UCRL-ID-120484

Smile

J.G. Fletcher

April 1988

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Smile

John G. Fletcher
April 15, 1988

This document defines the characteristics of Smile, a System/machine-independent local environment. This environment consists primarily of a number of primitives (types, macros, procedure calls, and variables) that a program may use; these primitives provide facilities, such as memory allocation, timing, tasking and synchronization, beyond those typically provided by a programming language. The intent is that a program will be portable from system to system and from machine to machine if it relies only on the portable aspects of its programming language and on the Smile primitives.

For this to be so, Smile itself must be implemented on each system and machine, most likely using non-portable constructions; that is, while the environment provided by Smile is intended to be portable, the implementation of Smile is not necessarily so. In order to make the implementation of Smile as easy as possible and thereby expedite the porting of programs to a new system or a new machine, Smile has been defined to provide a minimal portable environment; that is, simple primitives are defined, out of which more complex facilities may be constructed using portable procedures. The implementation of Smile can be as any of the following:

- o the underlying software environment for the operating system of an otherwise "bare" machine,
- o a "guest" system environment built upon a preexisting operating system,
- o an environment within a "user" process run by an operating system, or
- o a single environment for an entire machine, encompassing both system and "user" processes.

In the first three of these cases the tasks provided by Smile are "lightweight processes" multiplexed within preexisting processes or the system, while in the last case they also include the system processes themselves.

Language

Much of the environment for a program is provided by the programming language in which it is written, and Smile is intended to provide only facilities not provided by that language. Therefore Smile is based upon an assumption about what facilities are available in the programming language used. The simplest statement of this assumption is that the language provides the kinds of facilities available in the language C; these include common data types, structures, arrays, pointers, common control structures, potentially recursive subroutines (that can be implemented using a stack), and simple syntactic macros. Also the language used must compatibly link with C, because it has been assumed that C is a suitable portable language in which to implement Smile itself.

[illegible]

The Smile primitives that are not simply types, variables, or constants are here specified as though they are procedures in the C language. This does not imply that implementation in another language is inappropriate. More importantly, it does not imply that the primitives are in fact procedures; some of them may be implemented as macros, and the programmer must take the precautions implied by that possibility.

The discussion below covers, not only the general, implementation-independent characteristics of Smile, but also some aspects peculiar to a specific implementation, a C-language implementation within the Unix operating system, here called the <u>reference</u> implementation; it should always be clear what in the discussion is intended to be generally true and what is true only of the reference implementation. The reference implementation of Smile is itself largely portable, the system dependencies being isolated into specific modules.

Smile is an <u>environment</u>. Routines that execute within that environment should simply assume that it is there; they do not have to do anything to create it. The environment is created by the action of some program executing outside the environment, and, once existing, the environment persists until activity within it indicates that it should terminate; creation and termination of Smile are discussed further below. Routines executing within Smile and routines outside Smile may be able to communicate, for example, through shared global variables; such communication is also discussed below. However, the discussion now continues by describing Smile from the point of view of programs executing within that environment.

Memory

The (random access) memory available to a program, which (depending on the implementation) may be "real" or "virtual", consists of several parts:

o <u>Procedures</u> contain the instructions that executing <u>routines</u> read but do not modify. Therefore the same procedure may be reentrantly and/or recursively used by several routines. The term routine is here used to refer to an "instance of execution" of a procedure, that is, an object represented by an "activation record", "stack frame", or other rendition of an internal, evolving state.

o <u>Stacks</u> contain the filo (<u>f</u>irst <u>i</u>n, <u>l</u>ast <u>o</u>ut) state of routines. Routines that operate entirely among themselves in a filo fashion can share a single stack for this state, which is allocated and deallocated as the routines are created and terminate.

o <u>Globals</u> are permanently allocated variables and constants accessible to routines, in addition to the local variables that constitute their state on a stack.

o <u>Blocks</u> each contain a sequentially addressed portion of memory of some size that may vary from block to block. Unlike procedures, stacks, and globals, which are assumed to be allocated and/or managed automatically by the mechanisms of the programming language, blocks are allocated and managed explicitly by routines' calling upon Smile primitives, as discussed next.

Smile measures the size of memory blocks in units of (8 bit) bytes. It is recognized that some implementations (for example, those using a "buddy system" allocator) may have particular sizes of blocks that they preferentially allocate, that they may round up the size of a requested block to the next preferred size, and that there may be efficiencies in referring to a preferred size by other than its byte count. Smile therefore recognizes the notion of a size code, a negative integer that identifies one of the preferred sizes. So, when requesting a block, a routine may specify the desired size either with a byte count or with a size code, the two cases being distinguished by the fact that the latter are negative, while the former are not.

Smile recognizes that the memory may consist of a number of partitions, each partition being an independently addressed segment of memory. A typical case is that there is one partition for the system and one for each "user" process. However, it is only when there is a single Smile, encompassing both system and "user" processes, that such partitioning is relevant; a Smile that is limited to just the system or just a single "user" process typically must deal with only a single partition. The reference implementation consists of separate Smiles for the system and for each "user" process (that uses Smile); therefore, each such Smile has only one partition.

If there are multiple partitions within a Smile, then the notion of a privileged procedure is relevant. A privileged routine (one executing a privileged procedure) is able to access all partitions, while an unprivileged one is limited to just one partition, with which it is associated. The intended implementation is that system routines are privileged and "user" routines are not, except that certain privileged procedures are "system calls", invocable by unprivileged "user" routines. If there is only one partition, it is generally best to regard all procedures as privileged. The partitions accessible by a routine define the scope of that routine; that is, the scope of an unprivileged routine is its partition, while the scope of a privileged routine is the entire Smile. The extent to which restrictions, such as those on unprivileged routines, are fully enforced by Smile is system-dependent.

There are four Smile primitive procedures relating to memory blocks:

o int Memfnd (count)
 int count;
returns the size code appropriate to the (byte) count argument, that is, the code corresponding to the preferred size obtained by rounding up that count. If the count is negative (indicating that it is in fact, not a byte count, but a size code), then that value is returned. In an implementation for which there are no preferred sizes, the count itself may be returned as what is, in effect, a (non-negative) size code. Memfnd(0) should always be 0. Portable programs should not have size codes built into them; they should use Memfnd to find the size codes corresponding to

the byte counts of interest, and then use the results obtained (which may or may not be negative) thereafter.

-
  ```
  int Memsiz (code)
   int code;
  ```
  returns the byte count corresponding to the (size) code argument. If the code is non-negative (indicating that it is in fact, not a size code, but a byte count), then that value is returned. Note that Memfnd(Memsiz(code)) == code but that Memsiz(Memfnd(count)) >= count.

-
  ```
  char *Memobt(mode, size)
   char mode;
   int size;
  ```
  allocates a block of the size indicated by the size argument and returns a pointer to it or, if no such block is available, returns a Null value. The size may be either a byte count or a size code, although the latter may be more efficient; a size of zero is valid, although perhaps not very useful. The mode argument may be either 0 or 1. If the mode is 0, then the allocated block is within the partition of the routine invoking Memobt. If the mode is 1, then the invoking routine must be privileged, and the allocated block is within a special partition (e. g., the system) accessible only by privileged routines. The mode is of course not significant if there is only one partition (as is the case for the reference implementation).

-
  ```
  char *Memgiv(block, size)
   char *block;
   int size;
  ```
  discards the previously allocated block defined by the block and size arguments and returns a Null value; the size should be the same as was specified when the block was allocated. A Null value for block results in no operation.

Time

Smile defines a type When, which is the type for time intervals; this type is necessarily actuallly some form of integer, so that arithmetic can be performed. There are two Smile primitive procedures relating to time:

- When Tick()
  indicates the units in which time is measured by returning the number of such units that make up one second. A particular time interval of n seconds is therefore expressed as n*Tick().

- When Time()
  returns the current time measured in the time units indicated by Tick. The current time in seconds is therefore Time()/Tick(). The initial instant from which this time is measured is system dependent and not defined within Smile; that is, Smile provides only for measuring time <u>intervals</u>, not for calendar time or time of day.

Tasks

A _task_ is a collection of routines that is scheduled for execution by Smile, so that one task is executed for a while, then another, and so on, the execution of each task generally involving the successive invocations of a number of routines. Tasks are threads of execution that, at least logically, execute asynchronously, concurrently, in parallel with one another. A task has an evolving state that consists primarily of the states of the routines making it up. This state is therefore on a stack associated with the task, and each task has its own stack (if coroutines are provided by the language, which they are not in the reference implementation, then a single task might require multiple stacks). In addition to whatever state is on its stack, each task has some additional state accessible using the Smile primitives, as explained below. A task is at any moment in one of four states:

o An _inactive_ task has only whatever state is accessible through the Smile primitives; none of its stack is actually in use, and therefore the stack need not, and in fact does not, exist. So the inactive state provides a way for a task to minimize the resources it requires during periods of inactivity; also, having no stack at such times is essential for a kind of task discussed below. If the task is not inactive (is in one of the three states discussed next), then it is said to be _active_ and has a stack with some state on it.
o A _sleeping_ task differs from an inactive task only in that it has some state on its stack. An inactive or a sleeping task is quiescent and will not execute until some event (as discussed below) causes it to become _awake_ (a term that encompasses the two states discussed next).
o A _ready_ task is prepared to execute and is only waiting its turn to use a processor and enter the state described next, as determined by the Smile scheduler.
o A _running_ task is actually executing on a processor.

Each task, and all the routines in it, are associated with a single memory partition.

Smile defines a type Task, which is the type for identifiers of tasks. Routines using Smile should only assign such identifiers as the values of variables, pass them as arguments to procedures, or otherwise use them in a manner not dependent on how the type is actually implemented. For example, in the reference implementation, Task is actually a pointer to a data structure containing information about a task needed by Smile to manage that task, but routines using Smile should not attempt to access items in that structure. There are eight Smile primitive procedures relating to tasks:

o
```
Task Tskobt (entry, parameter, mode, size)
 char *(*entry)();
 char *parameter;
 char mode;
 int size;
```

creates a new task and returns its task identifier or, if no such task can be created, returns a Null value. (No actual task has a Null identifier.) The task will begin execution at the beginning of the procedure defined by the entry argument; that procedure is passed, as its single argument, the parameter argument, which is known as the task parameter. That is, the initial or main routine of the task executes the entry procedure with the task parameter as its argument; this routine typically invokes other routines as it carries out the work of the task. The mode argument may be either 0 or 1. If the mode is 0, then the new task is within the partition of the routine invoking Tskobt. If the mode is 1, then the invoking routine must be privileged, and the new task is the only task of a new partition; this of course is not permitted in implementations (such as the reference implementation) that only allow one partition. The size argument, which may be either a byte count or a size code, specifies the size of the stack for the new task; a special meaning for a size of zero is discussed below.

- Task Ego ()
  returns the identifier of the running task (a routine of which invoked Ego). This is an item of a task's state in addition to what is on its stack, albeit a constant one.

- void Sleep ()
  causes the executing task to suspend execution and enter the sleeping state. There are three ways in which a running task ceases running:
  - It may be preempted at an arbitrary moment by the Smile scheduler, in which case it becomes ready; preemption is discussed further below.
  - It may execute Sleep(), in which case it sleeps.
  - It may return from its main routine; there are two cases:
    - If the value returned is not Null, then the task becomes inactive (and its stack is discarded). The returned value is remembered by Smile as a new value for the task parameter; when the task again becomes active, then this value will be passed to the entry procedure as its argument, and the task will resume execution there. That is, each time that a task is reactivated, it gets a new stack and executes its entry procedure, and that procedure is passed as its argument the task parameter, which is redefined each time that the task becomes inactive. The task parameter is the remaining item in a task's state (in addition to the state on its stack and its identifier, obtainable by calling Ego); the intent is that the task parameter typically be castable to a pointer to a data structure that contains all the state of the task (except for its identifier) that is to survive while it is inactive.
    - If the value returned is Null, then the task is permanently terminated, and its task identifier becomes meaningless. Terminating a task disposes of its stack and any Smile records associated with it. It does not dispose of anything else, such as blocks allocated by

the task or whatever the task parameter may have pointed to. If all the tasks in a partition terminate, then the partition terminates, and if all the partitions terminate, then Smile itself terminates (this is discussed further below).

Note that a task does <u>not</u> stop running because it creates (with Tskobt) or alerts (as described next) another task.

o void Alert (task, interval)

```
 Task task;
 When interval;
```

wakens the task identified by the task argument after the time interval defined by the interval argument; if task is Null or is not within the scope of the routine invoking Alert, then no operation is performed. After an inactive or sleeping task is wakened, it becomes ready. When a task that is already awake is wakened, it becomes "hyper" and will immediately rewaken the next time that it executes Sleep or returns from its main routine with a non-Null value. More precisely, the Smile scheduler remembers for each task whether or not it is awake (one bit) and an <u>alarm time</u>; it obeys the following rules:

- o When an inactive or sleeping task becomes awake, its alarm time is recorded as <u>eternity</u> (a time so far into the future that it will never be reached).
- o Whenever Alert is called, the alarm time of the indicated task is recorded appropriately (i. e., as the time that is the indicated interval into the future from the current time), unless the alarm time already recorded is earlier than the one about to be recorded; this is so even if the task is awake. That is, if a task is alerted several times after it has become awake (and then perhaps subsequently gone to sleep or become inactive), then only the earliest of the indicated wake up times is effective.
- o A sleeping or inactive task becomes awake if its alarm time equals or precedes the current time. If a task attempts to go to sleep or become inactive when the its alarm time already equals or precedes the current time, then it is viewed as momentarily ceasing and then immediately resuming being awake; so its alarm time gets rerecorded as eternity. A newly created task is inactive but has an alarm time equal to the time of its creation; so it becomes awake as soon as the Smile scheduler can act.

The intended behavior for a task is that, before it goes to sleep or becomes inactive, it should first be certain that there is nothing more for it to do. Also, one task should alert another only after it has indicated to that task (e. g., by recording suitable information in shared memory) what work there is to do. It is then the case that a task will never be left asleep or inactive while there is work for it to do, which could be quite disastrous. However, tasks will sometimes be wakened "spuriously", that is, only to find that there is no new work, which, while perhaps inconvenient, has no severe consequences. Three useful special cases of Alert have been defined:

- o Wake(task), equivalent to Alert(task, 0), wakens the indicated task now.

- o Alarm(interval), equivalent to Alert(Ego(), interval), is used by a task to set an alarm for itself. Note that Alarm(0) is equivalent to Wake(Ego()).
- o Pause(), equivalent to (Alarm(0), Sleep()), causes a momentary hiatus in the execution of a task, permitting other tasks to use the processor. If a task goes to sleep or becomes inactive when the current time has already overtaken its alarm time, and the task therefore immediately reawakens, then the Smile scheduler nevertheless does not immediately continue to run the task; instead it carries out its scheduling algorithm, as it always does when a task ceases to run, even for a moment.

o char *Tskget (task)

```
 Task task;
```

returns the task parameter of the task indicated by the task argument; if task is Null or is not within the scope of the routine invoking Tskget, then a Null value is returned.

o char *Tskput (task, parameter)

```
 Task task;
 char *parameter;
```

returns the parameter argument, after making that parameter the task parameter of the task indicated by the task argument; if task is Null or is not within the scope of the routine invoking Tskput, then the task parameter is not changed, although the value of the parameter (argument) is returned.

o void Lock ()

provides a primitive synchronization among tasks. It is used in conjunction with Unlock (described next).

o void Unlock()

is used with Lock to synchronize tasks. The effect of Lock extends to all tasks within the scope of the routine invoking Lock. If a routine calls Lock and returns from it, then, until it subsequently calls Unlock, any other routine with the same scope that calls Lock will not return from it but will behave as though it is repeatedly calling Pause(); when Unlock is called by a routine with that same scope, then, if any routines are so "trapped" by the Lock, one of them will return from it. This is a very simple synchronization facility out of which more complex facilities, such as semaphores, can be constructed. The simplicity of the facility is further guaranteed by the following restrictions on its use:

- o A task that has called Lock without yet subsequently calling Unlock should not call Lock again, or call Sleep, or return from its main routine.
- o A task should not call Unlock unless it has first called Lock, and its calls to Lock and Unlock should alternate.

If any of these restrictions are violated, then the behavior is undefined.

An interrupt task is a task created by calling Tskobt with a size of zero. Whenever an interrupt task becomes active, it uses the top of the stack of some other task; that other task can of course not run again until the interrupt task has become inactive once more. Stack sizes must of course be selected to provide for their possible use by interrupt tasks, if there are any. An interrupt task typically is preemptively scheduled upon the occurrence of some event (a form of scheduling not provided in the reference implementation); such a task normally never calls Sleep, so that, when it is not awake, it is inactive and has no stack.

An implementation of Smile may be either preemptive or not. A global constant Prempt, provided by Smile, is zero if and only if the implementation is not preemptive and uses only a single processor. The reason that preemption and multiprocessing are thus linked together is that, from the point of view of programs using Smile, if either is present, then "critical" sections of program must be protected, using constructions typically built out of Lock and Unlock. So, if Prempt is zero, then these constructions may be conditionally compiled into no operations; in fact, if Prempt is zero, then Lock and Unlock themselves are typically implemented as no operations. (Also, in the case of a single processor with preemption, Lock can be implemented as disabling preemption and Unlock as reenabling it.) However, if there is no preemption (even though there may be multiprocessing), then programs may have to call Pause at suitable intervals or otherwise assure that no task can indefinitely occupy a processor. To be prepared for both the possibility of preemption and the possibility of no preemption, a programmer must provide both critical regions and strategic pauses.

Permanent State

A system typically has some state characterizing its behavior that should survive, even if the system "crashes" and has to be "dead started". There are two Smile primitive procedures, callable only by privileged routines, for saving and retrieving permanent state:

o
```
int Retrv (buffer, size)
 char *buffer;
 int size;
```
fetches the permanent state into the memory area indicated by the buffer and size arguments and returns zero, or it returns a non-zero value if such cannot be done; the reason for the failure in the latter case may be encoded into the non-zero value, but Smile does not define the meanings of such values. The size may be either a byte count or a size code. If there is more permanent state that the size indicates, then only the first part of that state is fetched; if there is less, then the end of the buffer is padded with zeros. The program working with the permanent state is expected to know its format and size. Typically a system using Smile calls Retrv exactly once, while initializing.

o
```
int Save (buffer, size)
```

```
 char *buffer;
 int size;
```

stores as the permanent state the information from the memory area indicated by the buffer and size arguments and returns zero, or it returns a non-zero value if such cannot be done; the reason for the failure in the latter case may be encoded into the non-zero value, but Smile does not define the meaning of such values. The size may be either a byte count or a size code. It is intended that all the permanent state is being stored; the behavior if the size is more or less than the available space for that state is defined only to the extent what Retrv fetches includes everything that the most recently preceding Save has stored, provided that that does not exceed the available space. Typically a system using Smile calls Save rarely, only when permanent characteristics are redefined.

Symbolic Constants

The primtive procedures with no arguments can also be treated as symbolic constants, which Smile has defined as follows:

```
#define TICK Tick()
#define TIME Time()
#define EGO Ego()
#define SLEEP (Sleep(), (char *) Null)
#define PAUSE (Pause(), (char *) Null)
#define LOCK (Lock(), (char *) Null)
#define UNLOCK (Unlock(), (char *) Null)
```

Booting

To bring Smile into existence a program (which of course already exists outside that environment) should call the following procedure:

o

```
int Smile (entry, parameter, size)
 char *(*entry)();
 char *parameter;
 int size;
```

initializes the environment, in effect executes Tskobt(entry, parameter, 0, size) to create a single task (the mode of 0 indicating that that task shares memory with the program that called Smile), and then causes that task to run. That task in general creates more tasks, and the several tasks run under the control of the Smile scheduler. Smile returns only if all the tasks terminate; the returned value is zero unless there was some form of failure (e. g., stack overflow). However, some implementations of Smile may never return, even when no tasks remain; such implementations provide for resumption of activity in a manner such as is discussed below. The size argument must be non-zero, unless <u>all</u> tasks are to be interrupt tasks, in which case there is only one stack, namely that of the routine invoking Smile.

So the idea is that, once Smile is brought into existence, it "takes over" and does not relinquish control until it has nothing more to do (and perhaps not even then). In practice this is not necessarily so, as is seen in the following discussion of the reference implementation.

Reference Implementation

It is possible for all the tasks to be sleeping or inactive at once; activity resumes when the current time reaches the alarm time of one of them. Even if all their alarm times are eternity, activity can still resume, because the reference implementation permits routines executing outside the environment of Smile to call Tskobt and Alert (and Wake and Alarm). Rather than simply aimlessly looping while waiting for one of the tasks to reawaken, when none are awake, the reference implementation gives up control of the processor by calling a suitable system-dependent procedure, indicating a time at which Smile wishes to resume execution; this procedure should return either when that time is reached or when Tskobt or Alarm is invoked. (For "user" processes, this procedure is a "system call" and should also provide for the needs of other facilities -- such as the APST primitives, discussed elsewhere -- that may have their own reasons for calling the system.) Furthermore, for the reference implementation in the system (i. e., not in a "user" process), if all the tasks terminate, then Smile does not return; instead it waits for Tskobt to be invoked and then resumes by running the task thus created.

For the reference "user" implementation, the programmer can supply a procedure called main, which (directly or indirectly through other procedures) can call Smile; after Smile returns, it may be called again. If no main procedure is supplied by the programmer, then a standard one is used, which has the following behavior:

- o The arguments of main commonly called argv and argc are stored into global variables Argv and Argc.
- o Then Smile(prime, (char *) Null, prmsz) is called; extern char *prime() and extern int prmsz must be defined by the programmer.
- o When Smile returns, the value it returns is returned by main.

The reference implementation package, programmed in C, is itself partially portable. The non-portable part (which must be reprogrammed for each new system or machine) is in files named "smpar" (system/machine parameters), while the portable part is in files named "smile"; in addition, a file named "smain" (Smile main) provides the default main procedure for a C program. To use the package, a program source file must include the following two header files in the indicated order:

- o smpar.h and
- o smile.h,

and it must be linked with the object files compiled from:

- o smpar.c,
- o smile.c, and
- o smain.c,

the last being used if and only if the program does not supply its own main procedure for C. The three object files are in the library file:

o liblincs.a;

the main procedure in the library will not be loaded if the program supplies its own.

Appendix: Remarks on Tasking

Tasking is in most situations probably the most important functionality provided by Smile, yet it is perhaps also the least familiar. This appendix provides some comments on a view of tasking that underlies the Smile design.

The purpose of multiple tasks is to provide multiple asynchronous, concurrent, parallel threads of execution. Sequential actions normally should be done using a single task. For example, a task, having carried out the beginning of some sequential activity, could _fork_ (i. e., create, by calling Tskobt) a second task to carry out the rest of the activity and then immediately terminate itself; a more straightforward and efficient approach would be to have the first task itself simply continue and complete the activity. Even when there must be many tasks, a good guideline is that each sequential activity is the work of a single task.

A common use of tasks is to perform simultaneously many similar sequential activities. Following the above guideline, each such activity can be assigned to a separate task, and the reentrant program for the tasks can for the most part be written as though there were only one such activity. For example, a server can be implemented as an unlimited collection of tasks, each waiting for a request from a client, getting the request, doing what is requested, replying, and then terminating. Of course, there cannot really be an unlimited number of tasks at any one time; so there is in fact only some small number of them that are waiting for requests, enough of them to handle any expected peak flurry of requests. When one of the waiting tasks receives a request, it immediately forks another task, so as to keep the number of waiting tasks at the desired number. Except for this act of forking, server initialization, and possible critical sections of program that can only be executed by one or a few tasks at a time, the program for each task is essentially what it would be if there were only a single client.

The preceding is an example of the general notion that multiple tasks can be used to wait for multiple events. Another example is that the APST communication primitives intend that each stream on which message activity is simultaneously awaited is handled by a separate task.

It is assumed that the language provides global variables (accessible by all routines in a partition) and local variables (accessible by a single routine). There is also usually a need for task variables (accessible by all routines in a task). Smile provides for these with the task parameter, which (after appropriate casting) is intended to be a pointer to a data structure that contains the task variables; unfortunately, these variables are a bit less convenient to access than global or local variables, because they involve a structure reference. The task variables also provide a way for the routine creating a task to pass to that task some parameters defining what it is to do.

In general, entry to and exit from a critical section of program, execution of which is limited to some number of tasks (often just one) at a time, must be controlled by some kind of lockout procedures (such as semaphore procedures). These procedures have their own internal lockout problems, which are intended to be solved by using the primitives Lock and Unlock. For example, a typical lockout procedure may involve examining a variable, conditionally changing that value, and perhaps manipulating a queue; these actions, which are brief, can be bracketed by Lock and Unlock. Lock and Unlock themselves are generally too primitive to be used as lockout procedures for extended critical sections.

Index