Libraries

COLLABORATORS			
	TITLE : Libraries		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 25 Exec Semaphores

Semaphores are a feature of Exec which provide a general method for tasks to arbitrate for the use of memory or other system resources they may be sharing. This chapter describes the structure of Exec semaphores and the various support functions provided for their use. Since the semaphore system uses Exec lists and signals, some familiarity with these concepts is helpful for understanding semaphores.

In any multitasking or multi-processing system there is a need to share data among independently executing tasks. If the data is static (that is, it never changes), then there is no problem. However, if the data is variable, then there must be some way for a task that is about to make a change to keep other tasks from looking at the data.

For example, to add a node to a linked list of data, a task would normally just add the node. However, if the list is shared with other tasks, this could be dangerous. Another task could be walking down the list while the change is being made and pick up an incorrect pointer. The problem is worse if two tasks attempt to add an item to the list at the same time. Exec semaphores provide a way to prevent such problems.

A semaphore is much like getting a key to a locked data item. When you have the key (semaphore), you can access the data item without worrying about other tasks causing problems. Any other tasks that try to obtain the semaphore will be put to sleep until the semaphore becomes available. When you have completed your work with the data, you return the semaphore.

For semaphores to work correctly, there are two restrictions that must be observed at all times:

1) All tasks using shared data that is protected by a semaphore must always ask for the semaphore first before accessing the data. If some task accesses the data directly without first going through the semaphore, the data may be corrupted. No task will have safe access to the data.

2) A deadlock will occur if a task that owns an exclusive semaphore on some data inadvertently calls another task which tries to get an exclusive semaphore on that same data in blocking mode. Deadlocks

and other such issues are beyond the scope of this manual. For more details on deadlocks and other problems of shared data in a multitasking system and the methods used to prevent them, refer to a textbook in computer science such as Operating Systems by Tannenbaum (Prentice-Hall).

Semaphore Functions Function Reference

1.2 25 Exec Semaphores / Semaphore Functions

Exec provides a variety of useful functions for setting, checking and freeing semaphores. The prototypes for these functions are as follows.

```
VOID  AddSemaphore ( struct SignalSemaphore *sigSem );
ULONG AttemptSemaphore( struct SignalSemaphore *sigSem );
struct SignalSemaphore *FindSemaphore( UBYTE *sigSem );
VOID InitSemaphore( struct SignalSemaphore *sigSem );

VOID ObtainSemaphore( struct SignalSemaphore *sigSem );
VOID ObtainSemaphoreList( struct List *sigSem );
void ObtainSemaphoreShared( struct SignalSemaphore *sigSem );

VOID ReleaseSemaphore( struct SignalSemaphore *sigSem );
VOID ReleaseSemaphoreList( struct List *sigSem );
VOID RemSemaphore( struct SignalSemaphore *sigSem );
```

The Signal Semaphore Multiple Semaphores Semaphore Example

1.3 25 / Semaphore Functions / The Signal Semaphore

Exec semaphores are signal based. Using signal semaphores is the easiest way to protect shared, single-access resources in the Amiga. Your task will sleep until the semaphore is available for use. The SignalSemaphore structure is as follows:

```
struct SignalSemaphore {
    struct  Node ss_Link;
    SHORT   ss_NestCount;
    struct  MinList ss_WaitQueue;
    struct  SemaphoreRequest ss_MultipleLink;
    struct  Task *ss_Owner;
    SHORT   ss_QueueCount;
};
```

ss_Link
: is the node structure used to link semaphores together. The ln_Pri and ln_Name fields are used to set the priority of the semaphore in a list and to name the semaphore for public access. If a semaphore is not public the ln_Name and ln_Pri fields may be left NULL.

ss_NestCount
: is the count of number of locks the current owner has on the

```
    semaphore.

ss_WaitQueue
    is the List header for the list of other tasks waiting for this
    semaphore.

ss_MultipleLink
    is the SemaphoreRequest used by ObtainSemaphoreList().

ss_Owner
    is the pointer to the current owning task.

ss_QueueCount
    is the number of other tasks waiting for the semaphore.

A practical application of a SignalSemaphore would be to use it as the
base of a shared data structure.  For example:

    struct SharedList {
        struct SignalSemaphore sl_Semaphore;
        struct MinList         sl_List;
    };

 Creating a SignalSemaphore Structure
 Making a SignalSemaphore Available to the Public
 Obtaining a SignalSemaphore Exclusively
 Obtaining a Shared SignalSemaphore
 Checking a SignalSemaphore
 Releasing a SignalSemaphore
 Removing a SignalSemaphore Structure
```

1.4 25 / / The Signal Semaphore / Creating a SignalSemaphore Structure

```
To initialize a SignalSemaphore structure use the InitSemaphore()
function.  This function initializes the list structure and the nesting
and queue counters.  It does not change the semaphore's name or priority
fields.

This fragment creates and initializes a semaphore for a data item such as
the SharedList structure above.

    struct SharedList *slist;

    if (slist=(struct SharedList *)
        AllocMem(sizeof(struct SharedList),MEMF_PUBLIC|MEMF_CLEAR))
    {
        NewList(&slist->sl_List);       /* Initialize the MinList       */
        InitSemaphore((struct SignalSemaphore *)slist);
                                        /* And initialize the semaphore */

        /* The semaphore can now be used. */
    }
    else printf("Can't allocate structure\n");
```

1.5 25 / / / Making a SignalSemaphore Available to the Public

A semaphore should be used internally in your program if it has more than one task operating on shared data structures. There may also be cases when you wish to make a data item public to other applications but still need to restrict its access via semaphores. In that case, you would give your semaphore a unique name and add it to the public SignalSemaphore list maintained by Exec. The AddSemaphore() function does this for you. This works in a manner similar to AddPort() for message ports.

To create and initialize a public semaphore for a data item and add it to the public semaphore list maintained by Exec, the following function should be used. (This will prevent the semaphore from being added or removed more than once by separate programs that use the semaphore).

```
UBYTE *name;   /* name of semaphore to add */
struct SignalSemaphore *semaphore;

Forbid();
/* Make sure the semaphore name is unique */
if (!FindSemaphore(name)) {
    /* Allocate memory for the structure */
    if (sema=(struct SignalSemaphore *)
        AllocMem(sizeof(struct SignalSemaphore),MEMF_PUBLIC|MEMF_CLEAR))
    {
        sema->ss_Link.ln_Pri=0;         /* Set the priority to zero */
        sema->ss_Link.ln_Name=name;
        /* Note that the string 'name' is not copied. If that is    */
        /* needed, allocate memory for it and copy the string. And  */
        /* add the semaphore the the system list                    */
        AddSemaphore(semaphore);
    }
}
Permit();
```

A value of NULL for semaphore means that the semaphore already exists or that there was not enough free memory to create it.

Before using the data item or other resource which is protected by a semaphore, you must first obtain the semaphore. Depending on your needs, you can get either exclusive or shared access to the semaphore.

1.6 25 / / The Signal Semaphore / Obtaining a SignalSemaphore Exclusively

The ObtainSemaphore() function can be used to get an exclusive lock on a semaphore. If another task currently has an exclusive or shared lock(s) on the semaphore, your task will be put to sleep until all locks on the the semaphore are released.

```
Semaphore Nesting.
------------------
SignalSemaphores have nesting.  That is, if your task already
owns the semaphore, it will get a second ownership of that semaphore.
This simplifies the writing of routines that must own the semaphore
```

```
    but do not know if the caller has obtained it yet.

To obtain a semaphore use:

    struct SignalSemaphore *semaphore;
    ObtainSemaphore(semaphore);

To get an exclusive lock on a public semaphore, the following code should
be used:

    UBYTE *name;
    struct SignalSemaphore *semaphore;

    Forbid();     /* Make sure the semaphore will not go away if found. */
    if (semaphore=FindSemaphore(name))
        ObtainSemaphore(semaphore);
    Permit();

The value of semaphore is NULL if the semaphore does not exist.  This is
only needed if the semaphore has a chance of going away at any time (i.e.,
the semaphore is public and  might be removed by some other program).  If
there is a guarantee that the semaphore will not disappear, the semaphore
address could be cached, and all that would be needed is a call to the
ObtainSemaphore() function.
```

1.7 25 / / The Signal Semaphore / Obtaining a Shared SignalSemaphore

```
For read-only purposes, multiple tasks may have a shared lock on a signal
semaphore.  If a semaphore is already exclusively locked, all attempts to
obtain the semaphore shared will be blocked until the exclusive lock is
released.  At that point, all shared locks will be obtained and the
calling tasks will wake up.

To obtain a shared semaphore, use:

    struct SignalSemaphore *semaphore;
    ObtainSemaphoreShared(semaphore);

To obtain a public shared semaphore, the following code should be used:

    UBYTE *name;
    struct SignalSemaphore *semaphore;

    Forbid();
    if (semaphore = FindSemaphore(name))
        ObtainSemaphoreShared(semaphore);
    Permit();
```

1.8 25 / / The Signal Semaphore / Checking a SignalSemaphore

```
When you attempt to obtain a semaphore with ObtainSemaphore(), your task
will be put to sleep if the semaphore is not currently available. If you
```

do not want to wait, you can call AttemptSemaphore() instead. If the semaphore is available for exclusive locking, AttemptSemaphore() obtains it for you and returns TRUE. If it is not available, the function returns FALSE immediately instead of waiting for the semaphore to be released.

To attempt to obtain a semaphore, use the following:

```
struct SignalSemaphore *semaphore;
AttemptSemaphore(semaphore);
```

To make an attempt to obtain a public semaphore, the following code should be used:

```
UBYTE *name;
struct SignalSemaphore *semaphore;

Forbid();
if (semaphore = FindSemaphore(name)) AttemptSemaphore(semaphore);
Permit();
```

1.9 25 / / The Signal Semaphore / Releasing a SignalSemaphore

Once you have obtained the semaphore and completed any operations on the semaphore protected object, you should release the semaphore. The ReleaseSemaphore() function does this. For each successful ObtainSemaphore(), ObtainSemaphoreShared() and AttemptSemaphore() call you make, you must have a matching ReleaseSemaphore() call.

1.10 25 / / The Signal Semaphore / Removing a SignalSemaphore Structure

Semaphore resources can only be freed if the semaphore is not locked. A public semaphore should first be removed from the system semaphore list with the RemSemaphore() function. This prevents other tasks from finding the semaphore and trying to lock it. Once the semaphore is removed from the system list, the semaphore should be locked exclusively so no other task can lock it. Once the lock is obtained, it can be released again, and the resources can be deallocated.

The following code should be used to remove a public semaphore:

```
UBYTE *name;
struct SignalSemaphore *semaphore;

Forbid();
if (semaphore=FindSemaphore(name))
{
    RemSemaphore(semaphore);       /* So no one else can find it... */
    ObtainSemaphore(semaphore);    /* Wait for us to be last user...*/
    ReleaseSemaphore(semaphore);   /* Ready for cleanup...          */
}
FreeMem(semaphore, sizeof(struct SignalSemaphore));
Permit();
```

1.11 25 / Semaphore Functions / Multiple Semaphores

The semaphore system has the ability to ask for ownership of a complete list of semaphores. This can help prevent deadlocks when there are two or more tasks trying to get the same set of semaphores. If task A gets semaphore 1 and tries to obtain semaphore 2 after task B has obtained semaphore 2 but before task B tries to obtain semaphore 1 then both tasks will hang. Exec provides ObtainSemaphoreList() and ReleaseSemaphoreList() to prevent this problem.

A semaphore list is a list header to a list that contains SignalSemaphore structures. The semaphore list must not contain any public semaphores. This is because the semaphore list functions use the standard node structures in the semaphore.

To arbitrate access to a semaphore list use another semaphore. Create a public semaphore and use it to arbitrate access to the list header of the semaphore list. This also gives you a locking semaphore, protecting the ObtainSemaphoreList() call. Once you have gained access to the list with ObtainSemaphore(), you may obtain all the semaphores on the list via ObtainSemaphoreList() (or get individual semaphores with ObtainSemaphore()). When you are finished with the protected objects, release the semaphores on the list with ReleaseSemaphoreList(), and then release the list semaphore via ReleaseSemaphore().

For example:

```
    ObtainSemaphore((struct SignalSemaphore *)SemaphoreList);
    ObtainSemaphoreList(SemaphoreList->sl_List);

    /* At this point the objects are protected, and can be manipulated */

    ReleaseSemaphoreList(SemaphoreList->sl_List);
    ReleaseSemaphore((struct SignalSemaphore *)SemaphoreList);
```

See the SharedList structure above for an example of a semaphore structure with a list header.

1.12 25 Exec Semaphores / Function Reference

The following charts give a brief description of the Exec semaphore functions. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for details about each call.

Table 25-1: Exec Semaphore Functions

Exec Semaphore Function	Description
AddSemaphore()	Initialize and add a signal semaphore to the system.
AttemptSemaphore()	Try to get an exclusive lock on a signal semaphore without blocking.

FindSemaphore()	Find a given system signal semaphore.
InitSemaphore()	Initialize a signal semaphore.
ObtainSemaphore()	Try to get exclusive access to a signal semaphore.
ObtainSemaphoreList()	Try to get exclusive access to a list of signal semaphores.
ObtainSemaphoreShared()	Try to get shared access to a signal semaphore (V36).
ReleaseSemaphore()	Release the lock on a signal semaphore.
ReleaseSemaphoreList()	Release the locks on a list of signal semaphores.
RemSemaphore()	Remove a signal semaphore from the system.