1070

Bell Laboratories Cover Sheet for Technical Memorandum

Title- **Proposal for UNIX Interprocess Communication**

Date- **March 17, 1976**

TM- **76-8234-4**

Other Keywords-

Author	Location	Extension	Charging Case- 49170-210
R. B. Brandt	**MH 2D-428**	**3669**	Filing Case- **40952-001**

ABSTRACT

As the community of UNIX users has grown, so has the need for new or improved operating system services and features. One such requirement has been the need for improved techniques for the exchange of data and timing signals between processes. This memorandum provides the functional design specifications for a comprehensive set of interprocess communication enhancements to the UNIX operating system.

These features include:

extension to semaphores to allow a choice between the lock-unlock type and the P-V counting type.

system symbol table to associate a name with a value.

messages to allow the exchange of small amounts of data between processes.

events to facilitate the passing of user defined timing signals.

shared memory to permit cooperating processes to share physical memory.

Pages Text 18	Other 0	Total 18	
No. Figures 0	No. Tables 0	No. Refs. 0	

Bell Laboratories

Subject: **Proposal for UNIX Interprocess Communication**
Case- 49170-210 -- File- 40952-001

date: **March 17, 1976**

from: **R. B. Brandt**

TM: **76-8234-4**

MEMORANDUM FOR FILE

Introduction

As the number of UNIX users has grown in recent years, so also has the need for new or improved operating system services and features. Not the least of these new requirements has been the need for different techniques for the exchange of timing signals and data between processes *(interprocess communication)*. In fact, many users have locally modified their systems already in order to provide additional capabilities in this area. In response to this need, a comprehensive set of interprocess communication facilities has been designed for incorporation in the UNIX Support Group's standard version of the UNIX operating system. These enhancements include an extension to the semaphore facility that is already available and the addition of a system symbol table, messages, events, and shared memory. The purpose of this memorandum is to provide the functional design specifications for these new features.

Why Interprocess Communication?

Before proceeding to describe the new interprocess communication features, a brief explanation of the motivation for their implementation is in order. As is well known, UNIX is a general purpose, interactive operating system. In the time-sharing environment, most user processes tend to be relatively short-lived; a process is created, in response to a user's command, to accomplish some specific short-term task (e.g. list a file or make a directory). Once initiated, the process usually has little, if any, need to communicate with other processes and it may be destroyed at the completion of its work with no ill effects. If the task to be accomplished is sufficiently large or complex, a series of processes may be required to complete it. This can usually be accommodated by not spawning any process of the series until the preceding one has completed its work and been terminated (e.g. compile and link a program). In these situations, intermediate results are usually passed on to succeeding processes by writing them to some temporary file. In such a *task-oriented* environment, there is little need for any of the more exotic forms of communication between processes.

However, there are instances where a process is acting as a monitor, continuously waiting for some event(s) to occur and performing work only when stimulated by the occurrence of the event(s). Such processes tend to remain in the system for long periods of time and typically require little, if any, user guidance once they have been initiated. Examples of such *transaction-oriented* processes in UNIX are *init*, *update*, and the various *daemons*. The technique of having a monitoring process run continuously is more expeditious than spawning a new process each time the awaited event or transaction occurs. A number of the applications that are being developed using UNIX as the base operating system have what might be called networks of processes of this nature. For example, an overseer process, in a manner somewhat analogous to the *shell*'s handling of commands, may receive and decode transactions and pass them on to any one of a number of other processes. Which process is chosen as recipient is determined by

what further action is required to complete the transaction. This receiving process, in turn, may be capable of only partially processing the transaction, so it may have to be passed on to yet another process in the network. In such situations, the currently available techniques for interprocess communication have not proven to be sufficient for various reasons.

At present there are four principal facilities available and commonly used in UNIX for interprocess communication: *arguments to exec, pipes, signals,* and *semaphores.* The passing of data to another process via arguments in an exec system call is restrictive because of the limited amount of data that may be passed (maximum of 512 bytes) and because it is essentially a one shot proposition. This does not satisfy the requirement of many users that processes be able to converse with one another.

Pipes, on the other hand, are a very convenient method of linking together a series of commands (i.e. processes). However, since they are one way and use the file system as the medium of communication, pipes have not been wholly satisfactory as a general solution of the data exchange problem. The most commonly voiced objections to pipes relate primarily to their speed and their impracticality for networks of processes.

Often the data that is to be exchanged is actually nothing more than notification that an event has occurred (a timing signal). The UNIX signal facility was designed for this type of data exchange. However, signals are used primarily by the operating system to advise processes of exceptional conditions that require attention (e.g. quit, illegal instruction, floating point exception). The default action taken by the system on behalf of a process that has received a signal is to terminate the process. Partly because of this underlying philosophy about signal handling and partly to avoid, or, at least, to minimize, vulnerability to future operating system modifications, users have been understandably reluctant to adopt signals as a general technique for timing signal exchange.

Semaphores have only recently been made available in UNIX and have proven quite useful to those applications that require a locking and unlocking capability for critical resources. In the absence of any other general facility, some applications also use semaphores for the exchange of timing signals. Although semaphores are useful for solving some timing problems, they are not terribly well suited for many of the data exchange problems with which applications are confronted.

In summation, UNIX users may be loosely categorized as either time-sharing oriented or support system oriented. The former group has been reasonably content with the interprocess communication facilities already available in UNIX. The latter has, however, because of the nature of their application, been much more concerned with the problems of process synchronization and data exchange.

Overview of New Features

As previously stated, the interprocess communication features designed consist of a system symbol table, messages, events, shared memory, and an extension to semaphores. In addition, a process will be able to determine more readily the system's process identification (the *processid)* of its parent. Lest the reader suddenly become engulfed in a sea of design details, a brief overview of each of these new features will be given.

Semaphores. One version of semaphores has already been incorporated in the standard UNIX system. In this implementation, the semaphore merely acts as a simple lock, as it may be in only one of two states: locked or unlocked. This is useful for ensuring that critical resources are used by processes in a single thread fashion. For some synchronization problems, such as the reader-writer problem, the counting version of semaphores (P-V) is more suitable. Here the semaphore acts as a counter for a resource, indicating how many are available (e.g. filled input buffers). Therefore, the semaphore facility will be extended to include this function. The user may choose the way the semaphore is to be manipulated by the system.

System Symbol Table. Almost all forms of interprocess communication rely on processes knowing to whom they wish to communicate. Processes are not known to UNIX by name; rather, they are assigned a unique processid by the system when they are created. Since the user has virtually no control over assignment of processids, it may be difficult for a process to determine the identification of another process. This is especially true if the processes are not directly related (parent-child). The system symbol table enables a process to post its logical name (e.g. "xyz") and processid on a system bulletin board for other processes to read. Further, since there is no restriction that the symbol's value be a processid, a process is free to associate a symbol with any value that may be meaningful to other interested processes.

Parent Process Identification. When a process spawns another via the *fork* system call, it is advised of the processid of the newly created process. The addition of the parent processid feature completes the parent-child identification circle, as it enables a process to easily determine its parent's identification.

Messages. The message facility permits a process to transmit small (say, up to 100 words) amounts of data to another existing process. Messages that are sent to a process are placed in its message queue in order of arrival. The sending process may then continue to execute, if it so desires, without waiting for the recipient to actually receive the message. The receiving process must explicitly request a message before it will actually receive one. It is then given the first message in its message queue.

Events. The event facility was designed to facilitate the exchange of timing signals (process synchronization). Processes are free to define their own meaning to eight event flags. An additional eight event flags are reserved for use by the operating system. A receiving process provides a mask to indicate those events in which it is interested, and it may either wait for one of those events to occur or may continue to execute after stipulating a function that is to be given control when one occurs.

Shared Memory. This facility enables a process to share a portion of its physical address space with one or more other processes. Thus, processes using this feature will be able to transfer data to one another very rapidly. The operating system's only responsibility in providing this service is to set up the process's segmentation registers in a way that makes the overlap possible. The processes sharing memory must work out their own synchronization protocol; this may entail use of one of the other interprocess communication facilities.

Philosophy and Constraints

As with other endeavors of this type, a certain underlying philosophy guided many of the design decisions. In addition, there are considerations and pressures from user projects that had to be recognized and taken into account. An explanation of some of the more prominent constraints and philosophical points will help to illustrate why some things were done the way they were.

Each of the interprocess communication facilities is viewed as, if you will, a dynamically allocatable resource. That is, a process must declare its intent to use a particular feature before it may avail itself of the service. This is not a foreign concept to UNIX; a process must, for example, open a file before it may read it. There are, of course, resources in UNIX that are provided to a process without it taking any explicit action (e.g. use of standard file descriptors for terminal I/O), but such resources are usually those that are essential for the proper execution of most processes. For those features that have both a sending process and a receiving process, the burden of declaration is placed on the receiver. This affords a process at least some control over its own destiny, as it prevents a talkative process from disrupting the solitude of one that would prefer to remain deaf.

As with other resources, one must consider the question of protection. Processes that have successfully established a line of communication to one another are, in theory at least, cooperating with one another. It is virtually impossible for the operating system to even attempt to determine the significance of their communiques and enforce any restrictions on content. The system can only ensure that communications are carried out in an orderly fashion according to the prescribed rules. What the system can do to afford some degree of protection, however, is to allow a process to designate the group of processes with which it is willing to communicate. This is especially necessary in the time-sharing environment, where the user should be protected from the whims of another user's process that is either malicious or insane. However, even for applications that are less time-sharing oriented, where the likelihood of a malicious process is greatly diminished, it is prudent to limit a process's sphere of influence as a hedge against the possibility that it may go beserk. Therefore, at the time a process declares its intent to receive communications via a certain facility, it must stipulate the class of processes that may communicate with it using that facility. At present there are only three such groupings: processes with the same userid, processes with the same groupid, or any process (i.e. no protection desired).

User impact has influenced these designs, especially with respect to semaphores. Different users have stated different requirements for interprocess communication facilities, and, in general, the intersection of these requirements was used as a point of departure in the design work. Not only these current, but also anticipated, requirements were taken into account as work progressed. An effort was exerted to make the resulting facilities upward compatible with features already implemented locally. When these features are actually implemented, a similar effort will be made to minimize the impact on existing operating system code. Thus, users not requiring a particular feature should be able to remove it with a minimal amount of effort. Although such practices are not encouraged, one must understand that for some systems, especially those on 11/40's, the operating system's memory requirement is a limiting factor.

Functional Descriptions

Each of the interprocess communication facilities will be described separately, and each description will be broken into a synopsis of operation, a description of the assembler interface, a description of the C interface, and rules governing the facility's use. The purpose of this document is to present only the functional specifications for these new features, so scrupulous care will be taken to avoid discussion of implementation details.

1. Semaphores

1.1. Synopsis

There are two flavors of semaphores: the lock-unlock version and the P-V (counting) version. Each semaphore, regardless of its type, is identified by a unique semaphore number. However, unlike open file descriptors, this number is unvarying across processes (i.e. known system wide). Each process wishing to use a particular semaphore must explicitly allocate it before it may be used. The type (lock-unlock or P-V) and scope (the class of process that may also allocate and use the semaphore) of a semaphore are determined on its initial allocation by a process. These definitions remain in effect as long as the semaphore is allocated to any process. Any subsequent process attempting to allocate the semaphore must be in the proper process class and must specify the same usage before it is granted permission to use the semaphore. If a process does not specify a semaphore number in its allocation request (i.e. a non-specific allocation), the system will choose an unallocated semaphore from its pool, assign the requested usage and scope, and advise the process of the semaphore number. Other processes would then allocate the semaphore by specifying this number (i.e. specific requests). This is the technique that would typically be used by users in a time-sharing environment, as it eliminates the necessity of hard-coding semaphore numbers in programs. The initial allocation of a semaphore may

also be accomplished by making a specific request. Since there is the chance that the same semaphore number may have been already allocated by another user, this type of initial allocation should only be done in a controlled environment where semaphore numbers are preassigned. When a semaphore is no longer needed by a process, it may be freed (deallocated). All of a process's semaphores are automatically freed when the process terminates. A semaphore that is no longer allocated to any process is returned to the system's semaphore pool for future allocation.

A lock-unlock semaphore is useful for such things as enforcing serial usage of a critical resource. A process wishing to utilize the resource locks the semaphore to indicate that it is in use. This locked state is indicated by the processid of the locking process. Other processes attempting to set the lock are put to sleep (roadblocked) until the original process relinquishes the resource by unlocking the semaphore. One of the waiting processes is then awakened and the semaphore is locked in its behalf. Besides the basic lock and unlock actions on the semaphore, there is also a conditional lock feature. If the semaphore is in the unlocked state, then it is locked for the process. However, if the semaphore is already locked, the requesting process is not roadblocked and the system returns control to the process with no further action. This version of semaphores is currently available.

The producer-consumer problem is one application of the P-V counting semaphores. Here the semaphore acts as a counter, which is incremented by one by the V operation and decremented by one by the P operation. The counter is never allowed to become negative, however. A process attempting a P operation when the semaphore is zero is roadblocked until another process does a V operation, which not only increments the counter, but also awakens any processes waiting for it to become positive. A conditional P operation is also available, analogous to the conditional lock available with lock-unlock semaphores. The semaphore is decremented if it was positive, but if it was zero, it is not decremented; in either case, the process is not roadblocked.

1.2. Assembler Interface

Call.

```
lock = 62.; not in assembler
sys lock; function; flag
```

The low byte of the argument *function* specifies the request type. The permissible values for this argument are as follows.

0	The semaphore is locked in behalf of the calling process. If the semaphore is already locked, the process is put to sleep until it is unlocked.
1	The semaphore is unlocked and any process that is waiting for it to be unlocked is awakened.
2	The semaphore is locked if it is not already so. The process is not roadblocked if the semaphore is already locked.
3	The semaphore's value is decremented by one if it is positive. However, if the value is zero, the process is roadblocked until it becomes positive, at which time the process is awakened and the semaphore is decremented. (P operation)
4	The semaphore's value is incremented by one. Any processes that are roadblocked because of a zero semaphore value are awakened. (V operation)

5 The semaphore's value is decremented by one, but never below zero. The process is not roadblocked if the semaphore was zero. (Test operation)

6 A semaphore is allocated to the process. The high byte of *function* indicates the semaphore's usage and scope. A value of 0, 1, or 2 allocates a lock-unlock semaphore with a scope of anyone, same userid only, or same groupid only, respectively. Similarly, a value of 3, 4, or 5 allocates a P-V semaphore with a scope of anyone, same userid only, or same groupid only, respectively.

7 The semaphore is freed.

The *flag* argument designates the semaphore number of the semaphore in question. In semaphore allocation requests a negative value for *flag* implies that an available semaphore from the system's pool is to be chosen and allocated. A negative value for *flag* in deallocation requests indicates that all of the semaphores allocated to the process are to be freed.

Returns. If an error occurs, regardless of the request type, the error bit (c-bit) is set and r0 contains the appropriate error number. The contents of r0 after successful calls depends on the action requested: the semaphore's previous value for lock, unlock, conditional lock, P, V, and conditional P operations; the number of the allocated semaphore for allocation requests; and zero for deallocation requests.

1.3. C interface

Calls.

```
allocsem(number, value)
int number, value;
```

The *allocsem* function allocates semaphore *number* to the invoking process. If *number* is -1, then an available semaphore is selected from the system's pool. The argument *value* is used to specify the semaphore's usage and scope. If *value* is 0, 1, or 2, then the semaphore is of lock-unlock type and is assigned a scope of any process (i.e. no restriction), same userid only, or same groupid only, respectively. Likewise, if this argument is 3, 4, or 5, the semaphore is of P-V type and is given a scope of any process, same userid only, or same groupid only, respectively.

```
freesem(number)
int number;
```

The deallocation of a semaphore is accomplished through the *freesem* function. The semaphore to be freed is designated by the value of *number*. If *number* is -1, then all of the semaphores allocated to the process are freed.

```
lock(number)
unlock(number)
tlock(number)
int number;
```

The *lock*, *unlock*, and *tlock* functions operate only on a semaphore of the lock-unlock type. As the names imply, these functions lock, unlock, and conditionally lock, respectively, the semaphore designated by *number*. The semaphore must have been previously allocated by the process (see the special note below regarding allocation).

```
p(number)
v(number)
test(number)
int number;
```

The *p*, *v*, and *test* functions may be used only in conjunction with semaphores of P-V type. They perform the P operation, V operation, and conditional P operation, respectively, on the semaphore designated by *number*. The semaphore in question must have already been allocated to the process (see the special note below regarding allocation).

Special Note. In order to effect upward compatibility for applications already using semaphores and yet still provide for dynamic allocation of semaphores for time-sharing use, a special capability will be included in the C library functions. In particular, if a *lock*, *unlock*, *tlock*, *p*, *v*, or *test* function is invoked without the semaphore having previously been allocated by the process, then the function will first attempt to allocate the semaphore before performing the requested operation. The semaphore will be assigned the appropriate usage and a scope of any process.

Returns. All of the C library semaphore functions will return a -1 if an error occurs, and the external variable *errno* will contain the appropriate error number. For successful calls, the *lock*, *unlock*, *tlock*, *p*, *v*, and *test* functions return the previous value of the semaphore. The *allocsem* function returns the number of the allocated semaphore when successful, while *freesem* always returns zero on success.

1.4. Rules of the Road

A semaphore operation *(lock, unlock, tlock, p, v,* or *test)* is permitted only on a previously allocated semaphore of the same type (except as indicated in the special note above). Thus, lock-unlock is not allowed on a P-V type semaphore, and vice versa.

The use and scope of a semaphore are established on the first allocation of it by a process. Subsequent processes attempting to allocate the semaphore must conform to these initial conditions, which remain in effect as long as the semaphore is allocated to any process. A semaphore returns to the free semaphore pool when it has been freed by all the processes that allocated it.

The effective userid or groupid is used to determine access permission to a semaphore. Superuser may, of course, allocate any semaphore, as long as the declared usage remains consistent.

A *freesem(-1)* is implicit upon termination of a process. A lock-unlock semaphore being freed by the process that has it locked is first unlocked.

Allocated semaphores are inherited across *fork*, but not across *exec*.

A process roadblocked on a semaphore will sleep at a positive software priority, so that signals (and events) may still be captured.

The number of semaphores that any one process may allocate will be limited to some reasonable value (say, four or five).

2. System Symbol Table

2.1. Synopsis

The system symbol table permits a process to associate a word (16 bits) of data with a name (symbol). Other processes then inquiring about the name are advised of its value. A typical use of this facility is to connect a process's name with its processid, so that other processes may learn its system identity in order to make use of other forms of interprocess communication. At the time a process enters a name in the symbol table (posts the name), it must designate the scope of the name. This scope not only delineates the class of processes that may inquire about the symbol (the name's access permission), but also serves as a modifier to the name. For ex-

ample, a process might post the name "xyz" with a userid level scope. Later, another user's process may post the same name, also with a userid level scope. This results in two distinct entries in the symbol table for the name "xyz". Another process belonging to the first user might then request information about "xyz" at the userid level. There is no ambiguity as to which "xyz" is being referenced. The assignment of a scope to a posted symbol is essential in a time-sharing environment in order to protect users that may happen to choose the same symbol.

A symbol may be removed from the symbol table at any time, but only by the process that first posted it. Once a symbol has been entered in the table, its value may be altered by simply reentering it with the new value. Since the symbol is perceived as being defined across the class of processes designated by its scope, any process in that class has access permission and is free to alter its value.

2.2. Assembler Interface

Call.

```
symbol = 33.; not in assembler
sys symbol; function; symptr
```

The low byte of the argument *function* specifies what type of operation is desired. There are only three permissible values for this argument.

- 0 The symbol is entered in the symbol table and assigned the value specified in r0. If the symbol (with the same scope) already exists in the table, then its value is changed to that given in r0.
- 1 The symbol is removed from the symbol table.
- 2 The value of the symbol is returned to the caller.

For each of these three operations, the scope of the symbol is indicated by the high byte of *function*. As explained, this eliminates any ambiguity about identification of like names. The allowable scopes are 0 for system-wide (all processes), 1 for same userid only, and 2 for same groupid only.

The argument *symptr* is a pointer to the symbol in question. The symbol itself is a null terminated string of not more than eight characters.

Returns. If any request is unsuccessful, then the error bit (c-bit) is set and r0 contains the appropriate error number. After a successful call, r0 is zero when a symbol is being entered or removed from the table. If the request is to determine a symbol's value, that value is returned in r0.

2.3. C Interface

Calls. In each C library function, a scope argument of 0 implies system-wide, 1 implies across same userid only, and 2 implies across same groupid only.

```
setname(strptr, value, scope)
char *strptr;
int value, scope;
```

The *setname* function causes the null terminated string addressed by *strptr* to be entered in the symbol table with the value *value*. The symbol's length may not exceed eight characters. The scope of this symbol is determined by *scope*. If the symbol (with the same scope) already exists, then its value is altered to *value*.

```
rmname(strptr, scope)
char *strptr;
int scope;
```

Removal of the symbol addressed by *strptr* is accomplished with the *rmname* function. The symbol's scope is determined by the *scope* argument.

```
nameval(strptr, &value, scope)
char *strptr;
int value, scope;
```

The value of the symbol addressed by *strptr* is determined and placed in the word *value* by the *nameval* function. As before, the argument *scope* specifies the scope of the requested symbol.

Returns. For all of these functions, a return of -1 indicates an error; the error number is placed in the external variable *errno*. All of the functions return a zero on success.

2.4. Rules of the Road

Only the process that first posts a symbol may remove it from the symbol table. However, any process with appropriate access permission may alter its value.

When a process terminates, all of the symbols that it actually entered in the symbol table are removed.

There is a limit on the number of symbols that any one process may enter in the table.

The actual ownership of a symbol is not inherited across *fork* or *exec*.

Only the super-user may give a symbol system wide scope.

The effective userid and groupid will be used to determine access permission and scope.

3. Parent Processid

3.1. Synopsis

When a new process is created with *fork*, the parent process is advised of the child's processid. The addition of a facility to obtain the parent processid completes the circle, as processes have a convenient way to determine their parent's identity. It should be pointed out that the *init* process, which is always processid 1, adopts all orphaned children in the system. Thus, a process can determine its true parent's processid only if the parent is still living.

3.2. Assembler Interface

Call.

```
getppid = 26.; not in assembler
sys getppid
```

Returns. The processid of the parent process is returned in r0. This call can "never" fail, although in a sick system the processid returned may not be accurate.

3.3. C Interface

Call.

```
getppid()
```

Return. The processid of the parent processid is returned. This value may not be accurate if the system is sick.

3.4. Rules of the Road

There are no special considerations regarding this call except that user processes should be cognizant of the fact that they are adopted by *init* when their parent dies.

4. Messages

4.1. Synopsis

The message facility enables a process to pass small amounts of data (say, less than a hundred words) to another process. Messages exceeding the maximum allowable length must be sent incrementally. In order to send a message, the sender must know the processid of the recipient. The message may be sent only if the receiving process has explicitly expressed a willingness to accept messages. The implication is, of course, that a message may not be sent to a non-existent process. When a process enables message reception for itself, it stipulates the class of processes that may send it messages. Any process within that group is then free to send it a message, which is placed at the end of its message queue. The sending process is then at liberty to continue execution, if it so wishes.

Messages are actually passed to the receiving process only at its request. When it does ask for a message, the first one in its message queue is removed and placed in the designated buffer. If there are no messages in the queue, the process is roadblocked until one is sent. Alternately, there is a conditional receive message operation. In this case, the receiver is given the first message in the message queue if the queue is not empty, but is not roadblocked if it is empty. Whether or not message reception is done conditionally, the process is allowed to indicate that it wants a message only if it was sent by some particular process. A receiving process may disable message reception at any time. When this is done, no further messages may be sent to it, and any unreceived messages on its queue are discarded.

Notice that at least three distinct actions are required before a message may be transferred from a buffer in process A to one in process B. First, process B must declare that it is willing to receive messages. Until this has been done, process A is not permitted to send it a message. However, once this is done, process A may send a message, which is placed on process B's message queue. There is no guarantee that process B will ever learn the contents. The final step required is for process B to request a message, which causes it to be placed in B's buffer.

4.2. Assembler Interface

Call.

```
msg = 39.; not in assembler
sys msg; function; buffer; size; pid
```

The low byte of the argument *function* is used to indicate the request type. Permissible values are as follows.

0 The reception of messages is enabled for the calling process. Until this is done, no messages may be sent to the process. The high byte of *function* specifies the class of processes that may send messages to the caller: 0 for anyone (no restriction), 1 for same userid only, and 2 for same groupid only.

1 Message reception is disabled. Any unreceived messages still on the message queue are flushed.

2 A message is sent to the designated process.

3 The first message in the queue (or, the first message from a designated process) is placed in the caller's buffer. If the queue is empty (or, devoid of messages from the indicated process), the calling process's execution is suspended until a suitable message is placed on the queue.

4 A message is removed from the queue and passed to the calling process as described above. However, if no suitable message is in the queue, the process is not roadblocked.

The argument *buffer* is a pointer to the user's message buffer from whence the message will be taken or placed, depending on whether the process is sending or receiving a message.

For a sending process, the argument *size* specifies the length of the message. If this exceeds the system's maximum allowable length, the message is not sent. For a receiving process, *size* specifies the length of the message buffer addressed by *buffer*. If *size* is less than the message's actual length, the message is truncated to *size*.

When a message is being sent, the argument *pid* must contain the processid of the destination process. However, when a process is receiving a message, *pid* should be the address of a word. If the contents of that word are -1, then the first message in the queue will be given to the process, regardless of the sender's identity. Any value other than -1 is taken to be a processid; the first message from that process is retrieved. In either case, if a message is successfully placed in the user's buffer, this word is filled in with the processid of the sender.

The arguments *buffer*, *size*, and *pid* are not used in requests to enable or disable message reception.

Returns. For unsuccessful calls, the error bit (c-bit) is set and r0 contains the appropriate error number. For successful message enable and disable requests, r0 is always zero. When a message is sent, received, or conditionally received, r0 is used to indicate the number of bytes actually sent or received. In addition, when a message is received, whether conditionally or not, the word addressed by *pid* contains the processid of the sending process.

4.3. C Interface

Calls.

```
msgenab(permission)
int permission;
```

The *msgenab* function enables message reception for a process. No messages may be sent to it before this is done. The argument *permission* delineates the class of processes that will be allowed to send messages to this process: 0 for anyone (no restriction), 1 for same userid only, and 2 for same groupid only.

```
msgdisab()
```

The *msgdisab* function disables message reception and clears the message queue of any unreceived messages.

```
msgsend(buffer, size, topid)
char *buffer;
int size, topid;
```

The transmission of a message to an existing process that has enabled message reception is accomplished with the *msgsend* function. A message of length *size* is taken from *buffer* and placed on the message queue of process *topid*.

```
msgrecv(buffer, maxsize, &frompid)
char *buffer;
int maxsize, frompid;
```

A process requests a message from its queue with the *msgrecv* function. If *frompid* is -1, a message from any sender is desired; otherwise, this argument indicates the processid of the only acceptable sender. The first message in the queue (or, the first message from the indicated process) is placed in *buffer*. *Maxsize* specifies the length of this message buffer; longer messages are truncated. If there is not a suitable message in the queue, the process is roadblocked until one is sent. In any case, when a message is finally returned to the calling process, *frompid* contains the processid of the sender.

```
msgtest(buffer, maxsize, &frompid)
char *buffer;
int maxsize, frompid;
```

The *msgtest* function performs like the *msgrecv* function, except that the process's execution is not suspended if a satisfactory message is not already on the queue.

Special Note. The question of whether or not to include a timing facility in the *msgrecv* function has been subject to a certain amount of controversy. The recent addition of *alarm* and *pause* to the UNIX programmer's repertoire has made asynchronous timing much more convenient than in the past. Therefore, users wishing to limit the sleep time that is possible in *msgrecv* should utilize the *alarm-pause* facility.

Returns. All of these C functions return a -1 on error, with the external variable *errno* containing the error number. The *msgenab* and *msgdisab* functions return zero when successful, while the *msgsend, msgrecv,* and *msgtest* functions return the number of bytes sent or received, as appropriate. In addition, the argument *frompid* in the *msgrecv* and *msgtest* functions contains the processid of the message's sender.

4.4. Rules of the Road

A process may not send a message to a non-existent process, to a process that has not enabled message reception, or to a process that has enabled messages, but has not included the sender in the class of processes eligible to send it messages.

Super-user may send messages to any process that has enabled message reception.

The effective userid and groupid will be used to determine if a process is eligible to send a message to another.

It is erroneous for a receiving process to wait for a message *(msgrecv)* from a non-existent or zombie process.

Message disabling *(msgdisab)* is implicit when a process terminates. Any unreceived messages on the message queue are flushed when a process disables message reception.

When a message being placed in a receiver's buffer exceeds the buffer size *(maxsize)*, it will be truncated. A special case of this is that a process may request a zero length message, which causes the first message in the queue to be flushed.

There is a system imposed limitation on the maximum allowable message length and the maximum number of unreceived messages that may reside in a message queue.

When a process is roadblocked for a message, it sleeps at positive priority, so that signals (and events) may be captured in a timely fashion.

A process's message queue is not inherited across *fork* or *exec*.

A process may send a message to itself if it is so inclined, but it may not wait for a message from itself.

5. Events

5.1. Synopsis

The event facility permits processes to exchange user defined timing signals. The actual mechanics of sending an event signal are similar to sending a message, insofar as the sending process is concerned. The sender specifies an eight bit event pattern and the processid of the target process. Logically, each bit in the pattern is viewed as representing some event that is meaningful to the sender and receiver. Before an event may be sent, however, the receiving process must have enabled event reception and included the sender in the class of processes that are eligible to send it events. Assuming that these prerequisites are met, the event pattern is *or*ed with the recipient's event word. The sender is then at liberty to continue execution.

There are two courses of action available to a process that has already enabled event reception and wishes to determine when a particular event occurs. It may choose to wait for one or more significant events, in which case it invokes the appropriate system call and indicates the awaited event(s) by providing an event mask. The mask is nothing more than a bit map of the event(s) in which it is interested. If the process's event word has already been posted with one of the desired events (that is, if the *or*ing of the event word and event mask is not zero), the process continues execution. However, if none of the necessary events have been posted, the process is roadblocked until one of them occurs. The other option available to the process is to catch significant events asynchronously. To do this, the process specifies a mask and the address of a function in the user's process. The process then continues execution, but when one of the desired events occurs, control is given to the indicated function. This allows processes to catch events in a manner analogous to the way a *signal* is caught.

Each process that enables events has an event word for the posting of events by other processes. The low eight bits of that word are available to the process for defining its own events. The remaining eight bits are reserved for system defined events. At present there are only two such events: death of a child process and receipt of a message on the message queue. The techniques described for learning of event occurrence are the same for both the user defined and the system defined events.

5.2. Assembler Interface.

Call.

```
event = 40.; not in assembler
sys event; function; pattern; addr
```

The low byte of the argument *function* determines the event request type. The permissible values are as follows.

0 Event reception is enabled for the calling process. No events may be posted for the process until this is done. The high byte of *function* designates the class of processes that may post events for this process: 0 for anyone (no restriction), 1 for same userid only, and 2 for same groupid only.

1 Event reception for the process is disabled. No more events for this process may be posted, and any pending events are lost.

2 The event pattern specified is sent to the designated process.

3 The execution of the process is suspended until one of the events indicated in the event mask is posted. If one has already occurred, return is immediate.

4 A function and event mask are defined to the system. The process is then free to continue execution. When one of the designated events is posted, the user's event handling function is given control.

The contents of the argument *pattern* depend on what operation is being requested. When sending an event, the low byte of this argument is the event pattern being sent. When a process is catching events, either synchronously or asynchronously, this argument contains the event mask that informs the system of which events are of interest.

Likewise, the contents of *addr* depend on the context of the system call. When sending an event, this argument contains the processid of the destination process. If the request is to define an event catching function, *addr* specifies the address of that function.

The argument *pattern* is not used when enabling or disabling events. The argument *addr* is not used when enabling or disabling events or when waiting for a significant event to happen.

Returns. If any request is unsuccessful, then the error bit (c-bit) is set, and r0 contains the error number. When the operations of enabling or disabling events or sending an event are successful, r0 is always zero. When an event is awaited and occurs, r0 contains the event word. The address of the previously defined event handling function (zero if there was none) is returned in r0 when such a function is being defined.

Special Notes. When a significant event is detected synchronously (the await event operation), the complete contents of the event word are returned to the process (in r0) and the event word and event mask are cleared. It is left as an exercise for the user to determine which particular event(s) in the mask occurred.

In asynchronous event detection, the complete contents of the event word are made available to the event catching function by placing them on the user's stack. The return from the function is accomplished by popping this word from the stack and executing a RTI instruction. Catching an event asynchronously clears the event word, event mask, and the address of the event catching routine. The function must be redeclared if future events are to be caught asynchronously.

5.3. C Interface

Calls.

```
enabev(permission)
int permission;
```

Event reception is enabled by the *enabev* function. No events may be sent to the process before this is done. The *permission* stipulates who is eligible to send events to this process: 0 for anyone, 1 for same userid only, and 2 for same groupid only.

```
disabev()
```

The *disabev* function disables event reception for the invoking process. All pending events are lost.

```
sendev(topid, event)
int topid, event;
```

The event pattern specified in the low byte of *event* is sent to process *topid*. The receiving process must have previously enabled event reception and included the sender in the class of processes eligible to send it events.

```
await(mask)
int mask;
```

The calling process is roadblocked until one of the events specified in *mask* occurs. Return is immediate if one has already been posted. Processes wishing to limit the length of their sleep should avail themselves of the *alarm* and *pause* facilities that are now available on standard UNIX.

```
event(mask, func)
int mask, (*func)();
```

The function *event* defines the address of a function *func* that is to receive control when one of the events indicated in *mask* is posted. This call must be reissued after a significant event occurs if future events are also to be caught asynchronously. The complete contents of the event word are passed to the event handling function *func* as an argument:

```
func(eventword)
int eventword;
```

Returns. A -1 (not just negative) return from any of these functions indicates an error; the error number is contained in the external variable *errno*. A return of zero from the *enabev*, *disabev*, or *sendev* functions indicates success. The *await* function returns the complete contents of the event word. For the *event* function, the address of the previously defined event handling function (zero if none) is returned on success (this function fails if event processing has not been enabled).

5.4. Rules of the Road

A process may not send events to a process that has not enabled them or included the sender in the class of processes to which it is willing to listen.

Super-user may send events to any process that has enabled them.

The effective userid and groupid are used to determine if a process may send events to another.

A *disabev* is implicit when a process terminates. Thus, events may not be sent to non-existent processes.

Event processing is not inherited across *fork* or *exec*.

If a process is catching both signals and events asynchronously and both occur, the signal handling takes precedence.

If an *event* function is followed by an *await* function, the effect of the former is negated.

Waiting for a significant event to occur *(await)* is done at a positive priority, so that signals sent to the process may be caught in a timely fashion.

6. Shared Memory

6.1. Synopsis

The shared memory facility allows cooperating processes to share portions of physical memory. This is the fastest method of data exchange between processes. Each shared memory segment requires a hardware segmentation register. Thus, if the user process does not have separate instruction (I) and data (D) spaces, at most six segments may be shared. The maximum segment length is 4K words. Synchronization of processes sharing memory is left as an exercise for the user; other interprocess communication facilities, such as semaphores or events, may be utilized for this purpose.

The first step in establishing a shared memory segment is the allocation of the segment. This allocation is done by only one of the cooperating processes. The process is advised of the segment descriptor assigned to the segment by the system; all other processes wishing to share the segment then reference this descriptor when making their requests to share the memory. It may be noted in passing that the segment descriptor is in no way related to the actual segmentation register used to establish access to the segment. Two permissions are associated with each segment when it is allocated: a software permission and a hardware permission. The former, as with other interprocess communication facilities, delineates the class of processes that are allowed to share the segment. The latter is the actual permission used in the hardware segmentation register when a sharing process establishes access. These permissions are no access, read only, or read/write. The system always chooses a process's first available segmentation register when setting up a shared memory segment; the user has no direct control over register choice.

Once the segment is allocated, other processes may request sharing. Access to all or just a portion of the segment may be requested. Thus, it is possible to subdivide a shared memory segment in such a manner that processes are hardware protected from encroachment by other processes using another portion of the same segment.

Both permissions associated with a shared memory segment remain in effect throughout the segment's existence, even if it has been freed by the original allocator. The shared segment itself exists as long as any process is attached to it; once it is deallocated by all sharing processes, the memory is returned to the system for reallocation.

6.2. Assembler Interface

Call.

```
maus = 49.; not in assembler
sys maus; function; ptraddr; size; arg4
```

The low byte of the argument *function* is used to specify the request type. The permissible values are as follows.

0 A shared memory segment of the requested size is allocated and assigned the indicated permissions. The high byte of *function* stipulates the class of users that may share the allocated segment. The software permissions are 0 for anyone (no restriction), 1 for same userid only, and 2 for same groupid only.

1 The shared memory segment whose segment descriptor is contained in r0 is disassociated (freed) from the calling process.

2 The process's segmentation registers are set up so that a segment previously allocated by another process may be accessed. The desired segment's descriptor must be in r0. Of course, the requesting process must have the proper software permission before this is done.

The virtual address to be used by the process in accessing the shared segment is placed in the word addressed by the *ptraddr* argument.

The *size* argument is used, when initially allocating a shared memory segment, to specify the length, in bytes, of the segment. The system will round this up to a multiple of 64 bytes if it is not already so. When requesting access to a previously allocated segment, *size* indicates how much of the segment is to be made accessible to the process.

Arg4's use depends on the request type. When a segment is initially being allocated, it is used to specify the hardware access permission that will be given to other processes that use the segment: 0 for no access, 2 for read only, and 6 for read/write. If, on the other hand, the request is to establish access to a previously allocated segment, *arg4* indicates the offset, in bytes, within the segment where access is to begin.

The arguments *ptraddr, size,* and *arg4* are ignored when the invoking process's request is for disassociation with a segment.

It should be noted that when a shared segment connection is established for a process, the actual access granted may be somewhat at odds with the requested offset and size. In particular, because the segmentation registers only permit access to memory in 64 byte multiples, the value for *arg4* (offset) and *size* may need to be rounded, respectively, down and up, so that the desired portion of the segment is enclosed.

Returns. For unsuccessful requests, the error bit (c-bit) is set and the error number is in r0. If the request is for initial allocation of a segment, the segment's descriptor is returned in r0. In successful requests to share a previously allocated segment or to free a shared segment, zero is returned in r0.

6.3. C Interface

Calls.

```
getmem(&ptr, size, hwperm, swperm)
char *ptr;
int size, hwperm, swperm;
```

A shareable segment *size* bytes long is allocated by using the *getmem* function. The actual length of the segment is *size* rounded to the next 64 byte multiple. *Size* may not exceed 8192 bytes. The virtual address that is to be used in referencing this memory is placed in *ptr.* Only processes that are in the class defined by *swperm* may share this memory. The permissible classes are 0 for anyone (no restriction), 1 for same userid only, and 2 for same groupid only. Other processes sharing this segment will be assigned the hardware access permission stated in *hwperm:* 0 for no access, 2 for read only, or 6 for read/write.

```
freemem(segd)
int segd;
```

The invoking process disassociates itself from the shared memory segment with descriptor *segd.*

```
sharemem(&ptr, segd, offset, size)
char *ptr;
int segd, offset, size;
```

A process gains access to the previously allocated shared memory segment *segd* by using the *sharemem* function. If access is granted, the variable *ptr* will contain the virtual address to be used in accessing the segment. The process will have access to *size* bytes of the segment, starting at *offset* within it. In actuality, because of PDP-11 architecture, access will start at *offset* rounded down to a 64 byte boundary, and will extend to *offset+size* rounded up to a 64 byte boundary.

Returns. A return of -1 from any of these functions indicate failure; the error number is contained in the external variable *errno*. The *getmem* function returns the segment descriptor of the allocated segment when successful, while *freemem* and *sharemem* return zero.

6.4. Rules of the Road

All of a process's shared segments are freed when the process terminates.

It is erroneous to attempt to allocate a shared segment larger than 8192 bytes. Likewise, in attempting to share a segment, *offset+size* may not exceed the segment's length.

Shared segments are inherited across *fork*, but not across *exec*.

The first available segmentation register is used when a process allocates or shares a segment.

The allocator of a shared memory segment is given read/write permission to it. Other sharing processes are given the hardware access permission stipulated by the allocator.

Super-user is given access permission to share any segment. The hardware permission for the segment may not be overridden, however.

Conclusion

A functional description for a set of UNIX interprocess communication facilities has been presented. The actual implementation and testing is, of course, the next step. Any UNIX project that would like to acquire and test a prerelease version of any (or all) of these new features on a *friendly* user basis is encouraged to contact the author.

R. B. Brandt

MH-8234-rbb-roff

20