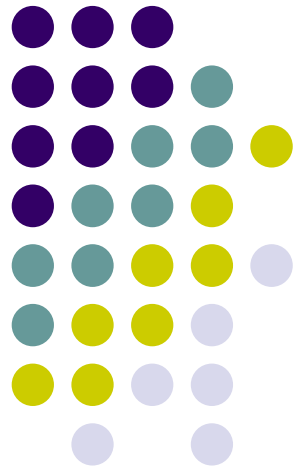


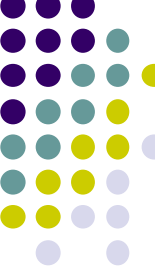


# Semáforos



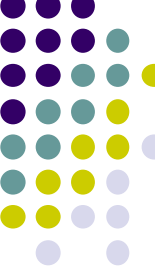
Eloy García Martínez  
Lorena Ascensión Olivero





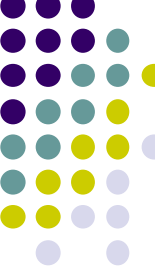
# Introducción

- El concepto de semáforo nace de la necesidad de crear un sistema operativo en el que puedan trabajar procesos cooperantes -> Dijkstra(1965)
- No es un mecanismo de comunicación sino de sincronización.
- Son utilizados para controlar el acceso a los recursos.



# Introducción (1)

- Un semáforo básico es una variable entera y dos operaciones atómicas que la manejan (sin interrupciones):
- Espera (P): Se usa cuando un proceso quiere acceder a un recurso compartido y puede ocurrir:
  - Si la variable entera es positiva, coge el recurso y decrementa dicho valor.
  - En caso de que el valor sea nulo el proceso se duerme y espera a ser despertado.
- Señal (V): Se utiliza para indicar que el recurso compartido está libre y despertar a los procesos que estén esperando por el recurso.



# Estructura sem

- Estructura que representa cada semáforo en el sistema.

```
85 struct sem {  
86     int semval; // valor actual  
87     int sempid; // pid de la última operación  
88};
```



# Estructura sembuf

- Se trata de una estructura que indica una operación a realizar sobre un semáforo en particular: incrementar, decrementar o esperar un valor nulo.
- Se usa en la llamada semop.

```
38 struct sembuf {  
39     unsigned short  sem_num; //Número de semáforo  
     del grupo  
40     short  sem_op; /*Operación sobre el semáforo*/  
41     short  sem_flg; /* Opciones */  
42 };
```



# Estructura sem\_queue

Se trata de la estructura de un nodo de la cola de procesos dormidos.

```
103 struct sem_queue {
104     struct sem_queue *   next;   /* Próxima entrada en la cola */
105     struct sem_queue **  prev;   /* Entrada previa en la cola */
106     struct task_struct*  sleeper; /* Proceso actual */
107     struct sem_undo *    undo;   /* Operaciones anuladas en caso de
                                     terminación */
108     Int pid;                /* Identificador del proceso solicitado */
109     int status;            /* Estado de terminación de la operación
*/
110 struct sem_array * sma; /* Conjunto de semáforos*/
111     int id;                /* Identificador de semáforo interno */
111     struct sembuf *sops; /* Estructura de operaciones pendientes*/
113     Int nsops;            /* Número de operaciones */
114     int alter;           /* Si se ha modificado el vector */
115 };
```

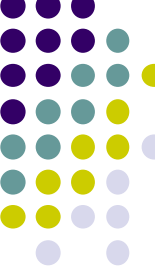


# Estructura sem\_array

Estructura de control asociada a cada Conjunto de semáforos existentes en el sistema.

Contiene información del sistema, punteros a las operaciones a realizar sobre el grupo, y un puntero hacia las estructuras sem que contienen información de cada semáforo.

```
91 struct sem_array {
92     struct kern_ipc_perm sem_perm; /* Permisos */
93     time_t sem_otime; /* Fecha de la última operación*/
94     time_t sem_ctime; /* Fecha del último cambio */
95     struct sem *sem_base; /* Puntero al primer semáforo del grupo*/
/*punteros a las operaciones a realizar sobre el grupo*/
96     struct sem_queue *sem_pending;//Operaciones pendientes
97     struct sem_queue **sem_pending_last;//Última operación en espera
98     struct sem_undo *undo; /* Operaciones anuladas en caso de
terminación*/
99     unsigned long sem_nsems; /* Número de semáforos del grupo*/
100 };
```



# Estructura semun

Se utiliza en la llamada `semctl` para almacenar o recuperar informaciones sobre los semáforos.

```
45 union semun {  
46 int val; /* Valor para SETVAL */  
47 struct semid_ds __user *buf; /* Memoria de datos para IPC_STAT e  
                                     IPC_SET */  
48 unsigned short __user *array; /* Tabla para GETALL y SETALL */  
49 struct seminfo __user * __buf; /* Memoria de datos para IPC_INFO */  
50 void __user * __pad; /* Puntero de alineación de la estructura */  
51 };
```

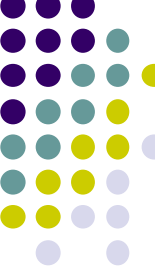




# Estructura seminfo

**Estructura que permite conocer los valores límite o actuales del sistema mediante una llamada a semctl**

```
53 struct seminfo {  
54     int semmap; //No se usa  
55     int semmni; //Número máximo de grupos de semáforos  
56     int semmns; //Número máximo de semáforos  
57     int semmnu; //No se usa  
58     int semmsl; //Número máximo de semáforos por grupo  
59     int semopm; //No se usa  
60     int semume; //No se usa  
61     int semusz; //Número de grupos de semáforos actualmente definidos  
62     int semvmx; //Valor máximo del contador de semáforos  
63     int semaem; //Número de semáforos actualmente definidos  
64 };
```

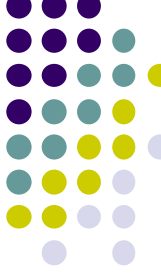


# Estructura sem\_undo

Sem\_undo es un nodo de una estructura que contiene las acciones anuladas en caso de terminación.

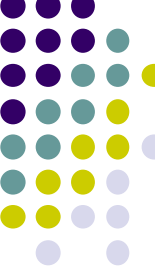
```
118 struct sem_undo {  
119 struct sem_undo * proc_next; /* Proxima Entrada de este  
proceso */  
120 struct sem_undo * id_next; // Proxima entrada de este conj. de  
semaforos  
121 int semid; /* Identificador del conjunto de semaforos */  
122 short * semadj; /* Vector de ajustes, uno por semaforo */  
123 };
```

# Llamadas al sistema para semáforos



Tendremos tres llamadas al sistema diferentes:

- **Semget**: creación y búsqueda de grupos de semáforos
- **Semctl**: control de los semáforos
- **Semop**: operaciones sobre los semáforos



# Semget

- Crea un grupo de semáforos o bien recupera el identificador de uno que ya existe.
- sys semget (key\_t key, int nsems, int semflg)
  - key: identificador del grupo de semáforos.
  - nsems: número de semáforos en el grupo.
  - semflag: máscara de operación.

# Sys\_semget



```
310 asmlinkage long sys_semget(key_t key, int nsems, int semflg)
```

```
311{
```

```
312    struct ipc_namespace *ns;
```

```
313    struct ipc_ops sem_ops;
```

```
314    struct ipc_params sem_params;
```

```
315
```

```
316    ns = current->nsproxy->ipc_ns;
```

```
317
```

**/\* Comprueba que el número de semaforos que pide sea un valor entre 0 y el maximo por conjunto (SEMMSL)\*/**

```
318    if (nsems < 0 || nsems > ns->sc_semmsl)
```

```
319        return -EINVAL;
```

# Sys\_Semget(1)



**//Indica las operaciones asociadas con el semaforo**

```
321    sem_ops.getnew = newary;  
322    sem_ops.associate = sem_security;  
323    sem_ops.more_checks = sem_more_checks;
```

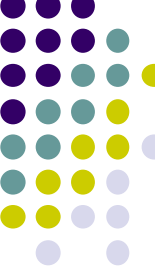
**//Guarda los parámetros del conjunto de semaforos**

```
325    sem_params.key = key;  
326    sem_params.flg = semflg;  
327    sem_params.u.nsems = nsems;  
328
```

**//Llama a ipcget para realizar la tarea**

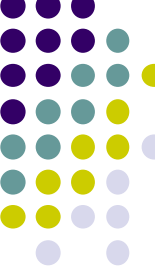
```
329    return ipcget(ns, &sem_ids(ns), &sem_ops, &sem_params);  
330}
```

# Ipcget



**/\*Si KEY = IPC\_PRIVATE llama a ipcget\_new, en caso contrario llama a ipcget\_public. La primera crea un nuevo conjunto de semáforos, mientras que la segunda busca el conjunto de semáforos y si no existe los crea.\*/**

```
755int ipcget(struct ipc_namespace *ns, struct ipc_ids *ids,  
756    struct ipc_ops *ops, struct ipc_params *params)  
757{  
758    if (params->key == IPC_PRIVATE)  
759        return ipcget_new(ns, ids, ops, params);  
760    else  
761        return ipcget_public(ns, ids, ops, params);  
762}
```



# ipcget\_new

**/\*La función ipcget\_new realiza los siguientes pasos: reserva memoria mediante la función idr\_pre\_get, llama a la función getnew (enlace a newary) para crear el conjunto de semáforos dentro de un cerrojo de escritura\*/**

```
251static int ipcget_new(struct ipc_namespace *ns, struct ipc_ids *ids,  
252        struct ipc_ops *ops, struct ipc_params *params)  
...  
256    err = idr_pre_get(&ids->ipcs idr, GFP_KERNEL);  
...  
261    down_write(&ids->rw_mutex);  
262    err = ops->getnew(ns, params);  
263    up_write(&ids->rw_mutex);
```

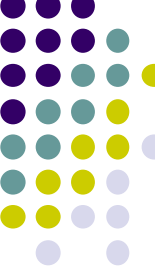




# ipcget\_public

**/\*La función ipcget\_public : Reserva memoria mediante idr\_pre\_get, crea un cerrojo de escritura y llama a la función ipc\_findkey: que se encarga de encontrar si existe un conjunto con dicha clave \*/**

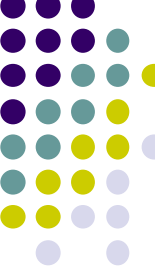
```
315static int ipcget_public(struct ipc_namespace *ns, struct ipc_ids *ids,  
316        struct ipc_ops *ops, struct ipc_params *params)  
317{  
    ...  
322    err = idr_pre_get(&ids->ipcs_idr, GFP_KERNEL);  
    ...  
328    down_write(&ids->rw_mutex);  
329    ipcp = ipc_findkey(ids, params->key);
```



# lpcget\_public(1)

**/\*Si la clave no está siendo utilizada y la máscara IPC\_CREAT está activa, se crea una nueva entrada y se llama a getnew (newary).\*/**

```
330     if (ipcp == NULL) {  
332         if (!(flg & IPC_CREAT))  
333             err = -ENOENT;  
334         else if (!err)  
335             err = -ENOMEM;  
336         else  
337             err = ops->getnew(ns, params);  
338     } else {
```



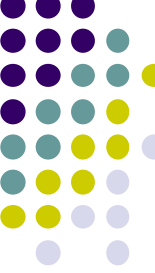
# lpcget\_public(2)

**/\*Si existe la clave suministrada:**

- Si esta activada IPC\_CREAT y IPC\_EXCL nos dará error,**
- Realiza ciertas comprobaciones con more\_checks y ipc\_check\_perms**
- Elimina el bloqueo de escritura y devuelve el identificador.\*/**

```
341     if (flg & IPC_CREAT && flg & IPC_EXCL)
342         err = -EEXIST;
343     else {
344         err = 0;
345         if (ops->more_checks)
346             err = ops->more_checks(ipcp, params);
347         if (!err) //ipc_check_perms retorna el identificador
352             err = ipc_check_perms(ipcp, ops, params);
353     }
354     ipc_unlock(ipcp);
355 }
356 up_write(&ids->rw_mutex); //Elimina el bloqueo de escritura
```

# Newary



**Crea un nuevo identificador sujeto a la disponibilidad de entradas libres en la tabla que gestiona el núcleo para los semáforos. Crea e inicializa un nuevo conjunto de semáforos**

```
231static int newary(struct ipc_namespace *ns, struct ipc_params *params){
```

```
...
```

**//Obtiene los parámetros del conjunto de semáforos**

```
237    key_t key = params->key;
```

```
238    int nsems = params->u.nsems;
```

```
239    int semflg = params->flg;
```

**//comprueba que el numero de semáforos que pide es correcto y si hay disponibilidad**

```
241    if (!nsems)
```

```
242        return -EINVAL;
```

```
243    if (ns->used_sems + nsems > ns->sc_semmns)
```

```
244        return -ENOSPC;
```

# Newary(1)



**//calculamos el tamaño que ocupará y asigna recursos**

```
246    size = sizeof (*sma) + nsems * sizeof (struct sem);  
247    sma = ipc_rcu_alloc(size);  
248    if (!sma) {  
249        return -ENOMEM;  
250    }  
251    memset (sma, 0, size);
```

**/\*Inicializa los parámetros del nuevo conjunto de semáforos\*/**

```
253    sma->sem_perm.mode = (semflg & S_IRWXUGO);  
254    sma->sem_perm.key = key;  
255  
256    sma->sem_perm.security = NULL;
```



# Newary(2)

```
257     retval = security sem alloc(sma);  
258     if (retval) {  
259         ipc_rcu_putref(sma);  
260         return retval;  
261     }  
262
```

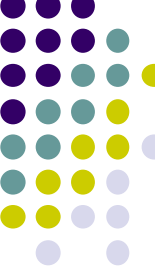
**/\*Obtiene el identificador y añade la nueva entrada\*/**

```
263     id = ipc_addid(&sem_ids(ns), &sma->sem_perm, ns->sc_semmni);  
264     if (id < 0) {  
265         security sem free(sma);  
266         ipc_rcu_putref(sma);  
267         return id;  
268     }
```

**/\*se actualiza el valor de la variable que lleva el número de semáforos utilizados\*/**

```
269     ns->used_sems += nsems;
```

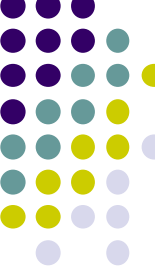
# Newary(3)



Ahora va a inicializar el resto de los campos de la estructura

`sem_array`

```
271     sma->sem_perm.id = sem_buildid(id, sma->sem_perm.seq);  
272     sma->sem_base = (struct sem *) &sma[1];  
273     /* sma->sem_pending = NULL; */  
274     sma->sem_pending_last = &sma->sem_pending;  
275     /* sma->undo = NULL; */  
276     sma->sem_nsems = nsems; /*Número de semáforos del grupo*/  
277     sma->sem_ctime = get_seconds(); /* Fecha del último cambio*/  
278     sem_unlock(sma); /*Desbloqueo*/  
279  
280     return sma->sem_perm.id;  
281}
```



# Semctl

Permite la modificación, consulta o supresión de un grupo de semáforos.

- sys semctl(int semid, int semnum, int cmd, union semun arg)
  - semid: identificador de semáforo válido
  - semnum: representa el ordinal o bien el número de semáforos.
  - cmd: operación a realizar



# sys\_semctl



```
936asmlinkage long sys_semctl (int semid, int semnum, int cmd, union semun arg){
```

```
...
```

```
//comprobamos el valor de semid
```

```
942    if (semid < 0)  
943        return -EINVAL;
```

```
...
```

```
948    switch(cmd) {  
949    case IPC_INFO:  
950    case SEM_INFO:  
951    case IPC_STAT:  
952    case SEM_STAT:  
953        err = semctl_nolock(ns, semid, cmd, version, arg);  
954        return err;
```

# sys\_semctl(1)



```
955     case GETALL:  
956     case GETVAL:  
957     case GETPID:  
958     case GETNCNT:  
959     case GETZCNT:  
960     case SETVAL:  
961     case SETALL:  
962         err = semctl_main(ns,semid,semnum,cmd,version,arg);  
963         return err;  
964     case IPC_RMID:  
965     case IPC_SET:  
966         down_write(&sem_ids(ns).rw_mutex);  
967         err = semctl_down(ns,semid,semnum,cmd,version,arg);  
968         up_write(&sem_ids(ns).rw_mutex);
```



# Semctl\_nolock

La llama `sys semctl()` para realizar las operaciones:

- `IPC_INFO` y `SEM_INFO`: causan una antememoria temporal `seminfo` para que sea inicializada y cargada con los datos estadísticos sin cambiar el semáforo.
- `IPC_STAT` y `SEM_STAT`: permite obtener las informaciones respecto a un grupo de semáforo. `Semunm` se ignora, `arg` es un puntero a la zona que contiene las informaciones. El proceso debe tener derechos de lectura para realizar la operación.

```
585static int semctl_nolock(struct ipc_namespace *ns, int semid,  
586                        int cmd, int version, union semun arg)  
587{  
588    int err = -EINVAL;  
589    struct sem_array *sma;
```

# Semctl\_nolock(1)

//En el caso de IPC\_INFO y SEM\_INFO

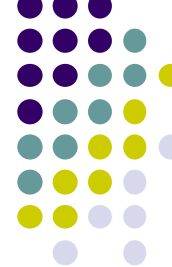
```
592     case IPC_INFO:  
593     case SEM_INFO: {
```

...

**//Inicializa los campos de la estructura seminfo**

```
602     memset(&seminfo,0,sizeof(seminfo));  
603     seminfo.semmni = ns->sc_semmni;  
604     seminfo.semmns = ns->sc_semmns;  
605     seminfo.semmsl = ns->sc_semmsl;  
606     seminfo.semopm = ns->sc_semopm;  
607     seminfo.semvmx = SEMVMX;  
608     seminfo.semmnu = SEMMNU;  
609     seminfo.semmap = SEMMAP;  
610     seminfo.semume = SEMUME;  
611     down_read(&sem_ids(ns).rw_mutex);
```

# Semctl\_nolock(2)



//actualizamos el numero de grupos de semaforos definidos y de semaforos  
actualmente

```
612     if (cmd == SEM INFO) {  
613         seminfo.semusz = sem_ids(ns).in_use;  
614         seminfo.semaem = ns->used_sems;  
615     } else {  
616         seminfo.semusz = SEMUSZ;  
617         seminfo.semaem = SEMAEM;  
618     }  
619     max_id = ipc_get_maxid(&sem_ids(ns));  
620     up_read(&sem_ids(ns).rw_mutex);  
621     if (copy_to_user (arg.__buf, &seminfo, sizeof(struct seminfo)))  
622         return -EFAULT; //direccion de arg no valida  
623     return (max_id < 0) ? 0: max_id;  
624 }
```

# Semctl\_nolock(3)



```
//En el caso de IPC_STAT y SEM_STAT
625     case IPC_STAT:
626     case SEM_STAT:{
...
631         if (cmd == SEM_STAT) {
632             sma = sem_lock(ns, semid);
633             if (IS_ERR(sma))
634                 return PTR_ERR(sma);
635             id = sma->sem_perm.id;
636         } else {
637             sma = sem_lock_check(ns, semid);
638             if (IS_ERR(sma))
639                 return PTR_ERR(sma);
640             id = 0;
641         }
```



# Semctl\_nolock(4)

**/\*comprobar si existen permisos para el proceso que llama\*/**

```
644         if (ipcperms (&sma->sem_perm, S_IRUGO))  
645             goto out_unlock;
```

...

**/\*los valores sem\_otime, sem\_ctime, y sem\_nsems son copiados en una pila de antememoria. \*/**

```
653         kernel to ipc64_perm(&sma->sem_perm, &tbuf.sem_perm);  
654         tbuf.sem_otime = sma->sem_otime;  
655         tbuf.sem_ctime = sma->sem_ctime;  
656         tbuf.sem_nsems = sma->sem_nsems;
```

**Los datos son entonces copiados al espacio de usuario después de tirar con el spinlock.\*/**

```
657         sem_unlock(sma);  
658         if (copy semid to user (arg.buf, &tbuf, version))  
659             return -EFAULT;  
660         return id;
```

# semctl\_main



Llamado por sys semctl(). Anteriormente a realizar alguna de las siguientes operaciones, `semctl_main()` cierra el spinlock global del semáforo y valida la ID del conjunto de semáforos y los permisos. El spinlock es liberado antes de retornar.

**GETALL** - carga los actuales valores del semáforo en una antememoria temporal del núcleo y entonces los copia fuera del espacio de usuario.

**SETALL**- copia los valores del semáforo desde el espacio de usuario en una antememoria temporal, y entonces en el conjunto del semáforo.

**GETVAL** - en el caso de no error, el valor de retorno para la llamada al sistema es establecido al valor del semáforo especificado.

**GETPID** - en el caso de no error, el valor de retorno para la llamada al sistema es establecido al pid asociado con las última operación del semáforo.

**GETNCNT** - número de procesos esperando en el semáforo siendo menor que cero.

**GETZCNT** - número de procesos esperando en el semáforo estando establecido a cero.

**SETVAL** - funciones realizadas después de validar el nuevo valor del semáforo (establecer valor del semaforo,)



# Semctl\_main(1)



```
671static int semctl_main(struct ipc_namespace *ns, int semid, int semnum,  
672          int cmd, int version, union semun arg){  
    ...  
681    sma = sem lock check(ns, semid); //Bloqueo  
    ...  
685    nsems = sma->sem_nsems; //Numero de semáforos del grupo  
    ...  
//comprueba los permisos  
688    if (ipcperms (&sma->sem_perm,  
    (cmd==SETVAL||cmd==SETALL)?S_IWUGO:S_IRUGO))  
689        goto out_unlock;
```

# Semctl\_main(2)



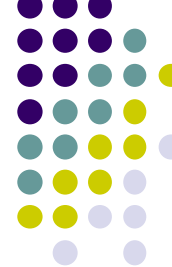
```
...
696     switch (cmd) {
/*carga los actuales valores del semáforo en una antememoria temporal del
   núcleo y entonces los copia al espacio de usuario.*/
697     case GETALL:
698     {

...
723         for (i = 0; i < sma->sem_nsems; i++)
724             sem_io[i] = sma->sem_base[i].semval;
725         sem_unlock(sma);
726         err = 0;
727         if(copy to user(array, sem_io, nsems*sizeof(ushort)))
728             err = -EFAULT;
729         goto out_free;
730     }
```

# Semctl\_main(3)

```
731     case SETALL:
732     {
733         int i;
734         struct sem_undo *un;
735
736         ipc_rcu_getref(sma);
737         /*Copia los valores del semáforo desde el espacio de usuario en una
738         antememoria temporal. El spinlock es quitado mientras se copian los
739         valores*/
740         sem_unlock(sma);
741         ...
742         if (copy_from_user (sem_io, arg.array, nsems*sizeof(ushort))) {
743             ipc_lock_by_ptr(&sma->sem_perm);
744             ipc_rcu_putref(sma);
745             sem_unlock(sma);
746             err = -EFAULT;
747             goto out_free;
748         }
```

# Semctl\_main(4)



//Los valores son copiados en le conjunto del semáforo

```
774         for (i = 0; i < nsems; i++)  
775             sma->sem_base[i].semval = sem_io[i];
```

//Los ajustes del semáforo de la cola deshacer para el conjunto del semáforo son limpiados

```
776         for (un = sma->undo; un; un = un->id_next)  
777             for (i = 0; i < nsems; i++)  
778                 un->semadj[i] = 0;  
779             sma->sem_ctime = get_seconds();
```

***/\*update\_queue es llamada para recorrer la cola de semops (operaciones del semáforo) pendientes y mirar por alguna tarea que pueda ser completada. Cualquier tarea pendiente que no sea más bloqueada es despertada\*/***

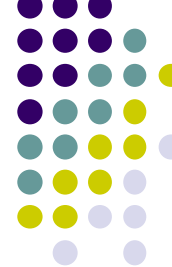
```
781         update_queue(sma);  
782         err = 0;  
783         goto out_unlock;  
784     }
```



# Semctl\_main(5)

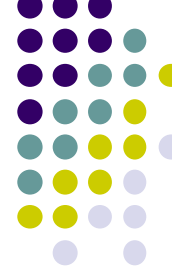
```
787     err = -EINVAL;  
//Comprueba el valor de semnum (posicion en el grupo de semáforos)  
788     if(semnum < 0 || semnum >= nsems)  
789         goto out_unlock;  
790  
791     curr = &sma->sem_base[semnum];  
792  
793     switch (cmd) {  
794     case GETVAL://Devuelve el valor del semáforo  
795         err = curr->semval;  
796         goto out_unlock;  
797     case GETPID://Devuelve el pid asociado con la última operación del  
semáforo  
798         err = curr->sempid;  
799         goto out_unlock;
```

# Semctl\_main(6)



```
800     case GETNCNT: //numero de procesos esperando para que el valor
      del semval aumente
801         err = count_semncnt(sma,semnum);
802         goto out_unlock;
803     case GETZCNT://numero de procesos esperando para que el valor
      del semval sea cero
804         err = count_semzcnt(sma,semnum);
805         goto out_unlock;
806     case SETVAL:
807     {
808         int val = arg.val;
809         struct sem_undo *un;
810         err = -ERANGE;
//Valida el nuevo valor del semáforo
811         if (val > SEMVMX || val < 0)
812             goto out_unlock;
813
```

# Semctl\_main(7)



**/\*La cola de deshacer es buscada para cualquier ajuste en este semáforo.  
Cualquier ajuste que sea encontrado es reinicializado a cero\*/**

```
814         for (un = sma->undo; un; un = un->id_next)  
815             un->semadj[semnum] = 0;
```

**/\*El valor del semáforo es establecido al valor suministrado\*/**

```
816         curr->semval = val;  
817         curr->sempid = task_tgid_vnr(current);  
818         sma->sem_ctime = get_seconds();  
820         update_queue(sma);  
821         err = 0;  
822         goto out_unlock;  
823     }  
824 }
```

```
825out unlock:
```

```
826     sem_unlock(sma);
```

```
827out free:
```

```
828     if(sem_io != fast_sem_io)
```

```
829         ipc_free(sem_io, sizeof(ushort)*nsems);
```

```
830     return err;
```

# semctl\_down

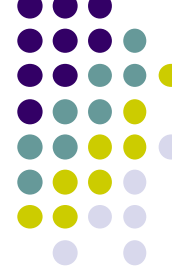


- Suministra las operaciones IPC\_RMID y IPC\_SET de la llamada al sistema `sys_semctl()`. La ID del conjunto de semáforos y los permisos de acceso son verificadas en ambas operaciones, y en ambos casos, el spinlock global del semáforo es mantenido a lo largo de la operación.
- **IPC\_RMID**- llama a freeary() para borrar el conjunto del semáforo.
- **IPC\_SET** - actualiza los elementos `uid`, `gid`, `mode`, y `ctime` del conjunto de semáforos.

```
873static int semctl_down(struct ipc_namespace *ns, int semid, int semnum,  
874                int cmd, int version, union semun arg)  
875{
```



# Semctl\_down(1)



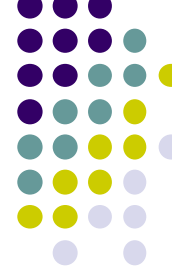
**//comprobamos si el proceso tiene o no derechos para realizar la operacion**

```
900     if (current->euid != ipcp->cuid &&  
901         current->euid != ipcp->uid && !capable(CAP_SYS_ADMIN)) {  
902         err=-EPERM;  
903         goto out_unlock;  
904     }
```

...

```
910     switch(cmd){  
911         case IPC_RMID: //llama a freeary  
912             freeary(ns, ipcp);  
913             err = 0;  
914             break;  
915         case IPC_SET: //modificamos los siguientes campos  
916             ipcp->uid = setbuf.uid;  
917             ipcp->gid = setbuf.gid;
```

# Semctl\_down(2)



```
918         ipcp->mode = (ipcp->mode & ~S_IRWXUGO)
919                 | (setbuf.mode & S_IRWXUGO);
920         sma->sem_ctime = get_seconds();
921         sem_unlock(sma);
922         err = 0;
923         break;
924     default:
925         sem_unlock(sma); //Liberamos el cerrojo
926         err = -EINVAL;
927         break;
928     }
929     return err;
931out unlock:
932     sem_unlock(sma);
933     return err;
934 }
```

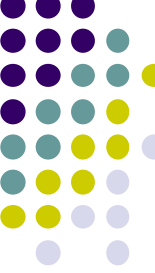
# freeary



## Libera un conjunto de semáforos

```
527static void freeary(struct ipc_namespace *ns, struct kern_ipc_perm *ipcp)  
528{  
529    struct sem_undo *un;  
530    struct sem_queue *q;  
531    struct sem_array *sma = container_of(ipcp, struct sem_array, sem_perm);  
532  
/*Invalida las estructuras undo (operaciones anuladas en caso de  
terminación) que existen para este conjunto de semaforos*/  
//Estas seran liberadas en exit_sem() o durante la siguiente llamada a semop.  
537    for (un = sma->undo; un; un = un->id_next)  
538        un->semid = -1;  
539
```

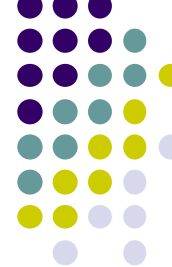
# Freeary(1)



**/\*Levanta todos los procesos pendientes y los deja como que han fallado con EIDRM. (el grupo de semaforos ha sido borrado)\*/**

```
541     q = sma->sem_pending;  
542     while(q) {  
543         struct sem_queue *n;  
545         q->prev = NULL;  
546         n = q->next;  
547         q->status = IN_WAKEUP;  
548         wake_up_process(q->sleeper); /* doesn't sleep */  
549         smp_wmb();  
550         q->status = -EIDRM; /* hands-off q */  
551         q = n;  
552     }
```

# Freeary(2)



```
554 /* Borra el conjunto de semaforos del vector de ID */  
555     sem_rmid(ns, sma);  
556     sem_unlock(sma); //Desbloqueo  
557  
558     ns->used_sems -= sma->sem_nsems;  
559     security_sem_free(sma);  
560     ipc_rcu_putref(sma);  
561 }
```

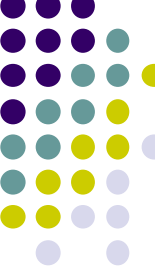
# Update\_queue



Recorre la cola de semops pendientes para un conjunto de semáforo y llama a try\_atomic\_semop() para determinar qué secuencias de las operaciones de los semáforos serán realizadas.

```
434 static void update_queue (struct sem_array * sma){
436     int error;
437     struct sem_queue * q;
439     q = sma->sem_pending; //obtiene las operaciones pendientes
440     while(q) { //se llama a try_atomic_semop para realizar la tarea
441         error = try_atomic_semop(sma, q->sops, q->nsops,
442                                 q->undo, q->pid);
445         if (error <= 0) {
446             struct sem_queue *n;
//quitamos el proceso de la cola de dormidos y los despertamos
447             remove_from_queue(sma,q);
448             q->status = IN_WAKEUP;
```

# Update\_queue(1)



*//si se modifoco el array comenzamos desde la cabecera*

```
460         if (q->alter)
461             n = sma->sem_pending;
462         else
463             n = q->next;
//lo levantamos
464         wake up process(q->sleeper);
468         smp_wmb();
469         q->status = error;
470         q = n;
471     } else {
472         q = q->next;
473     }
474 }
475}
476
```



# semop

- Para realizar las operaciones sobre los semáforos que hay asociados bajo un identificador (incremento, decremento o espera de nulidad)
- sys semop (int semid, struct sembuf \_\_user \*tsops, unsigned nsops)
  - semid: identificador del semáforo
  - tsops: puntero a un vector de operaciones
  - nsops: número de operaciones a realizar en esta llamada



# sys\_semop



La llamada al sistema `sys_semop` realiza una llamada a `sys_semtimedop`

```
1254asmlinkage long sys_semop (int semid, struct sembuf __user *tsops, unsigned  
nsops)  
1255{  
1256    return sys_semtimedop(semid, tsops, nsops, NULL);  
1257}  
1258
```



# Sys\_semtimedop

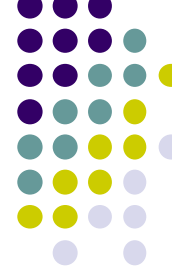
```
1090asmlinkage long sys_semtimedop(int semid, struct sembuf __user *tsops,  
1091 unsigned nsops, const struct timespec __user *timeout){
```

...

**//Valida los parámetros de la llamada**

```
1105 if (nsops < 1 || semid < 0)  
1106     return -EINVAL;  
1107 if (nsops > ns->sc_semopm)  
1108     return -E2BIG;  
1109 if(nsops > SEMOPM_FAST) {  
1110     sops = kmalloc(sizeof(*sops)*nsops,GFP_KERNEL);  
1111     if(sops==NULL)  
1112         return -ENOMEM;  
1113 }
```

# Sys\_semtimedop(1)



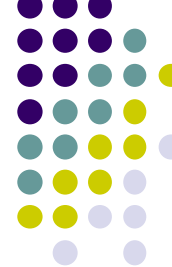
**//copia del espacio de usuario a una antememoria temporal**

```
1114     if (copy_from_user (sops, tsops, nsops * sizeof(*tsops))) {  
1115         error = -EFAULT;  
1116         goto out_free; }  
1131     max = 0;
```

**//Obtiene el numero máximo de semáforos utilizados en los grupos,  
establece undos a 1 si la etiqueta SEM\_UNDO está activada y establece  
alter a 1 si alguna de las operaciones modifica el valor de los semáforos**

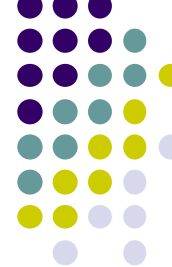
```
1132     for (sop = sops; sop < sops + nsops; sop++) {  
1133         if (sop->sem_num >= max)  
1134             max = sop->sem_num;  
1135         if (sop->sem_flg & SEM_UNDO)  
1136             undos = 1;  
1137         if (sop->sem_op != 0)  
1138             alter = 1;  
1139     }
```

# Sys\_semimedop(2)



```
...
1151     sma = sem lock check(ns, semid); //Cerrojo
...
1167     if (max >= sma->sem_nsems) //Chequea el número máximo de
        semáforos por grupo
1168         goto out_unlock_free;
1169
1170     error = -EACCES;
//Chequea permisos
1171     if (ipcperms(&sma->sem_perm, alter ? S_IWUGO : S_IRUGO))
1172         goto out_unlock_free;
1174     error = security sem semop(sma, sops, nsops, alter);
1175     if (error)
1176         goto out_unlock_free;
```

# Sys\_semimedop(3)



**//Llama a `try_atomic_semop` para realizar la tarea**

```
1178     error = try_atomic_semop (sma, sops, nsops, un, task_tgid_vnr(current));  
1179     if (error <= 0) {  
1180         if (alter && error == 0) //si se ha realizado con éxito y puede  
            existir modificación  
1181             update_queue (sma);  
1182             goto out_unlock_free;  
1183     }
```

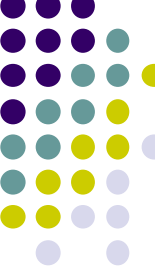
**//Si la tarea no se pudo realizar, el proceso actual debe dormirse. Se rellena la estructura `sem_queue`, se introduce en la cola y se llama al planificador**

```
1189     queue.sma = sma;  
1190     queue.sops = sops;  
1191     queue.nsops = nsops;  
1192     queue.undo = un;
```

# Sys\_semtimedop(4)

```
1193     queue.pid = task_tgid_vnr(current);  
1194     queue.id = semid;  
1195     queue.alter = alter;  
1196     if (alter)  
1197         append to queue(sma ,&queue);  
1198     else  
1199         prepend to queue(sma ,&queue);  
1201     queue.status = -EINTR;  
1202     queue.sleeper = current;  
1203     current->state = TASK_INTERRUPTIBLE;  
1204     sem_unlock(sma); //Se elimina el bloqueo  
1206     if (timeout)  
1207         jiffies_left = schedule_timeout(jiffies_left);  
1208     else  
1209         schedule();
```

# Sys\_semimedop(5)



//Una vez despertado se obtiene nuevamente el bloqueo

```
1223     sma = sem_lock(ns, semid);
```

...

//Estrae el proceos de la cola de procesos dormidos (FIFO)

```
1243     remove from queue(sma,&queue);
```

```
1244     goto out_unlock_free;
```

```
1245
```

```
1246out unlock free:
```

```
1247     sem_unlock(sma);//Desbloqueo
```

```
1248out free:
```

```
1249     if(sops != fast_sops)
```

```
1250         kfree(sops);
```

```
1251     return error;
```

```
1252}
```

# Try\_atomic\_semop



**Función que determina si una secuencia de operaciones del semáforo tendrán éxito o no ,intentando realizar cada una de las operaciones.**

```
369static int try_atomic_semop (struct sem_array * sma, struct sembuf * sops,  
370                               int nsops, struct sem_undo *un, int pid){
```

...

375 //Para cada una de las operaciones obtiene el semáforo en el que hay que realizar la tarea, la operacion a realizar.

```
376     for (sop = sops; sop < sops + nsops; sop++) {  
377         curr = sma->sem_base + sop->sem_num;  
378         sem_op = sop->sem_op;  
379         result = curr->semval;  
381         if (!sem_op && result) //no hay operacion pero hay semaforo  
382             goto would_block;  
383 //Se realiza la operacion en una variable auxiliar  
384         result += sem_op;
```



# Try\_atomic\_semop(1)



```
385         if (result < 0) //deberá bloquearse
386             goto would_block;
387         if (result > SEMVMX) // comprobamos el valor max del contador
           de semaforos
388             goto out_of_range;
/*Comprobamos si SEM_UNDO está posicionado para cada operación
   efectuada, para crear estructura y conservar el rastro de operaciones
   hechas*/
389         if (sop->sem_flg & SEM_UNDO)
390             int undo = un->semadj[sop->sem_num] - sem_op;
//Se actualiza el nuevo valor del semáforo
397         curr->semval = result;
398     }
```

# Try\_atomic\_semop(2)

```
400     sop--;
//asignar el identificador de cada proceso
401     while (sop >= sops) {
402         sma->sem base[sop->sem num].sempid = pid;
403         if (sop->sem flg & SEM UNDO)
404             un->semadj[sop->sem num] -= sop->sem op;
405         sop--;
406     }
408     sma->sem otime = get seconds();
409     return 0;
410 //si se ha producido algún fallo (fuera de rango)
411 out of range:
412     result = -ERANGE;
413     goto undo;
```

# Try\_atomic\_semop(3)



**//si el proceso no puede realizar la operación**

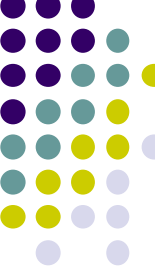
415would block:

```
416    if (sop->sem_flg & IPC_NOWAIT)  
417        result = -EAGAIN;  
418    else  
419        result = 1;
```

**//deshacemos las operaciones hechas**

421undo:

```
422    sop--;  
423    while (sop >= sops) {  
424        sma->sem_base[sop->sem_num].semval -= sop->sem_op;  
425        sop--;  
426    }  
428    return result;  
429}
```



# Ejemplo de uso de semáforos

**Como ejemplo expondremos la solución al problema de la Alarma usando semáforos:**

```
int semaforo;  
struct sembuf P,V;  
void crear_semaforo(){  
    key_t key=ftok("/bin/lis",1); //Crea la clave  
//Se crea un semáforo para controlar el acceso exclusivo al recurso compartido  
    semaforo = semget(key, 1, IPC_CREAT | 0666);  
//Se inicializa el semáforo a 1.  
    semctl(semaforo,0,SETVAL,1);
```



**//P decrementa el semaforo en 1 para cerrarlo y V lo incrementa para abrirlo.**

**//El flag SEM\_UNDO hace que si un proceso termina inesperadamente**

**//deshace las operaciones que ha realizado sobre el semaforo.**

P.sem\_num = 0;

P.sem\_op = -1;

P.sem\_flg = SEM\_UNDO;

V.sem\_num = 0;

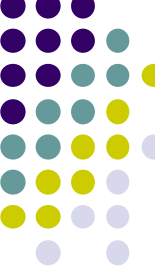
V.sem\_op = 1;

V.sem\_flg = SEM\_UNDO;

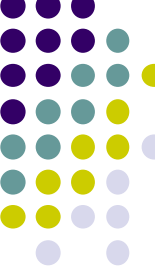
}



```
void PrintDate (int sig){
    pid_t pid;
    //Creamos un proceso hijo que sera el encargado de imprimir la fecha por pantalla
    pid = fork();
    // Comprobamos que ha sido posible crear el proceso hijo
    if (pid != -1){
        if (pid == 0){ //Si es el proceso hijo ejecuta el date con un exec
            // El proceso adquiere el semáforo para acceder al recurso
            // compartido (pantalla)antes de motrar la fecha.
            semop(semaforo,&P,1);
            execve("/bin/date",NULL,NULL); }
        else{//El proceso padre vuelve a poner la alarma a 4 segundos.
            alarm(4);} }
    else{
        cout<<"No se ha podido crear el proceso hijo"<<endl;
        exit(-1); }}}
```



```
int main (){
    signal (SIGALRM,PrintDate);
    //Se inicia la alarma a 4 segundos.
    alarm(4);
    crear_semaforo();
    while (true){
        sleep(1);
        //Adquirimos el semáforo con la operación P, accediendo al recurso compartido(pantalla).
        semop(semaforo,&P,1);
        cout<<"En un lugar de la mancha de cuyo nombre no quiero ni acordarme..."<<endl;
        //Una vez utilizado el recurso compartido, liberamos el semáforo con la operación V.
        semop(semaforo,&V,1);
    }
}
```



**FIN**