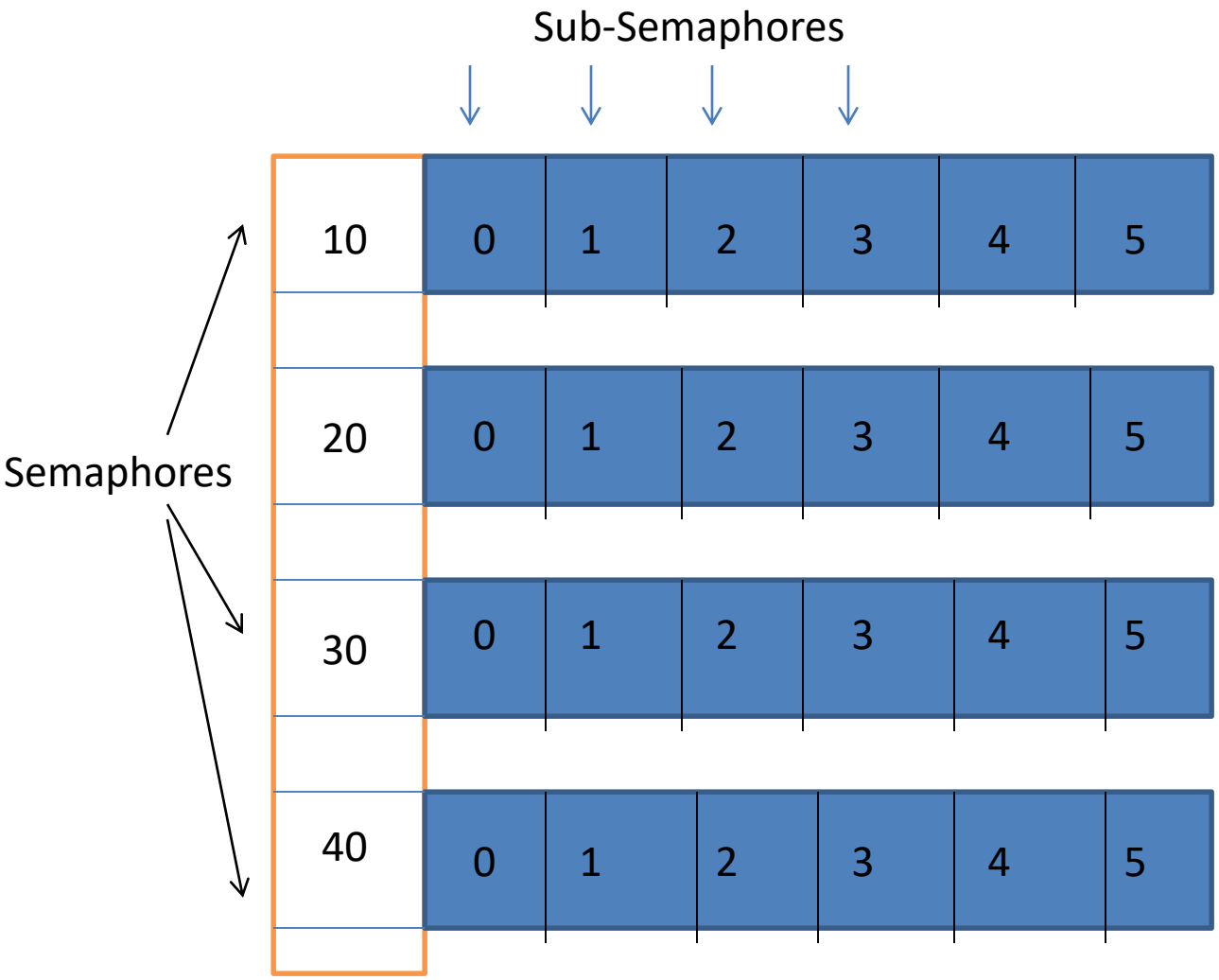
Semaphore

Semaphore structure



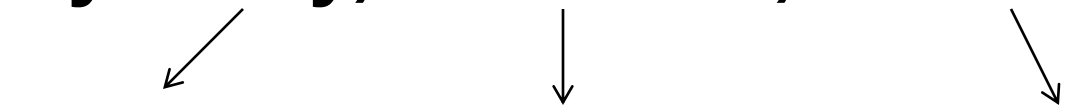
Creating and Accessing Semaphore Sets

int semget (key_t key,int nsems,int semflg)

Header:

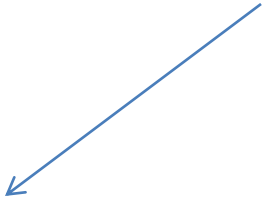
sys/types.h

sys/ipc.h




```
Main()
{
        key=(key_t)20
        nsem=1
        semid=semget(key, nsem, IPC_CREAT|0666)

}
```



ipcs –s

ID Key mode Owner nsems

Flag: IPC_EXCL: Exclusive creation of semaphore

IPC_CREAT|0666|IPC_EXCL

Setting and getting semaphore value

Setting a value:

Semctl(semid, subsem_id, SETVAL, value)

Getting value

int Semctl(semid, subsem_id, GETVAL, 0)

```
Main()
{
int semid;
Key=20;
Semid=semget(key,1,0666|IPC_CREAT);
Semctl(semid, 0, SETVAL, 1);
retval=semctl(semid, 0, GETVAL, 0);
Printf(“%d”, retval);

}
```

More on semctl()

- Getting the pid of the process who has last set the value of the semaphore

int Semctl(semid, sub-semid, GETPID, 0)

Process ID

```
Main()
{
int semid;
Key=20;
Semid=semget(key,1,0666|IPC_CREAT);

retval=semctl(semid, 0, GETPID, 0);

printf("PID retuned by semctl is %d and currnet pid is %d", retval,
getpid());

semctl(semid, 0, SETVAL, 1);


}
```

More on semctl()

SETALL and GETALL

```
Main()
{
        key=20;
        ushort val[5]={1, 6, 8, 11, 3}, retval[5];
        semid=semget(key, 5, 0666|IPC_CREAT);
        semctl(semid, 0, SETALL, val);
        semctl(semid, 0, GETALL, retval)
Printf("retval[0]=%d, retval[1]=%d, …….", retval[0], retval[1],,,)

}
```

More on semctl()

- Removing a semaphore

Semctl(semid, 0, IPC_RMID, 0);

Command

ipcrm –s <semid>

Atomicity: Implementing wait and signal

Concept

S

(sub)Semaphore variable

Sem_op

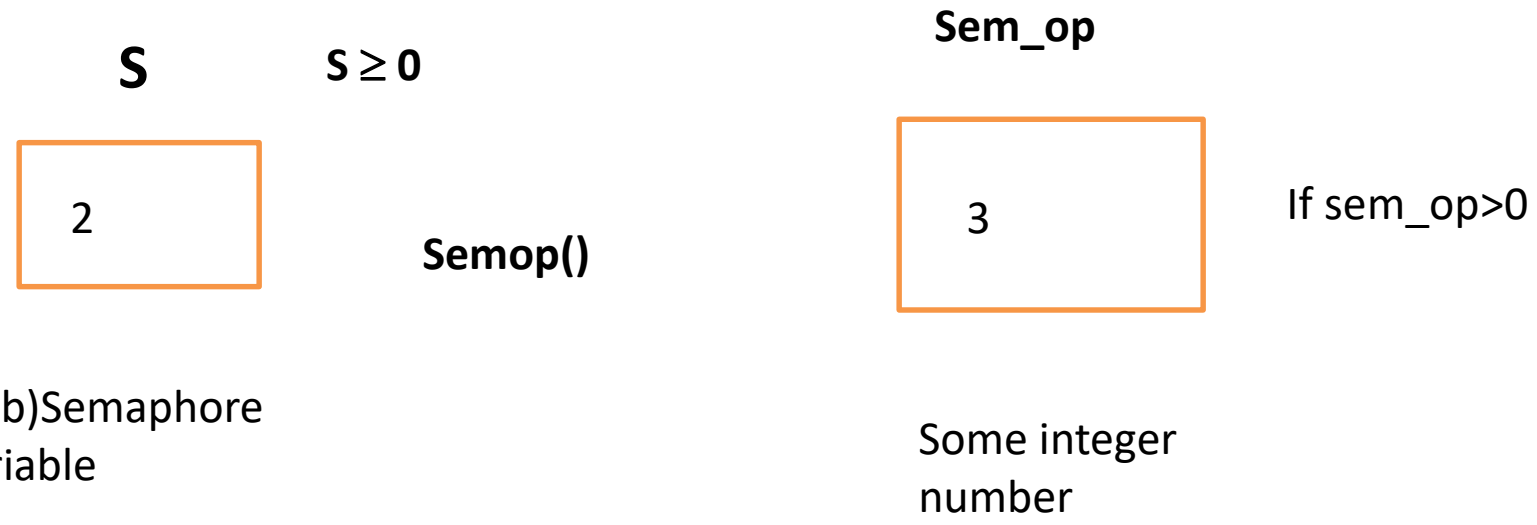
Some integer number

- **Semop()** system call
- Compares S with sem_op
- Takes an action
  - Either proceed
  - Or the process gets blocked (switch from running to waiting)

Atomic action

Atomicity: Implementing wait and signal

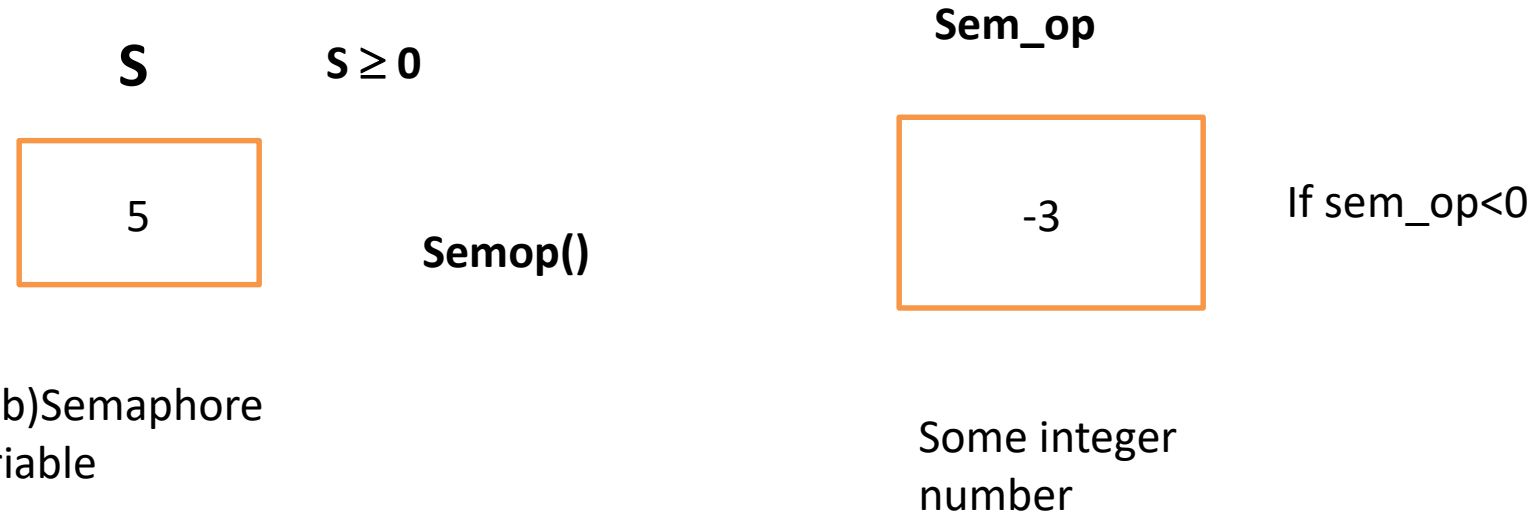
Concept



- S and sem_op both are positive
- Add (2+3) and update the value of semaphore
- Semop() returns and s becomes 5
- **Proceed** !

Atomicity: Implementing wait and signal

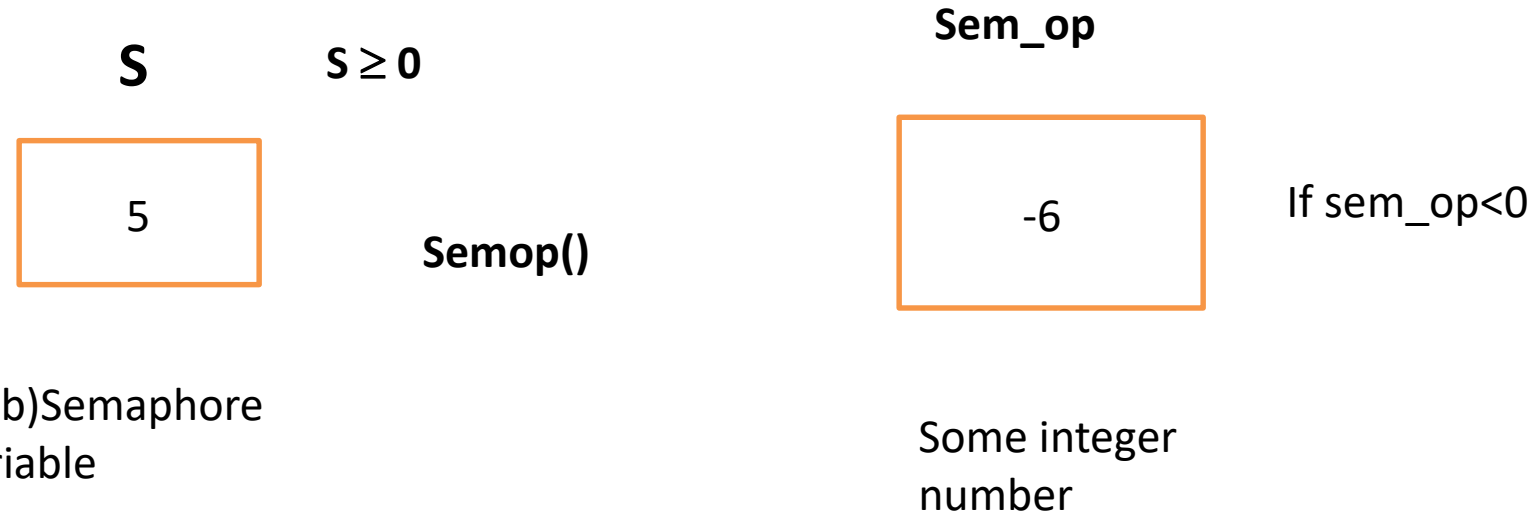
Concept



- sem_op is negetive
- Check if S ≥|sem_op|
- Update the value of S=S+sem_op
- Semop() returns and s becomes 2
- **Proceed!**

Atomicity: Implementing wait and signal

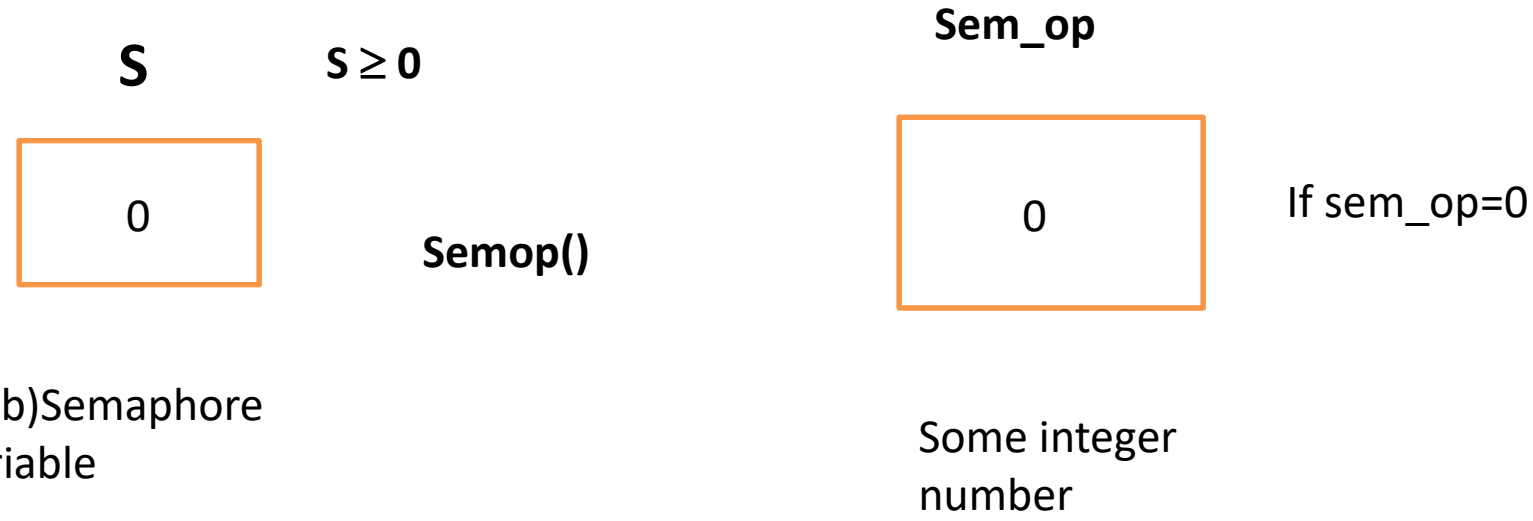
Concept



- sem_op is negetive
- Check if S <|sem_op|
- **Blocked!**
- Until S ≥|sem_op|

Atomicity: Implementing wait and signal

Concept



- sem_op is 0 (special case)
- Check if S ==0
- **If true, return (proceed)**
- **Else (S is positive)**
- **Block**

Atomicity: Implementing wait and signal

```
struct sembuf
{
        ushort sem_num;
        short  sem_op;
        short sem_flg;
}
```

Sub semaphore

0, IPC_NOWAIT, SEM_UNDO

int semop(int semid, struct sembuf *sops, unsigned nsops);

Atomicity: Implementing wait and signal

Set.c

```
Main()
{
Scanf("%d", &val);
Semid=semget(20, 1, IPC…);
Semctl(semid, 0, SETVAL, val)

}
```

Run.c

```
Main()
{
struct sembuf sop;
Semid=semget(20, 1, …);
Sop.sem_num=0;
Sop.sem_op=0;
Sop_sem_flg=0;
Semop(semid, &sop, 1);

}
```

Atomicity: Implementing wait and signal

```
Main()
{       struct sembuf sop;
        semid()=semget(20, 1, IPC_CREAT|0666);
        semctl(semid, 0, SETVAL, 1);
        pid=fork();
        if(pid==0)
        {

        }
```

Child process

```
Sop.sem_num=0;
Sop.sem_op=-1;
Sop.sem_flg=0;
Semop(semid, &sop, 1);
CRITICAL SECTION
Sop.sem_num=0;
Sop.sem_op=1;
Sop.sem_flg=0
Semop(semop,&sop,1);
```

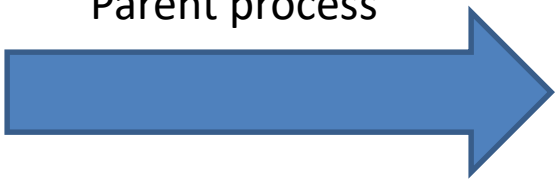
wait

signal

Atomicity: Implementing wait and signal

```
Main()
{       struct sembuf sop;
        semid()=semget(20, 1, IPC_CREAT|0666);
        semctl(semid, 0, SETVAL, 1);
        pid=fork();
        if(pid==0)
        {
                Child process
        }
        else
        {
                Parent process

        }
```



```
Sop.sem_num=0;
Sop.sem_op=-1;
Sop.sem_flg=0;
Semop(semid, &sop, 1);      wait
CRITICAL SECTION
Sop.sem_num=0;
Sop.sem_op=1;
Sop.sem_flg=0
Semop(semop,&sop,1);        signal
```

SEM_UNDO

```
struct sembuf
{
        ushort sem_num;
        short  sem_op;
        short sem_flg;
}
```

```
Sop.sem_num=0;
Sop.sem_op=-1;
Sop.sem_flg=0;
Semop(semid, &sop, 1);
CRITICAL SECTION
Sop.sem_num=0;
Sop.sem_op=1;
Sop.sem_flg=0
Semop(semop,&sop,1);
```

SEM_UNDO

Equivalent

Resets the semaphore value

SEM_UNDO

```
Main()
{
        semid=semget()
        semctl(semid, 0, SETVAL, 1);
        sop.sem_num=0;
        sop.sem_op=-1;
        sop.sem_flg=SEM_UNDO;
        pid=fork()
        if(pid==0)
        {
        }
        else
        {
        }
```

Child process

Semop(semid, &sop, 1);
CS

Parent process

Semop(semid, &sop, 1);
CS

Kernel data structures

Semaphore structure

Sem_ids

```
/* One sem_array data structure for each set of semaphores in the system. */

struct sem_array {
struct kern_ipc_perm    sem_perm;       /* permissions .. see ipc.h */
time_t                  sem_otime;      /* last semop time */
time_t                  sem_ctime;      /* last change time */
struct sem              *sem_base;      /* ptr to first semaphore in array */
struct sem_queue *sem_pending;  /* pending operations to be processed */
struct sem_queue        **sem_pending_last; /* last pending operation */
struct sem_undo         *undo;          /* undo requests on this array */
unsigned long           sem_nsems;      /* no. of semaphores in array */
};
```

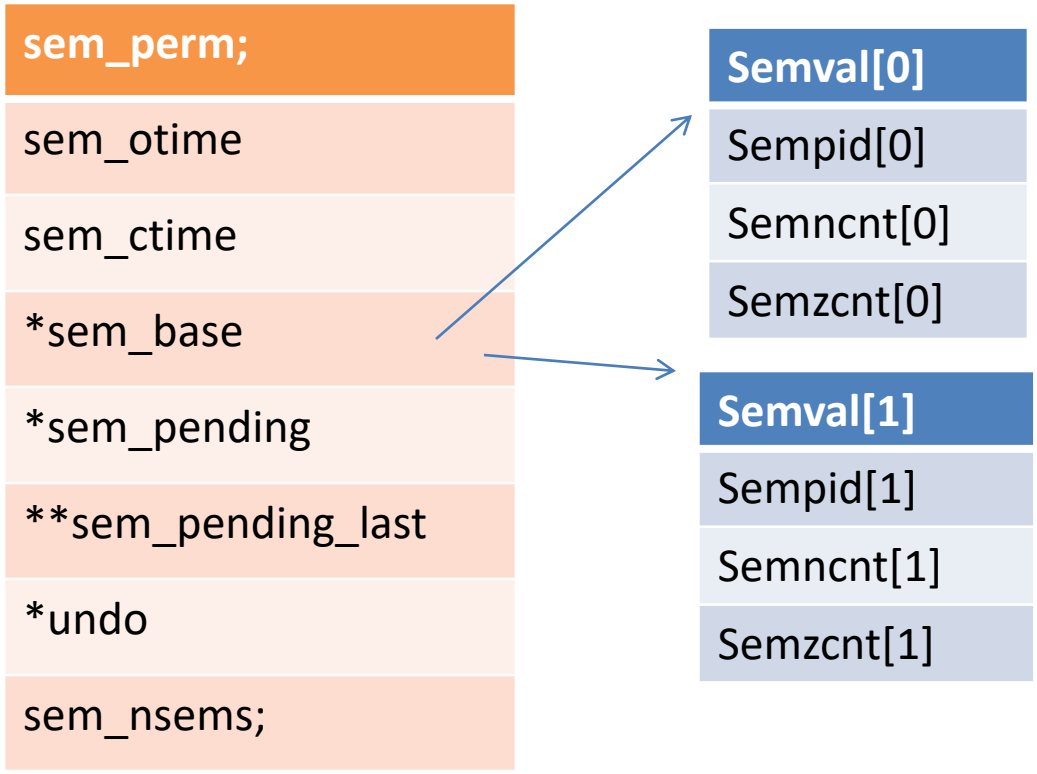
Sometime refer as **semid_ds**

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

```
struct sem {
        u_short         semval;
        short    sempid;
        u_short         semncnt;
        u_short         semzcnt;
};
```

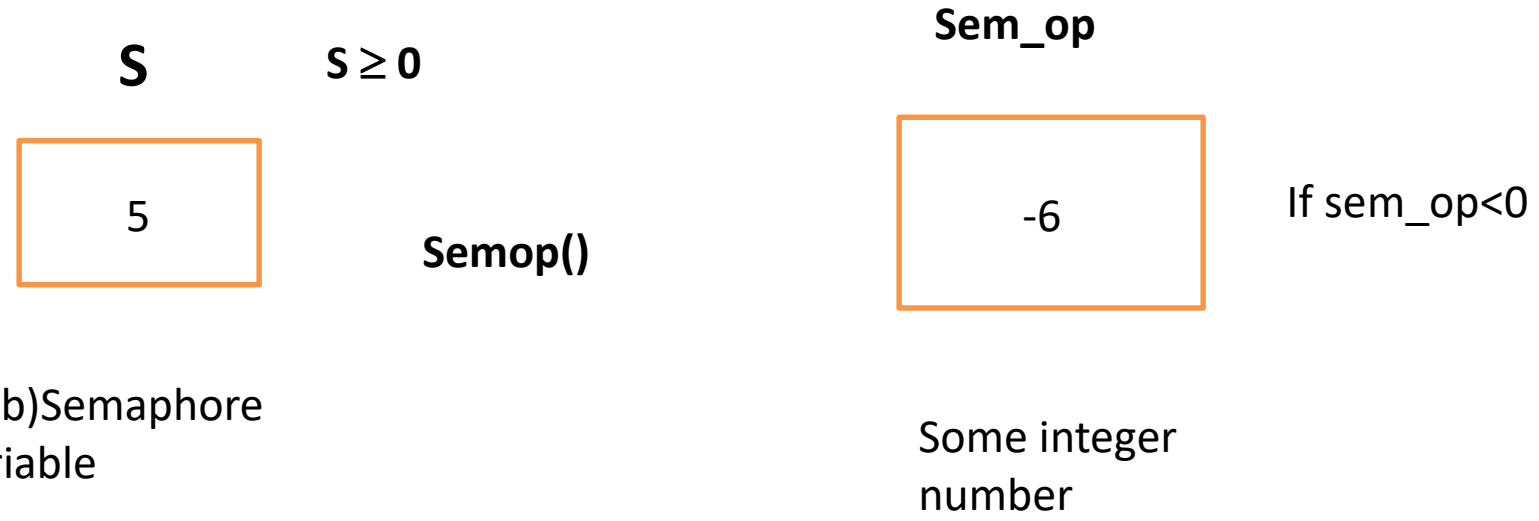
semncnt; → Waiting for positive value

semzcnt; → Waiting for zero


sem_perm;
sem_otime
sem_ctime
*sem_base
*sem_pending
**sem_pending_last
*undo
sem_nsems;
Semval[0]
Sempid[0]
Semncnt[0]
Semzcnt[0]
Semval[1]
Sempid[1]
Semncnt[1]
Semzcnt[1]

Atomicity: Implementing wait and signal

Concept

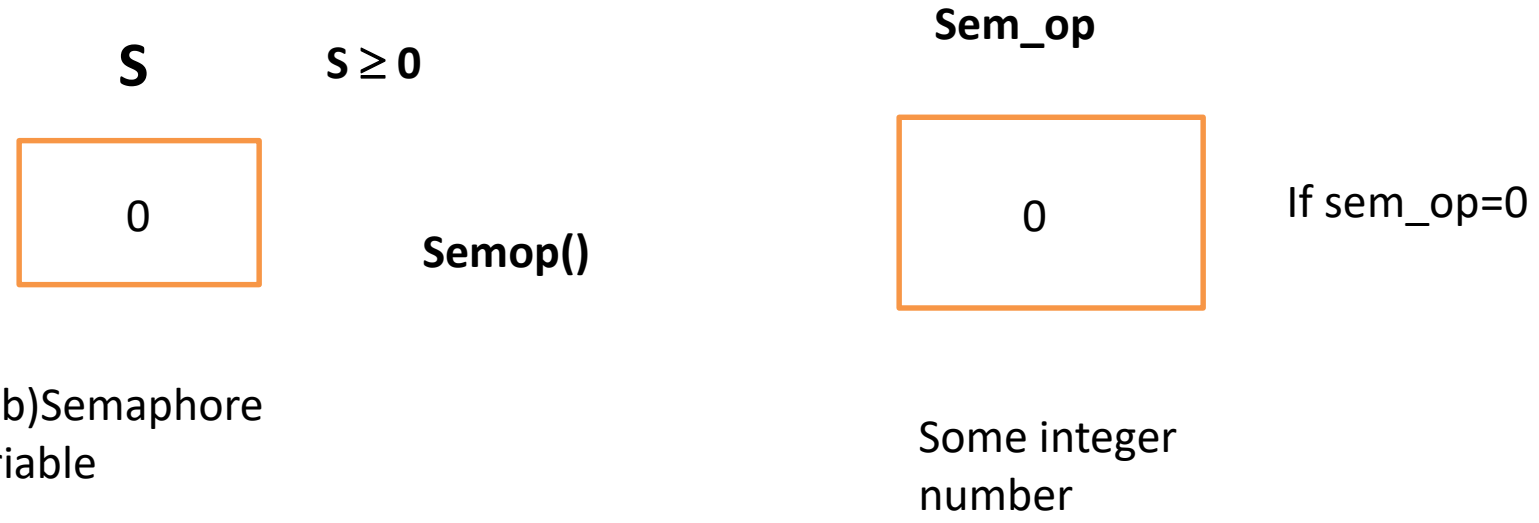


- sem_op is negetive
- Check if S <|sem_op|
- **Blocked!**
- Until S ≥|sem_op|

semncnt

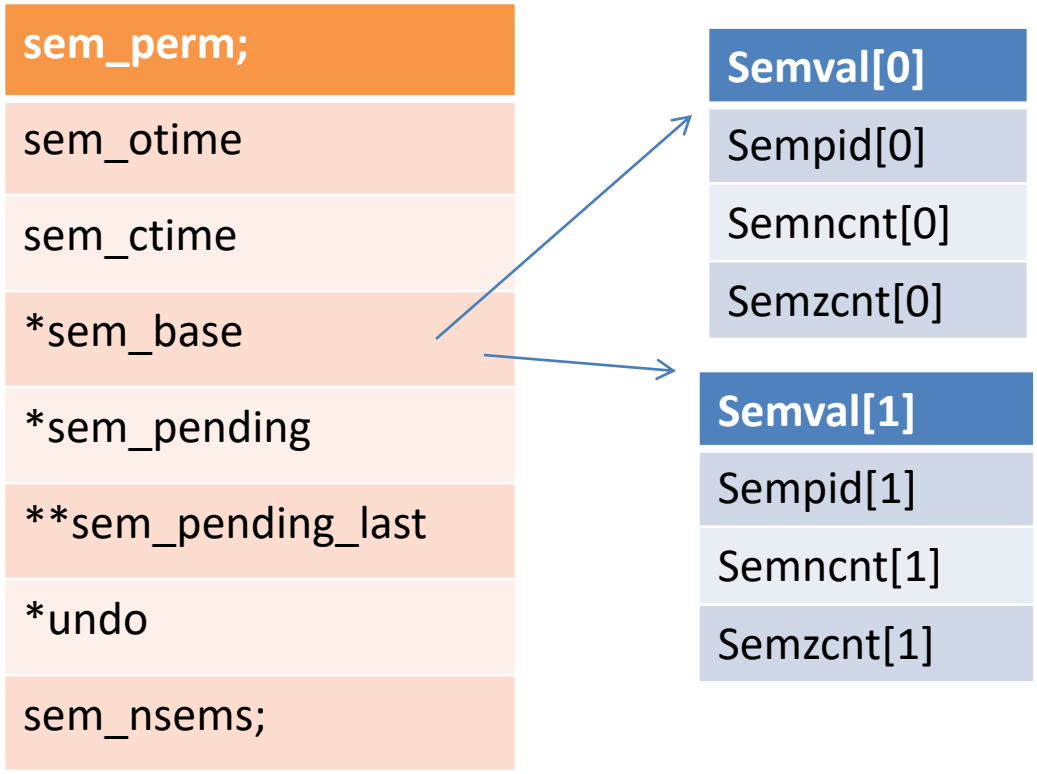
Atomicity: Implementing wait and signal

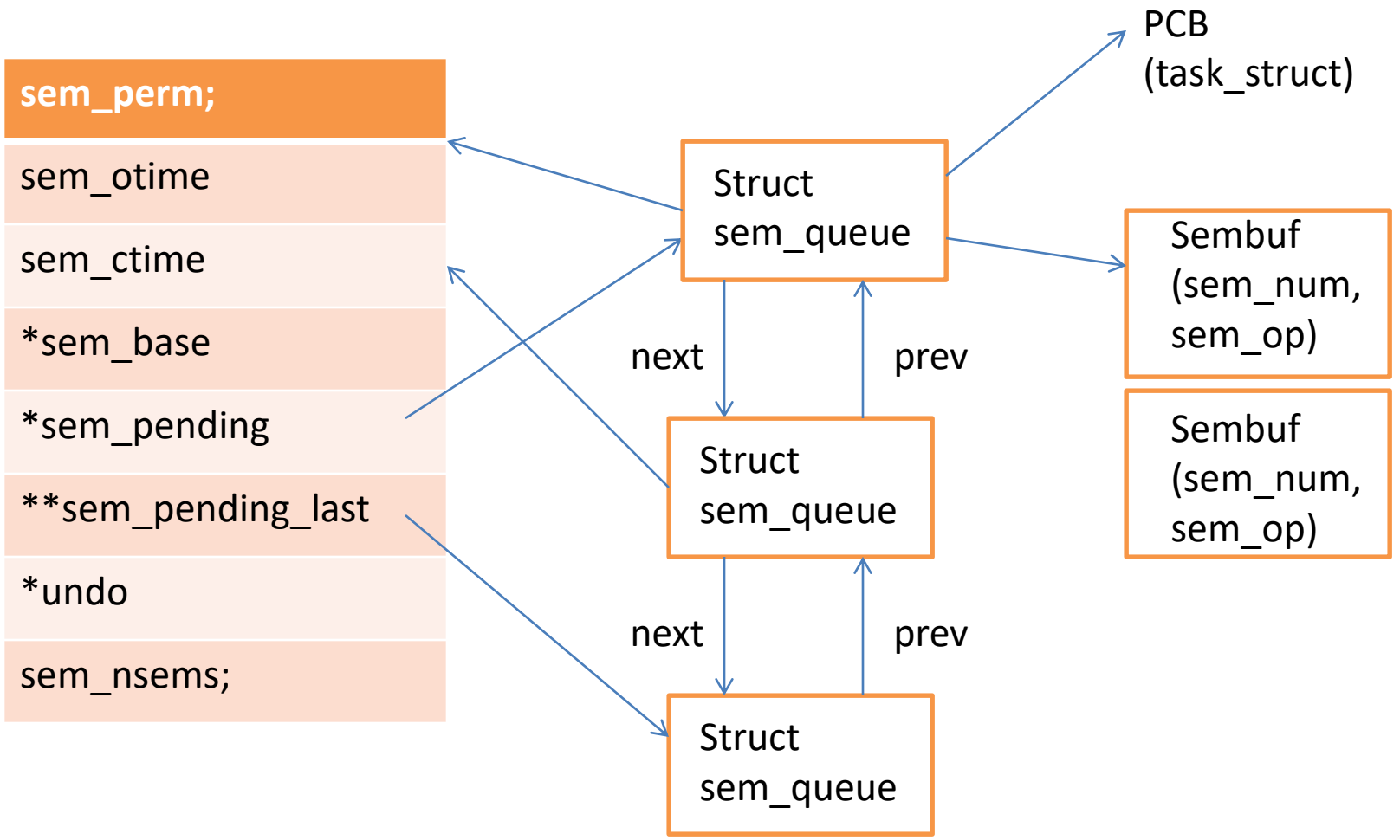
Concept



- sem_op is 0 (special case)
- Check if S ==0
- **If true, return (proceed)**
- **Else (S is positive)**
- **Block**

semzcnt


sem_perm;
sem_otime
sem_ctime
*sem_base
*sem_pending
**sem_pending_last
*undo
sem_nsems;
Semval[0]
Sempid[0]
Semncnt[0]
Semzcnt[0]
Semval[1]
Sempid[1]
Semncnt[1]
Semzcnt[1]

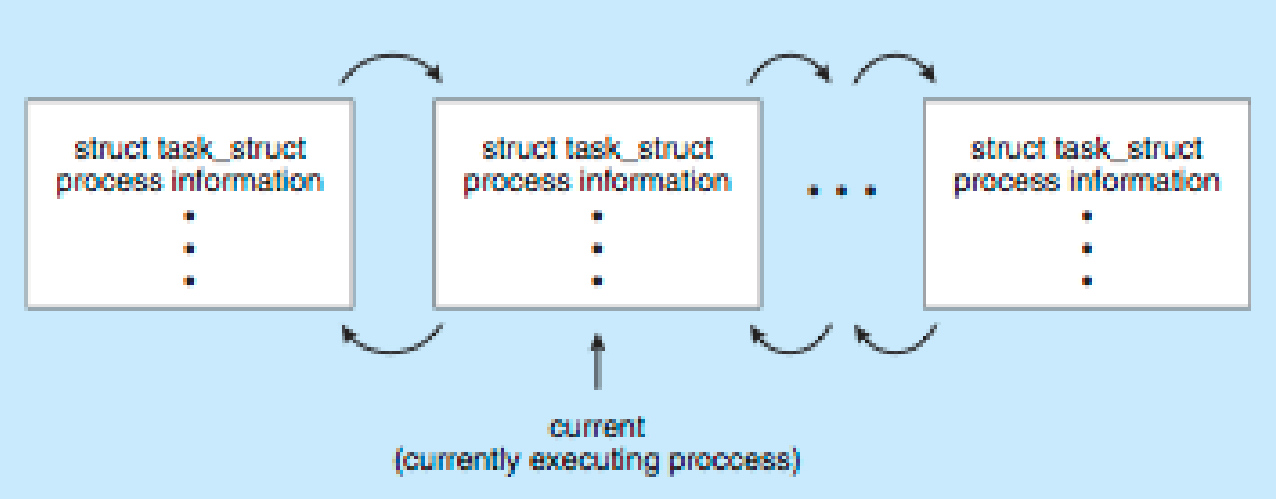

sem_perm;
sem_otime
sem_ctime
*sem_base
*sem_pending
**sem_pending_last
*undo
sem_nsems;
Struct sem_queue
Struct sem_queue
Struct sem_queue
next
prev
next
prev
PCB (task_struct)
Sembuf (sem_num, sem_op)
Sembuf (sem_num, sem_op)

Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this pro */
```

Doubly linked list



Sembuf

```
struct sembuf
{
        ushort sem_num;
        short  sem_op;
        short sem_flg;
}
```

Sub semaphore

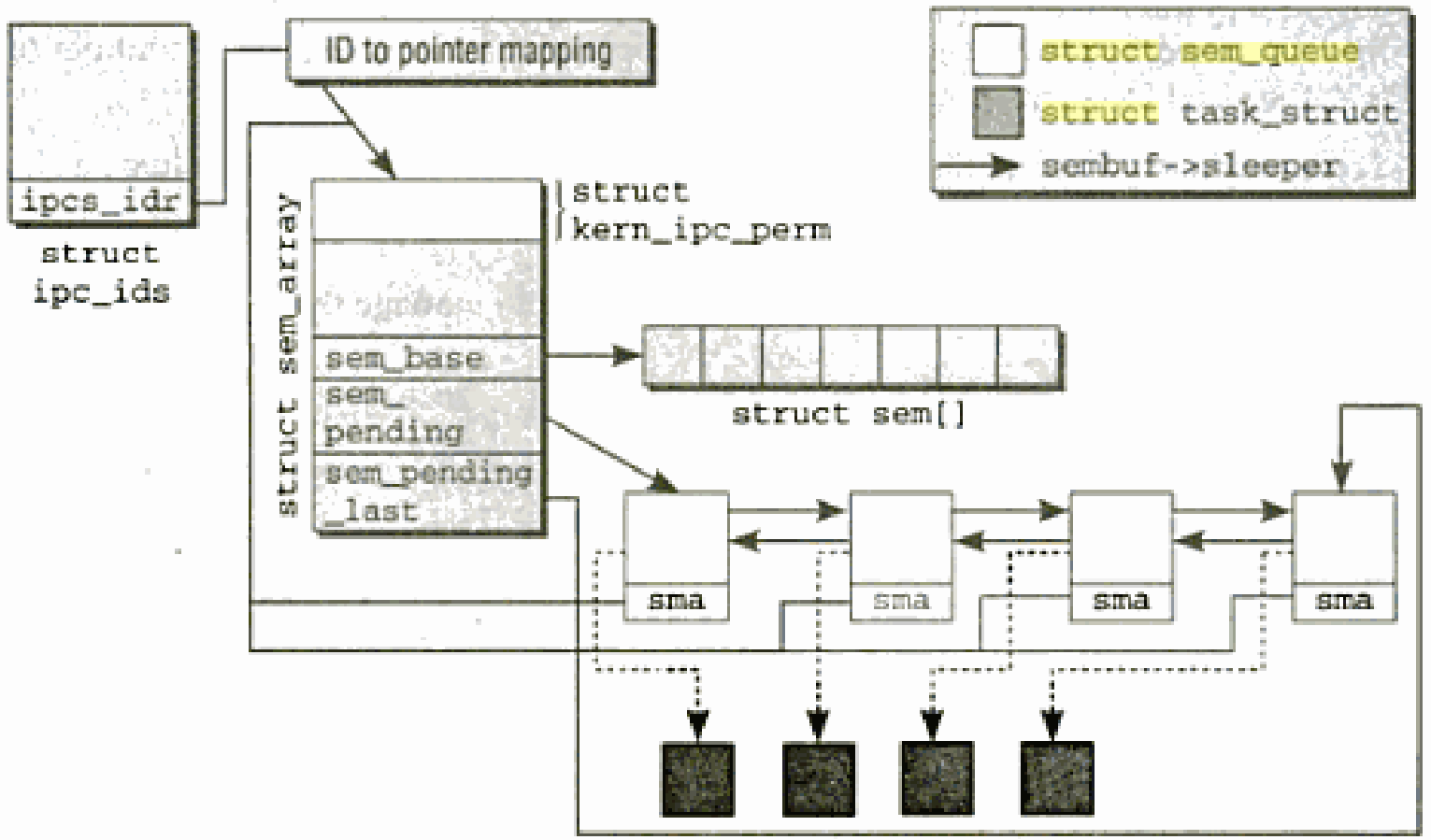
0, IPC_NOWAIT, SEM_UNDO

int semop(int semid, struct sembuf *sops, unsigned nsops);

```
/* One queue for each sleeping process in the system. */
struct sem_queue {
        struct sem_queue *      next;    /* next entry in the queue */
        struct sem_queue **     prev;    /* previous entry in the queue, *(q->prev) == q */
        struct task_struct*  sleeper; /* this process */
        struct sem_undo *  undo;    /* undo structure */
        int         pid;        /* process id of requesting process */

        struct sem_array *  sma;    /* semaphore array for operations */

        struct sembuf *         sops;    /* array of pending operations */
        int                     nsops;   /* number of operations */
        int                     alter;   /* operation will alter semaphore */
};
```


ID to pointer mapping
struct sem_queue
struct task_struct
sembuf->sleeper
ipcs_idr
struct
ipc_ids
struct sem_array
struct
kern_ipc_perm
sem_base
sem_
pending
sem_pending
_last
struct sem[]
sma
sma
sma
sma

```
struct sem_undo {
        struct sem_undo *  proc_next;        /* next entry on this process */
        struct sem_undo *  id_next;   /* next entry on this semaphore set */
        int                semid;            /* semaphore set identifier */
        short *     semadj;   /* array of adjustments, one per semaphore */
};
```

IPC_STAT/IPC_SET

Getting the status of semaphore variable

```
Main()
{
Struct semid_ds stat;
Semid=semget()
semctl(semid,0, IPC_STAT, &stat);
Printf("number of sub-semaphores=%d",stat.sem_nsems);
Printf("owner's userid=%d",stat.sem_perm.uid);
Printf("semop time=%d",stat.sem_otime);
}
```

Setting the status of semaphore variable

```
{
Stat.sem_perm.uid=102;
Stat.sem_perm.gid=102;
semctl(semid,0, IPC_SET &stat);

}
```

```
union semun {
        int val;                    /* value for SETVAL */
        struct semid_ds *buf;       /* buffer for IPC_STAT & IPC_SET */
        unsigned short *array;              /* array for GETALL & SETALL */
        }
```

Prototype of semctl

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

Semctl(semid, 0, GETNCNT,0)

Returns the number of processes waiting on semid (sub-sem=0)

If S <|sem_op|

Semctl(semid, 0, GETZCNT,0)

Returns the number of processes waiting on semid (sub-sem=0)

If sem_op=0