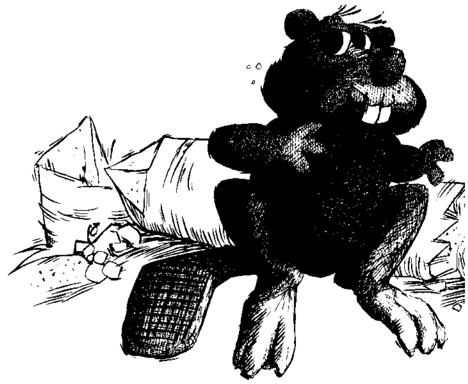
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



System Reference Manual
Version 4.2

Richard A. Stroobosscher
Peter A. Buhr

Research Report
CS-89-31

Revised May, 1990

μSystem Reference Manual

Version 4.2

Richard A. Stroobosscher and Peter A. Buhr

May 7, 1990

Contents

1. Introduction

The μSystem is a library of C [KR88] routines that provide light-weight concurrency on uniprocessor and multiprocessor computers running the UNIX[1] operating system. Concurrent operations in the μSystem are explicitly specified and not inferred from existing constructs in C. Users first design algorithms that are inherently concurrent and then explicitly code corresponding concurrent operations using the routines in the μSystem.

The μSystem uses a shared-memory model of concurrency. This shared-memory is populated by subroutines, coroutines and concurrently executing light-weight processes, called tasks. Coroutine mechanisms are provided to create coroutines within a task and to communicate information among the coroutines. Concurrency mechanisms are provided to create tasks, to synchronize execution of the tasks, and to communicate information between synchronized tasks. When shared memory exists between UNIX processes, UNIX processes are used as virtual processors and task execution is uniformly distributed across them. A clustering mechanism exists to group virtual processors and tasks together, restricting execution of these tasks to only these virtual processors. Partitioning into clusters must be used with care as it has the potential to inhibit concurrency when used indiscriminately. However, in several situations it will be shown that partitioning is essential. For example, concurrent UNIX I/O operations are possible through the clustering mechanism, when shared memory exists between UNIX processes.

The μSystem does not enter the UNIX kernel to perform a coroutine or task switch and uses shared memory among tasks. As a result, performance for execution and communication between large numbers of tasks is significantly increased over UNIX processes (e.g. two orders of magnitude in some cases). The maximum number of tasks that can be active is restricted only by the amount of memory available in a program. The minimum storage overhead for a task is machine dependent, but is as small as 256 bytes.

2. Compile Time Structure of a μSystem Program

A μSystem program is constructed exactly like a normal C program with one exception: the main (starting) routine is called **uMain** instead of the normal C name, **main**, for example:

```
... normal C declarations and routines

void uMain( int argc, char *argv[], char *envp[] ) {
    ...
}
```

The μSystem supplies and uses the **main** routine to initialize the μSystem runtime environment and create the first task which starts execution at **uMain**. The task **uMain** is passed the same three arguments that are passed to the routine **main**: **argc**, **argv**, and **envp**.

When **uMain** terminates, the current rule is that *all* other tasks are automatically terminated. It is not possible to start tasks that continue to execute after **uMain** terminates. Therefore, **uMain** must only terminate when the entire application program has completed. This rule was chosen because we found that managing multiple UNIX processes running in the background required too much knowledge from novice users. However, there is nothing in the μSystem that precludes supporting this feature.

3. Runtime Structure of a μSystem Program

The dynamic structure of an executing μSystem program is significantly more complex than a normal C program. There are four new runtime entities: coroutine, task, virtual processor, and cluster.

3.1 Coroutine

A coroutine is a program component whose execution can be suspended and resumed (see Reference [Mar80] for a complete discussion of coroutines). Execution of a coroutine is suspended as control leaves it, only to

[1] UNIX is a registered trademark of AT&T Bell Laboratories

carry on where it left off when control re-enters the coroutine at some later time. This means that coroutines are not entered from the beginning on each activation. In contrast, when a subroutine is invoked, it always starts execution from the beginning and its local variables only persist for that particular invocation. The state of a coroutine consists of:

- a current location which is initialized to a starting point and then traverses whatever part of the program that is reachable through the normal control-flow facilities.
- an execution state – blocked or active or terminated – which is changed by the coroutine constructs of the μSystem.
- a memory which holds the data items created by the code the coroutine is executing. This is the stack that contains the local variables for the coroutine and any subroutines called by the coroutine. This stack is the mechanism by which the local variables persist between successive activations of the coroutine.

As well, a coroutine identifier exists to reference the coroutine.

A coroutine executes synchronously with other coroutines created by the same task, and hence there is no concurrency among coroutines associated with a particular task. (Although, multiple instances of the same coroutine could be executing concurrently in different tasks.) While coroutines have no concurrency, they are valuable constructs in a programming language. A coroutine properly handles the class of problems that require state information to be retained between successive calls (e.g. finite state problems). Solutions to such problems without coroutines require variables with external visibility, or local visibility and static storage class. But since these variables are only allocated once, only one instance of such routines can be active. Because each coroutine has its own data area, multiple instances of the same coroutine can be active. Further, this class of problems illustrates the forms of control flow that are present in concurrent programs without the added complexity and expense of dealing with concurrent execution. Hence, coroutines are an intermediate step between subroutines and concurrent tasks, and valuable as a teaching device.

A μSystem coroutine is a C routine that is "cocalled". It can call or cocall any other C routine. A coroutine can interact with other coroutines by executing communication routines from the C routine started as the coroutine or from any of the routines it has called.

3.2 Task

A task is a program component with its own thread of control and has the same state information as a coroutine plus a task identifier. A task's thread of control is scheduled separately and independently from threads associated with other tasks. It is this thread of control that results in concurrent execution. On a multiprocessor computer, task execution is performed in parallel. On a uniprocessor computer, concurrency is achieved by interleaving of task execution to give the appearance of parallel execution. Because there may be more tasks to execute than processors to execute them, it is possible for a task to be ready to execute but not executing. Hence, tasks have one more execution state over a coroutine, the ready state.

Tasks are light-weight because of the low execution time cost and space overhead for creating a task and the many forms of communication which are easily and efficiently implemented for them. This is possible as all tasks in the μSystem execute within a single shared memory. This memory may be the address space of a single UNIX process or a memory shared between a set of UNIX processes. This has its advantages as well as its disadvantages. Tasks need not communicate by sending large data structures back and forth, but can simply pass pointers to data structures. However, there is no address space protection between tasks so one faulty task may overwrite another task's data area.

A μSystem task is a C routine that is "emitted". A task is composed of number of communicating coroutines. In theory, a C routine can be called, cocalled and emitted; in practise, the forms of communication used by a routine dictate how it must be started. When a C routine is emitted, a new thread of control is created and begins execution by cocalling the C routine; hence, the emitted task is also a coroutine. The coroutines created by a task's thread belong to that task and cannot communicate with coroutines from other tasks. This restriction follows naturally from the fact that only one thread can be using a coroutine's state, and in the μSystem, that thread is the one associated with the task that created it.

While a task that creates another task is conceptually the parent and the created task its child, the μSystem makes no implicit use of this relationship nor does it provide any facilities that perform actions based on this relationship. Once a task is emitted it has no special relationship with its emitter.

μSystem tasks are not implemented as UNIX processes for two reasons. First, UNIX processes have a high runtime cost for creation and execution. Second, each UNIX process is allocated as a separate address space (or perhaps several) and if the system does not allow memory sharing between address spaces, then tasks have to communicate using pipes and sockets. Pipes and sockets are expensive and would have to be used to simulate all the forms of interprocess communication that we intend to have. If shared memory is available, there is still the overhead of page table creation and management for the address space of each process. Therefore, UNIX processes are heavy-weight because of the high runtime cost and space overhead in creating a separate address space for a process, and the possible restrictions on the forms of communication among them. The μSystem provides access to UNIX processes only indirectly through virtual processors. A user is not prohibited from creating UNIX processes explicitly, but such processes will not be part of the μSystem.

3.3 Virtual Processor

A μSystem virtual processor is a "software processor" that executes tasks. A virtual processor is implemented as a UNIX process that is subsequently scheduled for execution on the actual processor(s) by the underlying operating system. Hence, the μSystem is not in direct control of the hardware processors; but when a virtual processor is executing, the μSystem controls scheduling of tasks on it. On a multiprocessor UNIX system, UNIX processes are usually distributed across the hardware processors. Because the UNIX processes execute simultaneously, the tasks executing on them will execute simultaneously. When multiple virtual processors are used to execute tasks, the μSystem scheduling may automatically distribute tasks between virtual processors and, thus, indirectly between hardware processors.

The μSystem uses virtual processors instead of actual processors so that programs do not actually allocate and hold hardware processors. Programs can be written to run using a large number of virtual processors and execute on a machine with a smaller number of actual processors. Thus, the way in which the μSystem accesses the concurrency of the underlying hardware is through an intermediate resource, the UNIX process. In this way, the μSystem is kept portable across different multiprocessor hardware designs. As long as the particular multiprocessor machine is running UNIX and has shared memory among UNIX processes, the μSystem can provide parallelism.

3.4 Cluster

A cluster is a collection of tasks and virtual processors that execute those tasks. Most programs will have only a single cluster as this will maximize utilization of virtual processors, which minimizes execution time. However, because of limitations of the underlying operating system or because of special hardware requirements, it is sometimes necessary to have more than one cluster.

A cluster uses a single-queue multi-server queueing model for scheduling its collection of tasks on virtual processors. This results in automatic load balancing of tasks on virtual processors. Figure 1 illustrates the runtime structure of a single cluster. An executing task is illustrated by its containment in a virtual processor. Because of appropriate defaults for virtual processors and clusters, it is possible to begin writing μSystem programs after learning about coroutines or tasks. More complex concurrent work may require the use of virtual processors and clusters. If several clusters exist, tasks can be explicitly migrated from one cluster to another. No automatic load balancing across clusters is performed by the μSystem.

When the μSystem begins execution, it creates two clusters: a system cluster and a user cluster. The system cluster contains a virtual processor which cannot execute user tasks. This is because the system cluster catches errors that occur on the user clusters, prints appropriate error information and shuts down the μSystem. A user cluster is created to contain the user tasks; the first user task is `uMain`. Most user applications will not explicitly create more clusters; however, certain operations may create clusters implicitly.

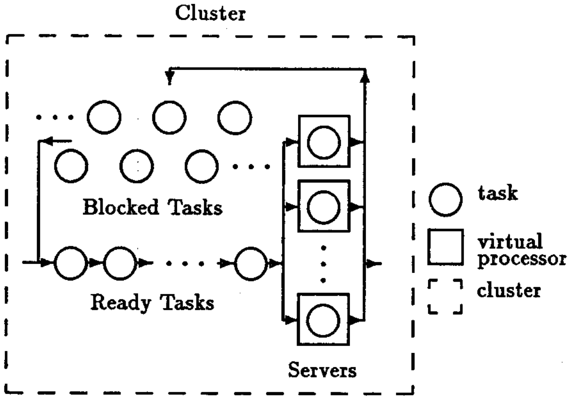


Figure 1: Runtime Structure of the μSystem

4. μKernel

The storage management of all objects in the μSystem, the scheduling of tasks on virtual processors, and the pre-emptive round-robin scheduling to interleave task execution is performed by the μKernel. The starting point for the μSystem was provided by the initial μKernel described in Reference [Cor88].

The μKernel exists in both a virtual uniprocessor and a virtual multiprocessor form, referred to as the unikernel and the multikernel. The form used depends on whether or not the UNIX operating system supports shared memory among UNIX processes. If there is no shared-memory between UNIX processes, the unikernel must be used. This limits all task execution to a single cluster containing a single virtual processor. If there is shared memory between UNIX processes, the multikernel can be used. If the machine has multiple processors, then tasks may actually execute in parallel if the UNIX processes execute in parallel. If multiple processors are not present, the multikernel can still take advantage of the shared memory so that problems like blocking UNIX operations can be handled.

While the interface to both kernels is identical, there are several differences between them, which all result from the unikernel having only one virtual processor. First, the semantics of the virtual processor and cluster routines are different for each kernel. In the unikernel, operations to increase or decrease the number of virtual processors are ignored, creation of a new cluster simply returns the current cluster, and destroying a cluster is ignored as there is only one. Hence, the system and user clusters are combined into a single cluster. Second, there is no parallelism in the unikernel so that concurrency must be simulated. This is done by a pre-emptive scheduling mechanism. The uniform interface allows almost all concurrent applications to be designed and tested on the unikernel, and then run on the multikernel after re-compiling.

The μKernel provides no support for automatic growth of stack space for coroutines and tasks because this would require compiler support. The μKernel has a debugging form which performs a number of runtime checks, one of which is to check for stack overflow whenever flow of control transfers between coroutines and between tasks. This catches most stack overflows; however, stack overflow can still occur if insufficient stack area is provided, which can cause an immediate error or unexplainable results.

5. Using the μSystem

To use the μSystem in a C program, include the file:

```
#include <uSystem.h>
```

at the beginning of each source file. This file also includes the following system files: `<stdio.h>`, `<sys/file.h>`, `<sys/types.h>`. These files are included to provide access to UNIX I/O, exception, and timing facilities.

5.1 Compiling μSystem Programs

Use the command concc to compile program(s) for the unikernel. This command works just like the UNIX cc command to compile C programs, for example:

concc *[C options]* *yourprogram.c* *[assembler and loader files]*

Use either the parcc command or the command concc with the `-multi` option to compile program(s) for the multikernel, for example:

parcc *[C options]* *yourprogram.c* *[assembler and loader files]*
concc `-multi` *[C options]* *yourprogram.c* *[assembler and loader files]*

The options available on the concc and parcc commands are:

-debug The user program is loaded with the debug version of the unikernel or multikernel. The debug version performs runtime checks to help during the debug phase of a μSystem program. This will slow the execution of the program down significantly. **This is the default.**

`-multi` The user program is loaded with the multikernel.

-nodebug The user program is loaded with the non-debug version of the unikernel or multikernel. **No runtime checks are performed so errors usually result in immediate program termination.** The runtime checks should only be removed after the program is completely debugged.

`-quiet` This suppresses printing of the μSystem compilation message at the beginning of a compilation.

`-compiler` *name* This specifies the name of the compiler used to compile the μSystem program(s). This allows compilers other than the default GNU C compiler to be used to compile a μSystem program using concc.

These commands are available by including `/u/ukernel/bin` in your command search path, which is usually located in your `.cshrc` file.

5.2 Preprocessor Variables

When programs are compiled using concc or parcc the following preprocessor variables are passed to the C preprocessor. If the `-multi` compilation option is specified, then the preprocessor variable `__U_MULTI__` is passed to the C preprocessor. If the -debug compilation option is specified, then the preprocessor variable `__U_DEBUG__` is passed to the C preprocessor. This allows conditional compilation of programs that must work differently in these situations.

5.3 Context Switching

A context switch occurs when control transfers from a coroutine or task to another coroutine or task. The switch involves saving the state of the currently executing party and restoring the state of the other party. In theory, the compiler can determine the state that must be saved. However, because the μSystem has no compiler support, it is necessary for a programmer to make part of this determination. All coroutines and tasks use the fixed-point registers, while only some use the floating-point registers. Hence, the fixed-point registers are always saved during a context switch, but it may or may not be necessary to save the floating-point registers. Because there is a significant execution cost in saving the floating-point registers, they are not automatically saved.

If a coroutine or task performs floating-point operations, then it must invoke the routine `uSaveFloat` immediately after starting execution. From that point on, both the fixed-point and floating-point registers are saved during a context switch. It is possible to revert back to saving just the fixed-point registers by invoking the routine `uSaveFixed`. However, in general, switching between saving fixed and floating registers in the same task is likely a dangerous programming practise. It is too easy to accidently put a floating point operation outside the range where the floating-point registers are saved.

5.4 Message passing

Except for arguments passed at creation, communication of information between coroutines and tasks is done by message passing. A message is a block of untyped bytes that is copied, as is, between communicating parties. The message is specified by a pointer to the block of bytes and the length of the block. The message receiver must provide the address of an area that is large enough to contain the message that is sent or the communication fails. The length of the message can be less than the length of the receiving area, but then the message must contain information on the actual length of the message. The message and the receiving area should be specified as data items of the same or structurally equivalent types. Unless the message type is an array or a pointer, the message data-item must be preceded by an `&`. The length of the message and the receiving area should be specified with `sizeof(`*data-item*`)`, unless the data item varies in size. There is essentially no limit on the size of a message (on some machines the implementation limits the length to 64K bytes). If no message is to be sent during a communication, the message pointer must be set to `U_NULL` and its length set to zero.

6. Coroutine Facilities

Like a subroutine, a coroutine can access all the external variables of a C program and the heap area. Also, any `static` variables declared within the definition of a coroutine are shared among all instances of that coroutine.

Two slightly different mechanisms are provided for passing control between coroutines. The first permits a coroutine to resume its invoker; the second permits it to resume an arbitrary coroutine. These two mechanisms are provided in order to accommodate two somewhat different styles of coroutine usage: a semi-coroutine which acts much like a subroutine by always resuming its invoker, and a full coroutine which acts somewhat like a task by resuming some other coroutine.

A coroutine is associated with the task that created it. If another task attempts to resume a coroutine that it did not create, an error will result. Since coroutines determine flow of control within a task, their execution is performed by one of the virtual processors associated with the cluster on which the task is executing.

6.1 Coroutine Type

`uCoroutine` is the type of a coroutine identifier, as in:

```
uCoroutine x, y, z;
```

which creates three variables that contain coroutine-identifier values.

6.2 Coroutine Creation

The routine `uLongCocall` starts a C routine running as a coroutine.

> *coroutine-id* = `uLongCocall(` *reply-area*, *reply-area-length*, *stack-size*,
> *routine*, *argument-length*, *arguments ...* `);`

coroutine-id is an instance of `uCoroutine` which is the coroutine identifier of the newly created coroutine and must be retained to subsequently communicate with the coroutine. No coroutine will ever have the identifier value `U_NULL`.

reply-area is the address of the reply area into which the reply message from the first suspend or resume of the newly created coroutine will be copied.

reply-area-length is the size in bytes of the reply area.

stack-size is the size in bytes of the stack that will be allocated for the coroutine.

routine is the name of a C routine to be called as a coroutine. The routine cannot return a value (i.e. it must have return type `void`) but may have any parameters allowed by C.

argument-length is the number of bytes that will be copied as arguments to the coroutine. This number should be the sum of the `max(sizeof(int), sizeof(`$argument_i$`))` for all arguments passed to *routine*.

arguments ... are any number of arguments passed as-is to *routine*. Because all arguments in C are passed by value, it is necessary to pass an argument's address if the argument is to be modified (e.g. *&argument*).

The cocaller suspends its execution at the `uLongCocall` and the coroutine begins execution just as if the C routine is called directly. The difference is that the coroutine is executing on its own stack.

The following example creates a new coroutine with a stack size of 8000 bytes, starting execution in routine `f` with an argument length set to the size of two floating point values and passing two floating point arguments:

```
uCoroutine corid;
float a, b, reply;
void f(float x, float y) { ... }      /* routine to be cocalled */
...
corid = uLongCocall( &reply, sizeof(reply), 8000, f, sizeof(a) + sizeof(b), a, b );
```

Because users rarely want to bother specifying explicit stack sizes and argument lengths, there exists a short form of the `uLongCocall` routine. `uCocall` performs the same function as `uLongCocall` using a default stack size and argument length. The following example starts `f` running as a coroutine using `uCocall`.

```
corid = uCocall( &reply, sizeof(reply), f, a, b );
```

The default values start at machine dependent values, which are no less than 4000 bytes for the stack size and 64 bytes for the argument length. Changing the defaults is discussed in the section on clusters.

The routine `uThisCoroutine` is used to determine the identifier of the current coroutine.

coroutine-id = `uThisCoroutine();`

coroutine-id is an instance of `uCoroutine` which is the coroutine identifier of the calling coroutine.

6.3 Coroutine Communication

The `uResume` and `uSuspend` routines are used to transfer control and communicate among coroutines.

The routine `uResume` suspends execution of the current coroutine and resumes execution of a specifically named coroutine.

`uResume(` *coroutine-id*, *reply-area*, *reply-area-length*, *send-message*, *send-message-length* `)`

coroutine-id is a `uCoroutine` identifier to a coroutine that is be resumed, passing it a particular message.

reply-area is the address of a reply area into which the reply message of a suspending coroutine will be copied.

reply-area-length is the size in bytes of the reply area.

send-message is the address of the message to be sent to the resumed coroutine.

send-message-length is the size in bytes of the message to be sent.

A resume operation establishes an implicit link from the resumed coroutine back to the resumer. This link is used by the uSuspend operation to perform an implicit resumption.

The routine uSuspend suspends execution of the current coroutine and resumes execution in the cocaller/resumer.

`uSuspend(` *reply-area*, *reply-area-length*, *send-message*, *send-message-length* `)` *reply-area* is the address of a reply area into which the reply message of a suspending coroutine will be copied.

reply-area-length is the size in bytes of the reply area.

send-message is the address of the message to be sent to the resumed coroutine.

send-message-length is the size in bytes of the message to be sent.

Routine call `uSuspend(...)` is essentially equivalent to `uResume(`*cocaller-id/resumer-id*`, ...)` except that the suspender's implicit link back to its resumer is set to U_NULL. Therefore, it is not possible to establish suspend-suspend cycles between coroutines.

6.4 Coroutine Termination

A coroutine is terminated by calling either the `uResumeDie` or the `uSuspendDie` routine. These routines can be invoked at any level of nested subroutine invocation to terminate the coroutine.

The routine `uResumeDie` terminates execution of the current coroutine and resumes execution of a specifically named coroutine.

`uResumeDie(` *coroutine-id*, *send-message*, *send-message-length* `);`

coroutine-id is a `uCoroutine` identifier to a coroutine that is be resumed, passing it a particular message.

send-message is the address of a message to be sent to the resumed coroutine.

send-message-length is the size in bytes of the message to be sent.

The routine `uSuspendDie` terminates execution of the current coroutine and resumes execution of the last cocaller/resumer.

`uSuspendDie(` *send-message*, *send-message-length* `);`

send-message is the address of a message to be sent to the resumed coroutine.

send-message-length is the size in bytes of the message to be sent.

Routine call `uSuspendDie(...)` is equivalent to `uResumeDie(`*cocaller-id/resumer-id*`, ...)`.

Executing a `return` statement in a cocalled routine is the same as the routine call `uSuspendDie(U_NULL, 0)`. This resumes the last cocaller/resumer and returns no message. The same action occurs if control runs off the end of the cocalled routine. Therefore, if a value is to be returned at coroutine termination, it must be passed back using one of `uSuspendDie` or `uResumeDie`.

The following example shows the simple case of a coroutine being used as a function.

```
void f(float x, float y) {
    float result;
    ...
    uSuspendDie(&result, sizeof(result));   /* return function result */
}

uMain() {
    float result, a, b;
    uCoroutine corid;
    ...
    corid = uCocall(&result, sizeof(result), f, a, b);
    ...
}
```

Appendix A contains a complete coroutine program.

7. Task Facilities

Like a coroutine, a task can access all the external variables of a C program and the heap area. However, because tasks execute concurrently, there is the general problem of several tasks accessing the same shared variables. Global references from tasks and `static` variables within a task that is instantiated multiple times can lead to inconsistent data values in these variables. The same problem can occur if a coroutine makes global references or has `static` variables and is instantiated multiple times by different tasks. Therefore, it is suggested that these kinds of references not be used or used with extreme caution. The μSystem provides routines to safely communicate information between tasks and allow safe access to the heap.

7.1 Task Type

`uTask` is the type of a task identifier, as in:

```
uTask x, y, z;
```

which creates three variables that contain task identifier values.

7.2 Task Creation

The routine `uLongEmit` starts a C routine running asynchronously with the calling task.

task-id = `uLongEmit(` *cluster*, *stack-size*, *routine*, *argument-length*, *arguments ...* `);`

task-id is an instance of `uTask` which is the task identifier of the newly created task and must be retained to subsequently communicate with the task. No task will ever have the identifier value `U_NULL`.

cluster is a `uCluster` identifier that this task is associated with. (Clusters are discussed in a following section. Most tasks are created on the current cluster, which is given by calling routine `uThisCluster()`.)

stack-size is the size in bytes of the stack that will be allocated for the task.

routine is the name of a C routine to be executed asynchronously. The routine cannot return a value (i.e. it must have return type `void`), but may have any parameters allowed by C.

argument-length is the number of bytes that will be copied as arguments to the task. This number should be the sum of the `max(sizeof(int), sizeof(`$argument_i$`))` for all arguments passed to *routine*.

arguments ... Any number of arguments passed as-is to *routine*. Because all arguments in C are passed by value, it is necessary to pass an argument's address if the argument is to be modified (e.g. `&`*argument*).

The following example creates a new task executing on the current cluster with a stack size of 8000 bytes, starting execution in routine `f` with an argument length set to the size of two floating point values and passing two floating point arguments:

```
uTask tid;
float a, b;
void f( float x, float y ) { ... }      /* routine to be emitted */
...
tid = uLongEmit( uThisCluster(), 8000, f, sizeof(a) + sizeof(b), a, b );
```

There is a short form of `uLongEmit`, called `uEmit`, that assumes the current cluster, a default stack size, and a default argument length. The following example starts `f` running as a task using `uEmit`:

```
tid = uEmit( f, a, b );
```

The default values for stack and argument length are the same as for coroutines.

The routine `uThisTask` is used to determine the identifier of the current task.

task-id = `uThisTask( );`

task-id is an instance of `uTask` which is the task identifier of the calling task.

7.3 Task Synchronization and Communication

7.3.1 Counting Semaphore

Semaphores are a mechanism for synchronizing the execution of tasks. The semaphores implemented in the μSystem are counting semaphores as described by Dijkstra [Dij68]. A counting semaphore has two parts: a counter and a list of waiting tasks. The counter is accessible to users, while the list of waiting tasks is managed by the μKernel.

uSemaphore is the type of a semaphore and it must be initialized before it is used; appropriate count values are integer values ≥ 0. To initialize a semaphore variable, the macro U_SEMAPHORE is used, as in:

```
uSemaphore x = U_SEMAPHORE(0), y = U_SEMAPHORE(1), z = U_SEMAPHORE(4);
```

This declares three variables that are semaphores and initializes them to the value 0, 1, and 4, respectively. The macro can be used at execution time to initialize a declared semaphore or initialize a dynamically allocated one, as in:

```
uSemaphore x = U_SEMAPHORE(0), y, *z;
...
y  = U_SEMAPHORE(1);
z  = uMalloc( sizeof(uSemaphore) );
*z = U_SEMAPHORE(4);
```

(The routine uMalloc is provided by the μSystem and detailed below. uMalloc returns void * and so it is unnecessary to cast its result to uSemaphore * before assigning to z.) Normally, a semaphore is only initialized *once*; any further modification to the semaphore is done *only* by routines uP and uV. However, if a semaphore is re-initialized, it should have no tasks waiting on it. This is because initialization is done by assignment, and hence, there is no way to generate an error or unblock the waiting tasks. Any waiting tasks will remain blocked and be inaccessible.

The routines uP and uV are used to perform the classical counting semaphore operations. uP decrements the semaphore counter if the value of the semaphore is greater than zero; otherwise, the calling task blocks. uV wakes up the task blocked for the longest time if there are tasks blocked on the semaphore; otherwise, the semaphore counter is incremented.

uP(*semaphore-address*);
uV(*semaphore-address*);

semaphore-address is the address of a uSemaphore variable which is modified by uP or uV. Unless the argument is already a pointer to a uSemaphore, it must be preceded by an &.

The routine uC returns the current value of a semaphore's counter.

counter = uC(*semaphore-address*);

counter is the value of the semaphore's counter

semaphore-address is the address of a uSemaphore variable. Unless the argument is already a pointer to a uSemaphore, it must be preceded by an &.

If the counter is positive, that indicates the number of uP operations that can occur before a task blocks. If the counter is zero or negative, then the absolute value of the counter value is the number of blocked tasks waiting on the semaphore.

Appendix B contains a complete P/V program.

7.3.2 Send/Receive/Reply

Message passing is used for synchronizing tasks and passing data between them. The two tasks involved in a communication are called the sender and receiver tasks. What characterizes send/receive/reply is that the sender blocks (i.e. does not continue execution) until the receiver receives the message and explicitly replies. All sends must be replied to, but the receiver does not need to reply to messages in the order that they were received. The following routines perform send/receive/reply communication between tasks and are largely derived from Thoth [Che82].

The sender takes on one of two states during a communication:

send-blocked which means the sender has done a send but the message has not been received.

reply-blocked which means the sender's message has been received but a reply has not been performed.

The receiver can be in the following state during a communication:

receive-blocked which means the receiver has done a receive but no message has been sent.

The routine uSend is used to transmit a message to another task. uSend blocks until the receiver has replied to the sent message.

replier-task-id = `uSend(` *receiver-task-id*, *reply-area*, *reply-area-length*, *send-message*, *send-message-length* `)`

replier-task-id is the uTask identifier of the task that replied to this send. Because of the ability to forward a message (detailed below), the replying task is not necessarily the same as the task sent to.

receiver-task-id is the uTask identifier of the receiving task.

reply-area is the address of a reply area into which the reply message of the receiving task will be copied.

reply-area-length is the size in bytes of the reply area.

send-message is the address of a message to be sent to the receiving task.

send-message-length is the size in bytes of the message to be sent.

Send transmits the argument *send-message* to the receiving task's *receive-area*.

The routine uReceive is used to receive a message sent from another task. uReceive receives a message sent to it from any task; it cannot be used to receive a message from a particular task. uReceive blocks if there is no task currently sending to it.

sender-task-id = `uReceive(` *receive-area*, *receive-area-length* `)`

sender-task-id is the uTask identifier of the sending task.

receive-area is the address of a receive area into which the sent message of the sending task will be copied.

receive-area-length is the size in bytes of the receive area.

When a message arrives, data from the sender's *send-message* argument is copied into the receiver's *receive-area* argument. After a task has received a message, it is obligated to reply to the sender task or to delegate the reply responsibility to another task by forwarding.

The routine uReply is used to reply to another task and to transmit a message back to the sender. uReply does not block.

`uReply(` *sender-task-id*, *reply-message*, *reply-message-length* `)`

sender-task-id is the uTask identifier of a task that has sent a message to this task. The reply will fail if the specified sender task did not send a message to the replying task, or the replying task has already replied to this message from that sender.

reply-message is the address of a reply message to be sent back to the sender.

reply-message-length is the length of the reply message to be sent back to the sender.

The reply copies the argument *reply-message* back to the sending task's *reply-area* argument and the sending task is then unblocked and continues execution.

The routine `uForward` is used to transmit a message to another task on behalf of the task that originally sent the message. Once a message is forwarded, only the new receiving task can reply to it, unless the new receiving task forwards the message again. `uForward` blocks until the new receiver has received the forwarded message; no reply is necessary to the forwarder of a message, nor can a receiving task determine if a message was forwarded or sent by the original sender.

`uForward(` *forward-task-id*, *send-message*, *send-message-length*, *sender-task-id* `)`

forward-task-id is the `uTask` identifier of the task to which the message is forwarded.

send-message is the address of a message that is to be forwarded.

send-message-length is the length of the message to be forwarded.

sender-task-id is the `uTask` identifier of the original message sender. The forward will fail if the specified sender task did not send a message to the forwarding task.

There is no obligation on the part of the forwarder to forward the same message that it originally received from a sender. The forwarder can receive a message and forward a new message to another task on behalf of the original sender. The receiving task will service this new message and reply to the original sender or it can perform another forward.

Appendix C contains a complete message passing program.

7.4 Task Termination

A task is terminated by executing the `uDie` routine. This routine can be invoked at any level of nested coroutine or subroutine invocation to terminate a task. `uDie` is used in conjunction with routine `uAbsorb`, which allows a task to wait for the completion of another task. The pair of routines allows a result to be passed from the terminating task back to the task waiting for its completion.

The routine `uAbsorb` waits for completion of a specified task, accepts its last result sent by `uDie`, and deallocates its resources.

`uAbsorb(` *task-id*, *reply-area*, *reply-area-length* `);`

task-id is the `uTask` identifier of the completing task.

reply-area is the address of the reply area into which the message sent from `uDie` will be copied.

reply-area-length is the size in bytes of the reply area.

The routine `uDie` terminates execution of a task and passes back a result to some task awaiting its completion using `uAbsorb`.

`uDie(` *send-message*, *send-message-length* `);`

send-message is the address of the message to be sent to a task waiting for termination of this task.

send-message-length is the size in bytes of the message to be sent.

Each task terminated using `uDie` must be absorbed by only one task. If the terminating task is not absorbed, its resources will not be recovered. If multiple tasks absorb a task, only one will be successful and continue execution. The other absorbing tasks will block forever. Currently, there is no mechanism to explicitly unblock such a task (or any tasks that are blocked on it) or a timeout facility that can be specified on `uAbsorb` to implicitly unblock it.

The following shows how `uAbsorb` and `uDie` can be used to return a result from a task:

```
void f(float x, float y) {
    float result;
    ...                             /* calculate result concurrently with emitter */
    uDie(&result, sizeof(result)); /* terminate task and return result */
}

uMain() {
    float result, a, b;
    uTask tid;
    ...
    tid = uEmit(f, a, b);      /* start a task running f concurrently */
    ...                        /* continue concurrently with f */
    uAbsorb(tid, &result, sizeof(result)); /* wait for task's completion and result */
}
```

Executing a `return` statement in an emitted routine is the same as the routine call `uDie(U_NULL, 0)`. This causes the task to terminate and wait to be absorbed. The same action occurs if control runs off the end of the emitted routine. Therefore, if a value is to be returned at task termination, it must be passed back with an explicit call to `uDie`.

8. Virtual Processor and Cluster Facilities

A cluster is a collection of μSystem tasks and virtual processors; it provides a runtime environment for their execution. This environment contains a number of variables that can be modified to affect how tasks and virtual processors behave in the cluster.

The creation of a cluster allocates a data structure to store the values of the cluster environment variables and a list of virtual processors that are associated with the cluster. A number of routines are available to modify the cluster environment variables, and to add and remove virtual processors. The address of the cluster data-structure acts as the cluster reference. A cluster reference can be used in operations like `uLongEmit` to create a task on a particular cluster.

To ensure maximum concurrency, it is desirable that a task does not execute an operation that will cause the virtual processor it is executing on to block. It is also essential that all virtual processors in a cluster only execute on hardware processors that can execute any task in that cluster, since task execution is distributed across all virtual processors of a cluster. When tasks or virtual processors cannot satisfy these conditions, it is essential that such tasks or virtual processors be grouped into a separate cluster in order to avoid adversely affecting other tasks. Each of these points will be examined.

For each virtual processor that blocks, the potential for concurrency decreases; therefore, it is better to have a separate cluster that contains a task that performs a blocking operation on a separate virtual processor. This maintains a constant number of virtual processors for concurrent computation in a computational cluster. Computational tasks can then communicate with the tasks that execute blocking operations in the separate cluster without causing any of the virtual processors in the computational cluster to block. In most versions of UNIX, all I/O operations cause the UNIX process to block, and therefore, all I/O in the μSystem is delegated to tasks on separate clusters. The relationship between a computational and an I/O cluster is illustrated in Figure 2. Depending on the kind of I/O, there may be one or several tasks on the I/O cluster. To simplify the complexity of cluster creation for I/O operations, the μSystem supplies a library of I/O operations that perform the cluster creation automatically (detailed below).

On some multiprocessor computers, all hardware processors are not equal. For example, all of the hardware processors may not have the same floating-point units; some units may be faster than others. Therefore, it may be necessary to create a cluster of virtual processors that are attached to these specific hardware processors. (The mechanism for attaching virtual processors onto hardware processors is operating system specific and not part of the μSystem.) All tasks that need to perform high-speed floating-point operations can be created on this cluster. This still allows tasks that do only fixed-point calculations to continue on another cluster, potentially increasing concurrency, but not interfering with the floating-point calculations.

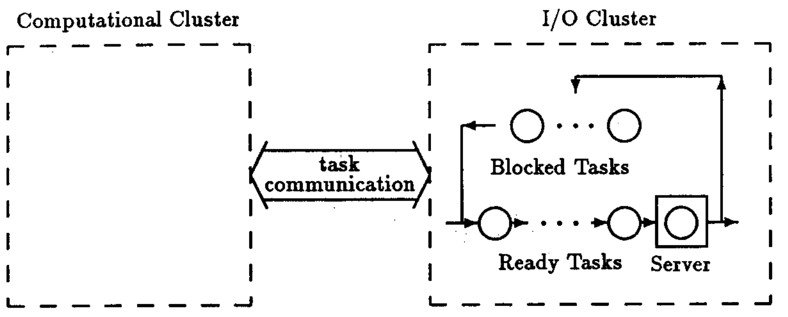


Figure 2: Multiple Clusters

8.1 Cluster Variables

Each cluster has a number of environment variables that are used implicitly by tasks and virtual processors associated with that cluster (see Figure 3):

stack size is the default stack size used when coroutines or tasks are created with `uCocall` or `uEmit` on a cluster.

argument length is the default argument length used when coroutines or task are created with `uCocall` or `uEmit` on a cluster.

number of virtual processors is the number of virtual processors currently allocated on a cluster.

time slice duration is the interrupt duration for all virtual processors on a cluster.

spin duration is the spin duration before an idle virtual processor sleeps for all virtual processors on a cluster.

Each of these variables is either explicitly set or implicitly assigned a system-wide machine-dependent default value when the cluster is created. The mechanisms to read and reset the values are detailed below.

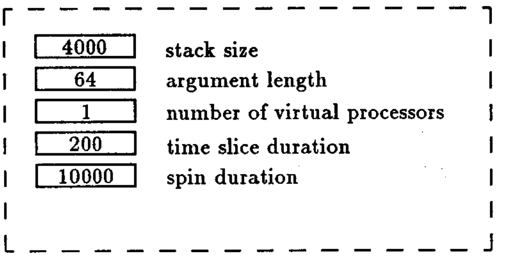


Figure 3: Cluster Variables

8.1.1 Default Stack Size

The routine `uSetStackSize` is used to set the default stack size value for a cluster.

`uSetStackSize(` *new-stack-size* `);`

new-stack-size is an integer value representing the number of bytes that is used as default stack size.

For example, the call `uSetStackSize(8000)` sets the default stack size to 8000 bytes.

The routine `uGetStackSize` is used to read the value of the default stack size for a cluster.

default-stack-size = `uGetStackSize();`

default-stack-size is an integer value that is the current default stack size value.

For example, the call `i = uGetStackSize()` sets integer `i` to the value 8000.

The μSystem provides the routine `uVerify` to verify whether or not the current coroutine or task has overflowed its stack. If it has, a μSystem error results and is handled by the μSystem's error handling mechanisms. When debugging is enabled, `uVerify` is called on each context switch. Since a coroutine or task often calls no other subroutines, it is suggested that a call to `uVerify` be included at the beginning of each, as in the following example:

```
void f( ... ) {
    ... declarations
    uVerify();
    ... routine body
}
```

Thus, after each coroutine or task has allocated its own local stack space, a verification is made that the stack has not overflowed. If a coroutine or task calls subroutines, each subroutine would have to start with a call to `uVerify` to check for stack overflow.

8.1.2 Default Argument Length

The routine `uSetArgLen` is used to set the default argument length for a cluster.

`uSetArgLen(` *new-argument-length* `);`

new-argument-length is an integer value representing the number of bytes that is used as the default argument length.

For example, the call `uSetArgLen(100)` sets the default argument length to 100 bytes.

The routine `uGetArgLen` is used to read the value of the default argument length for a cluster.

default-argument-length = `uGetArgLen();`

default-argument-length is an integer value that is the current default argument length.

For example, the call `i = uGetArgLen()` sets integer `i` to the value 100.

In theory, it is possible to determine the argument length automatically from the argument list; however, this is only possible if the facilities to start a coroutine or task are integrated into the programming language. There are problems when an argument length is specified that is not the exact size of the argument list. If the length is greater, extra bytes are copied, which is runtime inefficient. If the length is less, argument information is not copied, which results in unpredictable behaviour or failure of the coroutine or task.

8.1.3 Virtual Processors on a Cluster

The routine `uSetProcessors` will create or destroy virtual processors as needed to have the specified number of processors on the current cluster.

`uSetProcessors(` *number-of-processors* `);`

number-of-processors is the number of virtual processors that will exist on the current cluster.

For example, the call `uSetProcessors(5)` will increase or decrease the number of virtual processors on a cluster to 5.

The routine `uGetProcessors` is used to read the current number of processors on a cluster.

current-number-of-processors = `uGetProcessors();`

current-number-of-processors is an integer value that is the current number of virtual processors on this cluster

For example, the call `i = uGetProcessors()` sets integer `i` to the value 5.

The system dependent macro `U_PHYSICAL_PROCESSORS` returns the maximum number of hardware processors available on a computer.

Changing the number of virtual processors is expensive, since a request is made to UNIX to allocate or deallocate UNIX processes. This operation often takes at least an order of magnitude more time than task creation. Further, there is often a small maximum number of UNIX processes (e.g. 20–40) that can be created for a UNIX program. Therefore, virtual processors should be created judiciously, normally at the beginning of a program.

The following are points to consider when deciding how may virtual processors to create for a cluster. First, there is no advantage in creating significantly more virtual processors than the average number of simultaneously active tasks on the cluster. For example, if on average three tasks are eligible for simultaneous execution, then creating significantly more than three virtual processors will not achieve any execution speed up and wastes resources. Second, while it is possible to create more virtual processors than actual hardware processors, there is usually a performance decrease in doing so. Having more virtual processors than actual processors can result in extra context switching of the heavy-weight UNIX processes, which is runtime inefficient. This same problem can occur between clusters. If a computational problem is broken into multiple clusters and the total number of virtual processors associated with all the clusters exceeds the number of hardware processors, extra context switching of the UNIX processes will occur. The exception to this rule is when multiple clusters are used to handle blocking I/O problems. In this case, the virtual processors associated with I/O clusters spend most of their time blocked and do not interfere with virtual processors on computational clusters. Finally, a μSystem program usually shares the actual hardware processors with other user programs. Therefore, the overall UNIX system load will affect how many virtual processors should be allocated to avoid unnecessary context switching of UNIX processes.

8.1.4 Implicit Task Scheduling

Pre-emptive scheduling is enabled by default on both unikernel and multikernel. Each virtual processor in the μSystem is periodically interrupted by a UNIX timer in order to reschedule the currently executing task. Note that interrupts are not associated with a task but with a virtual processor; hence, tasks do not receive a time slice, virtual processors do. A task is pre-empted at non-deterministic locations in its execution when the virtual processor's time-slice expires. All virtual processors on a cluster have the same interrupt duration but the interrupts are not synchronized. The default virtual-processor time-slice is machine dependent but is approximately 0.1 seconds on most machines. The effect of this pre-emptive scheduling ensures that users do not write programs that depend on the order or the speed of execution of any particular task or tasks in their program. Further, on the unikernel, the effect is to accurately simulate parallelism.

The routine `uSetTimeSlice` is used to set the default interrupt-duration for each virtual processor on the current cluster.

`uSetTimeSlice(` *milliseconds* `);`

milliseconds is an integer value representing the number of milliseconds between interrupts.

For example, the call `uSetTimeSlice(50)` sets the default interrupt-duration to 0.05 seconds for each virtual processor on this cluster. To turn pre-emption off, call `uSetTimeSlice(0)`. On many machines the minimum time slice duration may be 10 milliseconds (0.01 of a second). Setting the duration to an amount less than this simply sets the interrupt time interval to this minimum value. (On System V UNIX, pre-emption occurs at most once a second, which may not be often enough to adequately test that a concurrent program does not depend on order or speed of execution of its tasks.)

The overhead of pre-emptive scheduling depends on the frequency of the interrupts. Further, because interrupts involve entering the UNIX kernel, they are relatively expensive. We have found that an interrupt interval of 0.05 to 0.1 seconds adequately verifies that a concurrent program does not depend on order or speed of task execution and increases execution cost by less than 1% for most programs.

The routine `uGetTimeSlice` is used to read the current default interrupt-duration for a cluster.

```
time-slice-duration = uGetTimeSlice();
```

time-slice-duration is an integer value that is the current interrupt duration on this cluster.

For example, the call `i = uGetTimeSlice()` sets integer `i` to the value 50.

8.1.5 Idle Virtual Processors

When there are no tasks on a cluster ready queue for a virtual processor to execute, the idle virtual processor has to spin in a loop or sleep or both. In the μKernel, an idle virtual processor spins for a user specified amount of time, before it sleeps. During the spinning, the virtual processor is constantly checking the ready queue for the arrival of new work. An idle virtual processor is ultimately put to sleep so that machine resources are not wasted. The reason that the idle virtual processor spins is because the sleep/wakeup cycle can be large in comparison to the execution of tasks in a particular application. If an idle virtual processor goes to sleep immediately upon finding no work on the ready queue, then the next executable task will have to wait for completion of a UNIX system call to restart the virtual processor. Alternatively, if the idle processor spins for a short period of time any task that arrives during the spin duration will be processed immediately. Selecting a spin time is application dependent and it can have a significant affect on performance.

The routine `uSetSpin` is used to set the default spin-duration for each virtual processor on the current cluster.

```
uSetSpin( microseconds );
```

microseconds is an integer value representing the number of microseconds the idle virtual processor will spin before it sleeps.

For example, the call `uSetSpin(50000)` sets the default spin-duration to 0.05 seconds for each virtual processor on this cluster. To turn spinning off, call `uSetSpin(0)`.

The routine `uGetSpin` is used to read the current default spin-duration for a cluster.

```
spin-duration = uGetSpin();
```

spin-duration is an integer value that is the current spin duration on this cluster.

For example, the call `i = uGetSpin()` sets integer `i` to the value 50000. The precision of the spin time is machine dependent and can vary from 1 to 50 microseconds.

8.2 Cluster Type

`uCluster` is the type of a cluster identifier, as in:

```
uCluster x, y, z;
```

which creates three variables that contain cluster identifier values.

8.3 Cluster Creation

`uClusterVars` is the type of a structure that contains the initial defaults for a new cluster created by `uLongCreateCluster` (detailed next):

```
typedef struct {
    long Processors;
    long TimeSlice;
    long Spin;
    long StackSize;
    long ArgLen;
} uClusterVars;
```

uClusterVars variables must be initialized with the macro U_CLUSTER_VARS() before use, as in:

```
uClusterVars cv = U_CLUSTER_VARS();
```

This initializes the fields of cv to the system-wide machine-dependent default values. Individual fields can then be changed to user specified values. By always initializing uClusterVars variables with macro U_CLUSTER_VARS(), new cluster variables can be added in the future and programs do not have to be changed, only re-compiled.

The routine uLongCreateCluster creates a cluster with at least one virtual processor associated with it.

cluster-id = uLongCreateCluster(*cluster-variable-address*);

cluster-id is the uCluster identifier of the new cluster. This value must be retained as it is used to subsequently place tasks on the cluster, or to destroy the cluster.

cluster-variable-address is the address of a uClusterVars variable which contains the initial defaults for the new cluster.

The maximum number of clusters that can be created is indirectly limited by the number of UNIX processes a program can create, as the sum of the virtual processors on all clusters cannot exceed the limit set by UNIX for a program.

The following shows how a cluster is created with 5 processors, no time slicing, a stack size default of 8000 bytes, and the machine-dependent default for the task argument-length and virtual-processor spin-time.

```
uClusterVars cv = U_CLUSTER_VARS(); /* set machine-specific defaults */
uCluster c;

cv.Processors = 5;                    /* change specific fields */
cv.TimeSlice = 0;
cv.StackSize = 8000;

c = uLongCreateCluster( &cv );
```

There is a short form of uLongCreateCluster, called uCreateCluster, that assumes the machine specific defaults for all cluster variables except number of processors and virtual-processor time-slice.

cluster-id = uCreateCluster(*number-of-processors*, *milliseconds*);

cluster-id is the uCluster identifier of the new cluster. This value must be retained as it is used to subsequently place tasks on the cluster, or to destroy the cluster.

number-of-processors is the number of virtual processors that will be initially associated with the new cluster.

milliseconds is an integer value representing the number of milliseconds between interrupts for each virtual processor on the cluster.

The routine uThisCluster is used to determine which cluster a task currently resides in.

cluster-id = uThisCluster();

cluster-id is the uCluster identifier of the cluster on which the calling task resides.

8.4 Cluster Termination

The routine `uDestroyCluster` deallocates the specified cluster, which destroys all virtual processors associated with the cluster.

`uDestroyCluster(` *cluster-id* `);`

cluster-id is a `uCluster` identifier of the cluster to be destroyed.

It is the user's responsibility to ensure that no tasks are executing on a cluster when it is destroyed; therefore, a cluster can only be destroyed from a task on another cluster. If tasks are executing on a cluster when it is destroyed, they will block and be inaccessible.

8.5 Explicit Task scheduling

The routine `uYield` gives up control of the virtual processor to another ready task. For example, the routine call `uYield()` returns control to the μKernel to schedule another task, hence giving up control of the virtual processor. If there are no other ready tasks, the yielding task will be restarted.

`uYield` allows a task to relinquish control when it has no further work to do or when it wants other tasks to execute before it performs more work. An example of the former situation is when a task is polling for an event, such as a hardware event. After the polling task has determined the event has not occurred, it can relinquish control to another ready task. An example of the latter situation is when a task is creating other tasks. The creating task may not want to create a large number of tasks before the created tasks have a chance to begin execution. (Task creation occurs so quickly that it is possible to create 30-50 tasks before pre-emption occurs.) If after the creation of several tasks the creator yields control, then each created task will have an opportunity to begin execution (possibly only one instruction before pre-emption occurs) before the next group of tasks is created. This facility is not a mechanism to control the exact order of execution of tasks like resume does with coroutines; pre-emptive scheduling and/or multiple processors make this impossible.

The routine `uDelay` invokes `uYield` N times. For example, the routine call `uDelay(5)` calls `uYield()` 5 times, hence immediately giving up control of the virtual processor and ignoring the next 4 times the task is scheduled for execution.

8.6 Defaults for uMain

Because all the defaults are set for the initial user cluster *before* `uMain` begins execution, the task `uMain` is normally created with the machine-dependent cluster-values. This can cause problems if the default stack size is insufficient for the variables declared in `uMain`. If the default stack size for `uMain` is exceeded, `uMain` will terminate with an addressing error on the first reference to a local variable beyond the stack. It is possible to reset any of the defaults for the initial user cluster by defining an optional routine `uStart` in the user application and calling the above routines to change the defaults, for example:

```
void uStart( void ) {
    if ( uGetStackSize() < 8000 ) {   /* check machine dependent stack size */
        uSetStackSize( 8000 );        /* set stack size to at least 8000 bytes */
    }
    uSetTimeSlice( 0 );               /* turn off time slicing before uMain begins */
}
```

`uStart` is called by the μKernel to set up the user cluster before it emits `uMain`. If the user does not supply a `uStart` routine, a default routine with a null execution body is supplied.

8.7 Migration

Most tasks will execute on only one cluster. However, some applications may need to move a task from one cluster to another so that it can access resources that are peculiar to that cluster's virtual processors. The routine `uMigrate` moves a specified task to a specified cluster.

`uMigrate(` *task-id*, *cluster-id* `)`

task-id is a `uTask` identifier of the task to be moved.

cluster-id is a `uCluster` identifier of the cluster that the task is moved to.

9. Memory Management

All data that the μSystem manipulates must reside in shared memory. In the unikernel case, there is a single data address-space. All memory allocated during execution comes from this address space, and hence, is shared. In the multikernel case, several data address-spaces exist, one for each UNIX process. These data address-spaces have private memory accessible only by a single process and shared memory that is accessible by all the UNIX processes.

In order to make memory management operations portable across both versions of the μSystem and guarantee that storage is sharable, the μSystem provides memory management routines, called `uMalloc`, `uRealloc` and `uFree`, that will allocate and free memory correctly for each version of the μSystem. These routines provide identical functionality to the UNIX `malloc`, `realloc` and `free` routines.

9.1 Memory Allocation

The routine `uMalloc` allocates memory.

address = `uMalloc(` *number-of-bytes* `);`

address the address of a block of memory of at least the requested size. `uMalloc` returns `void *` and so it is unnecessary to cast its result to the pointer type expected at the usage site.

number-of-bytes the number of bytes of memory to be allocated.

If `uMalloc` cannot allocate the requested memory, an error is reported via the μSystem error handling facilities.

The following code shows an example of how to allocate memory.

```
int size;
void *addr;
...
addr = uMalloc( size );
```

The routine `uRealloc` increases or decreases the size of an existing allocated block of memory or moves the block to a new location that is at least of the specified size.

address = `uRealloc(` *allocated-memory-address*, *number-of-bytes* `);`

address the address of a block of memory of at least the requested size. `uRealloc` returns `void *` and so it is unnecessary to cast its result to the pointer type expected at the usage site.

allocated-memory-address the address of an existing allocated area of memory.

number-of-bytes the number of bytes that the old allocated area is to be re-sized to.

If `uRealloc` cannot allocate the requested memory, an error is reported via the μSystem error handling facilities.

9.2 Memory Deallocation

The routine uFree deallocates memory.

```
uFree( address );
```

address of the block of memory to deallocate.

The following code shows how to deallocate memory.

```
void *addr;
...
uFree( addr );
```

10. Interaction with the UNIX File System

In UNIX, file and socket operations cause the UNIX process performing the operation to block. This defeats concurrency in the unikernel and inhibits concurrency in the multiprocessor. Different techniques are used to mitigate this problem in the unikernel and multikernel. In both cases, cover routines to perform the I/O operations should be used. The I/O cover routines have essentially the same syntax as the normal UNIX I/O routines; however, instead of a UNIX file descriptor being passed around as the reference to a file or socket, a μSystem file descriptor is used. None of the μSystem I/O routines return an error code; errors are checked for and handled through an internal mechanism in the μSystem (detailed below).

10.1 Unikernel File Operations

UNIX supports non-blocking I/O operation; however, not all UNIX systems support a signalling mechanism to indicate completion of the I/O operation. In general, it is necessary to poll for completion of non-blocking I/O operations.

To retain concurrency in the unikernel during I/O operations, the μSystem I/O routines check the ready queue before performing their corresponding UNIX I/O operation. If there are no tasks waiting to execute, a blocking I/O operation is performed. If there are tasks to execute, a nonblocking I/O operation is performed. The task performing the I/O operation then goes into a polling loop checking for completion of the I/O operation and yielding control of the processor if the operation has not completed. This allows other tasks to progress with a slight degradation in performance due to the polling tasks.

If all ready tasks are performing I/O operations, then these tasks spin checking for I/O completion, which wastes processor resources. If one or more of the I/O operations are to disk, then the spin time is relatively low (e.g. tens of milliseconds). Only if all the I/O operations are to terminal like devices will the spin time be a potential problem; however, we believe that multiple terminal input operations are rare. Hence, this solution is a compromise between retaining concurrency and not wasting processor resources given the lack of a signalling facility to indicate I/O completion. In general, it works sufficiently well to accurately test programs performing concurrent I/O operations on a uniprocessor.

10.2 Multikernel File Operations

In the multikernel, not only is there the blocking I/O problem, but some UNIX systems associate the internal information needed to access a file (i.e. a file descriptor) with a virtual processor (i.e. UNIX process) in a non-shared way. This means that if a task opens a file on one virtual processor it will not be able to read or write the file if the task is scheduled for execution on another virtual processor. Both problems can be solved by creating a separate cluster that has a single virtual processor containing the file descriptor. Any task that wants to access the file migrates to the I/O cluster to perform the operation. In this manner, a task performing an I/O operation can access the private UNIX file descriptor, and the blocking I/O operations do not affect virtual processors of a computational cluster only the task that called the μSystem cover routine.

In detail, a cluster and a virtual processor are automatically created when a user task opens a file. Hence, each open file has a corresponding UNIX process. The exception to this rule is `stdin`, `stdout` and `stderr` which are all open implicitly on the system cluster. A user task then performs I/O operations by executing the equivalent μSystem cover routine, which migrates the task to the cluster containing the file descriptor and performs the appropriate operation. When the user task closes the file, the cluster and all of its resources are released. To ensure that multiple tasks are not simultaneously performing I/O operations, each μSystem file descriptor has a semaphore that is used to serialize operations. Figure 4 illustrates the runtime structures created for accessing a file (UNIX resources are illustrated with an oval).

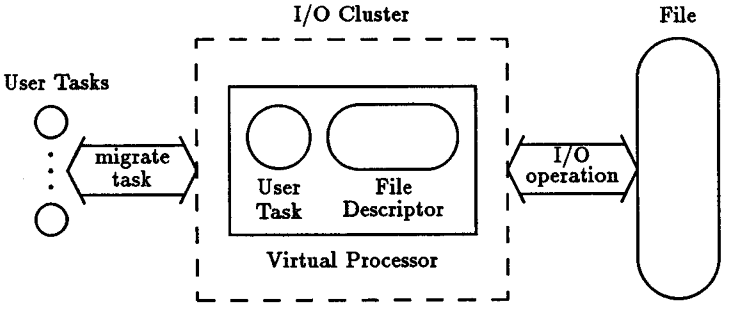


Figure 4: UNIX File I/O Cluster

Notice that serialization only occurs per file descriptor. If a file is opened multiple times, each opening creates a new and independent cluster, virtual processor and file descriptor. Access to these file descriptors on different clusters are not synchronized. This is not a problem if all tasks are reading, but will not work, in general, for multiple writer tasks or a combination of reader and writer tasks to the same file.

11. Formatted I/O

To aid the programmer, there are cover routines for the UNIX formatted I/O operations, which work like their UNIX counterpart, but perform their operations on a separate cluster. In the unikernel case, creating a cluster returns a reference to the current cluster so all files are open on the virtual processor for this single cluster. As well, there are three file identifiers `uStdout`, `uStdin`, and `uStderr` which identify the file descriptors managing the corresponding files `stdout`, `stdin`, and `stderr`, respectively. For complete details on each cover routine, first refer to the man pages for the corresponding UNIX routine. None of the μSystem formatted I/O routines returns an error code, as errors are handled in a different way (detailed below).

11.1 Stream Type

`uStream` is the type of a formatted file identifier, as in:

```
uStream input, output;
```

which creates two variables that contain formatted file identifier values.

11.2 Opening a Formatted File

A formatted file is opened with the uFopen routine.

file-id = uFopen(*unix-file-name*, *open-type*);

file-id is a `uStream` identifier to the formatted file. This value must be retained as it is used to subsequently access the file.

unix-file-name is the address of a string of ASCII characters representing a UNIX path name to the file, terminated by a null character.

open-flag indicates how the file is to be opened for access. See the man entry for `fopen` for the options.

uFopen creates a cluster and places one virtual processor on that cluster. Then uFopen migrates the calling task to the new cluster, which performs an actual UNIX open operation, creating the UNIX file descriptor on the virtual processor for that cluster. The calling task is then migrated back to its original cluster. The following example shows the opening of a formatted file:

```
uStream input;
...
input = uFopen( "test.c", "r" );
```

11.3 Reading and Writing from a Formatted File

The routine uPrintf converts, formats, and prints its arguments on the standard output file which is usually the interactive terminal.

uPrintf(*format*, *arguments ...*);

format is a format string containing text to be printed and format codes which describe how to print the following variable number of arguments.

arguments ... is a list of arguments to be formatted and printed on standard output. The number of elements in this list must match with the number of format codes.

The following example shows printing on the standard output file:

```
uPrintf( "Hello World\n" );
```

The routine uPutc writes a character on the specified output file (identical to uFputc below).

character = uPutc(*character*, *file-id*);

character the character that is written is returned.

character the character to be appended to the end of the file denoted by the specified *file-id*.

file-id is a uStream identifier to the formatted file.

The following example shows printing a character on an arbitrary output file:

```
int ch;
uStream output;
...
ch = uPutc( 'c', output );
```

The routine uPutchar writes a character on the standard output file which is usually the interactive terminal).

character = uPutchar(*character*);

character the character that is written is returned.

character the character to be appended to the end of the standard output file.

The following example shows printing on the standard output file:

```
uPutchar( 'c' );
```

The routine uPuts writes a character string on the standard output file which is usually the interactive terminal.

uPuts(*character-string*);

character-string the string of characters terminated with `'\0'` to be appended to the end of the standard output file.

The following example shows printing on the standard output file:

```
uPuts( "abc" );
```

The routine `uFprintf` performs the same operation as `uPrintf` but does not default to printing on standard output. Instead, it can print formatted output on any specified file.

`uFprintf(` *file-id*, *format*, *arguments ...* `);`

file-id is a `uStream` identifier to the formatted file.

format is a format string containing text to be printed and format codes which describe how to print the following variable number of arguments.

arguments ... is a list of arguments to be formatted and printed on the file denoted by the specified *file-id*. The number of elements in this list must match with the number of format codes.

The following example shows printing on an arbitrary output file:

```
uStream output;
...
uFprintf( output, "This is the number %d\n", 1 );
```

The routine `uFputc` writes a character on the specified output file (identical to `uPutc` above).

character = `Fputc(` *character*, *file-id* `);`

character the character that is written is returned.

character the character to be appended to the end of the file denoted by the specified *file-id*.

file-id is a `uStream` identifier to the formatted file.

The following example shows printing a character on an arbitrary output file:

```
uStream output;
...
uFputc( 'c', output );
```

The routine `uFputs` performs the same operation as `uPuts` but does not default to printing on standard output.

`uFputs(` *character-string*, *file-id* `);`

character-string the string of characters terminated with `'\0'` to be appended to the end of the file denoted by the specified *file-id*.

file-id is a `uStream` identifier to the formatted file.

The following example shows printing a character string on an arbitrary output file:

```
uStream output;
...
uFputs( "abc", output );
```

Unfortunately, when writing to a terminal the occasional carriage return is lost due to a bug in UNIX. If output is directed into a file (e.g. > or >>) or through a filer (|), there is no problem.

`a.out | more` *output from μSystem program piped into* `more` *ensures no loss of carriage returns*

The routine `uScanf` reads characters from the standard input file, interprets then according to the specified format codes, and stores the result in the arguments.

number-of-characters = `uScanf(` *format*, *arguments ...* `);`

number-of-characters is the number of successfully matched and assigned input items.

format is a format string containing text to be matched and format codes which describe how to interpret input text for assignment to the following variable number of arguments.

arguments ... is a list of pointers to variables that are assigned the interpreted text values from standard input. The number of elements in this list must match with the number of format codes.

The following shows scanning input from the standard input file:

```
int a, b;
...
uScanf( "%d %d", &a, &b );
```

The routine `uGetc` reads a character from the specified input file (identical to `uFgetc` below).

integer-value = `uGetc(` *file-id* `);`

integer-value the next character, returned as an integer, to read from the file denoted by the specified *file-id.*

file-id is a `uStream` identifier to the formatted file.

The following example shows reading a character from an arbitrary output file:

```
int ch;
uStream input;
...
ch = uGetc( input );
```

The routine `uGetchar` reads a character from the standard input file which is usually the interactive terminal.

integer-value = `uGetchar();`

integer-value the next character, returned as an integer, from the standard input file.

The following example shows reading from the standard input file:

```
int ch;

ch = uGetchar();
```

The routine `uGets` reads n-1 characters, or up to a newline, from the standard input file into a string area.

string-pointer = `uGets(` *string-pointer*, *number-of-characters* `);`

string-pointer the value of the first argument.

string-pointer a pointer to a string area into which characters are read from the standard input file. The string is terminated by a newline character.

number-of-characters the maximum number of characters to be read into the string area plus the newline character.

The following example shows reading from the standard input file:

```
char *s;
...
s = uGets( s, 21 );
```

The routine `uUngetc` pushes the specified character onto the specified input file so that it can be read as if it appeared in the input.

character = `uUngetc(` *character*, *file-id* `);`

character the character that is pushed is returned.

character the character to be pushed back onto the file denoted by the specified *file-id*.

file-id is a `uStream` identifier to the formatted file.

The following example shows pushing a character back onto an arbitrary output file:

```
int ch;
uStream input;
...
ch = uUngetc( ch, input );
```

The routine `uFscanf` performs the same operation as `uScanf` but does not default to reading from standard input. Instead, it can read from any specified file.

number-of-characters = `uFscanf(` *file-id*, *format*, *arguments ...* `);`

number-of-characters is the number of successfully matched and assigned input items.

file-id is a `uStream` identifier to the formatted file.

format is a format string containing text to be matched and format codes which describe how to interpret input text for assignment to the following variable number of arguments.

arguments ... is a list of pointers to variables that are assigned the interpreted text values from standard input. The number of elements in this list must match with the number of format codes.

The following shows scanning input from an arbitrary input file:

```
int a, b;
uStream input;
...
uFscanf( input, "%d %d", &a, &b );
```

The routine `uFgetc` reads a character from the specified input file (identical to `uGetc` above).

integer-value = `uFgetc(` *file-id* `);`

integer-value the next character, returned as an integer, to read from the file denoted by the specified *file-id*.

file-id is a `uStream` identifier to the formatted file.

The following example shows reading a character from an arbitrary output file:

```
int ch;
uStream input;
...
ch = uFgetc( input );
```

The routine **uFgets** performs the same operation as **uGets** but does not default to reading from standard input.

string-pointer = **uFgets(** *string-pointer*, *number-of-characters*, *file-id* **);**

string-pointer the value of the first argument.

string-pointer a pointer to a string area into which characters are read from the file denoted by the specified *file-id*. The string is terminated by a newline character.

number-of-characters the maximum number of characters to be read into the string area plus the newline character.

file-id is a **uStream** identifier to the formatted file.

The following example shows reading a character from an arbitrary output file:

```
char *s;
uStream input;
...
s = uFgets( s, 21, input );
```

11.4 Flushing a Formatted File

It may be necessary to flush the output buffer to a file to insure that all output is written before the program continues. A file buffer is flushed with the `uFflush` routine.

`uFflush(` *file-id* `);`

file-id is a `uStream` identifier to the formatted file.

The following shows how to flush a file:

```
uStream input;
...
uFflush( input );
```

11.5 Closing a Formatted File

A formatted file is closed with the `uFclose` routine.

`uFclose(` *file-id* `);`

file-id is a `uStream` identifier to the formatted file.

`uFclose` closes the file, and destroys the virtual processor and the cluster associated with it. The following example shows the closing of a file:

```
uStream input;
...
uFclose( input );
```

Appendix D contains a complete formatted file program.

12. Unformatted I/O

To aid the programmer, there are cover routines for the UNIX unformatted I/O operations, which work like their UNIX counterpart, but perform their operations on a separate cluster. In the unikernel case, creating a cluster returns a reference to the current and only cluster. For complete details on each cover routine, first refer to the man pages for the corresponding UNIX routine. None of the μSystem unformatted I/O routines returns an error code, as errors are handled in a different way (detailed below).

To use the unformatted I/O facilities in a C program, include the file:

```
#include <uFile.h>
```

at the beginning of each source file. This file also includes the following system files: `<sys/types.h>`, `<sys/file.h>`, `<sys/un.h>`, `<socket.h>`.

12.1 File Type

`uFile` is the type of a file identifier, as in:

```
uFile input, output;
```

which creates two variables that contain unformatted file identifier values.

12.2 Opening an Unformatted File

An unformatted file is opened with the uOpen routine.

file-id = uOpen(*unix-file-name*, *open-flag*, *protection-mode*);

file-id is a `uFile` identifier to the unformatted file. This value must be retained as it is used to subsequently access the file.

unix-file-name is the address of a string of ASCII characters representing a UNIX path name to the file, terminated by a null character.

open-flag indicates how the file is to be opened for access. See the man entry for open for the options.

protection-mode is the protection mode for a newly created file. See the manual entry for open for the protection modes.

uOpen creates a cluster and places one virtual processor on that cluster. Then uOpen migrates the calling task to the new cluster, which performs an actual UNIX open operation, creating the UNIX file descriptor on the virtual processor for that cluster. The calling task is then migrated back to its original cluster. The following example shows the opening of an unformatted file:

```
uFile input;
...
input = uOpen( "test.c", O_RDONLY, 0 );
```

12.3 Reading and Writing from an Unformatted File

After opening a file, it is read and/or written with the routines uRead and `uWrite`. Both uRead and `uWrite` have the same parameters.

count = uRead(*file-id*, *buffer-address*, *number-of-bytes*);
count = uWrite(*file-id*, *buffer-address*, *number-of-bytes*);

count is the number of bytes actually read from the file by uRead or written to the file by `uWrite`. This number may not be the same as the number of bytes requested either because the end of file is reached for a read operation, or no more bytes can be written by the write operation.

file-id is a **uFile** identifier to the unformatted file.

buffer-address is the address of an area into which the bytes are read into or written from.

number-of-bytes is the number of bytes to be read from or written to the file buffer.

The following example shows reading from and writing to a file:

```
uFile input, output;
char *buf;
int len, count;
...
count = uRead( input, buf, len );
count = uWrite( output, buf, len );
```

12.4 Random Access Within an Unformatted File

The current location of the file pointer associated with an open file may be modified with the uLseek routine.

pos = uLseek(*file-id*, *offset*, *whence*);

file-id is a **uFile** identifier to the unformatted file.

offset depending on the value of *whence*, this value sets the file pointer, increments the file pointer, or extends the file.

whence determines how the value of *offset* is interpreted.

pos receives the updated value of the file pointer.

The following example shows how to modify a file pointer:

```
uFile direct;
off_t pos;
...
pos = uLseek( direct, 0, L_SET );     /* move pointer to beginning of file */
pos = uLseek( direct, 100, L_INCR ); /* pointer is moved forward 100 bytes */
pos = uLseek( direct, 200, L_XTND ); /* pointer is moved 200 bytes past end of file */
```

12.5 Synchronizing an Unformatted File

The routine uFsync causes all modified data and attributes of a file to be saved on permanent storage. This normally results in all modified copies of buffers for the associated file to be written to disk.

uFsync(*file-id*);

file-id is a uFile identifier to the unformatted file.

12.6 Closing an Unformatted File

An unformatted file is closed with the uClose routine.

uClose(*file-id*);

file-id is a uFile identifier to the unformatted file.

uClose closes the file, and destroys the virtual processor and the cluster associated with it. The following example shows the closing of a file:

```
uFile input;
...
uClose( input );
```

13. Socket I/O

To aid the programmer, there are cover routines for the UNIX socket operations, which work like their UNIX counterpart, but performs the operations on a separate cluster. In the unikernel case, creating a cluster returns a reference to the current and only cluster. For complete details on each cover routine, first refer to the man pages for the corresponding UNIX routine. None of the μSystem socket routines returns an error code, as errors are handled in a different way (detailed below).

A client-server model of socket communication is be used, where each client connects with a particular server and each server can connect with multiple clients. Once a connection is established between client and server, communication can be bidirectional between them. After a socket is created, it is specialized as either a server socket or a client socket. A socket can be closed and subsequently re-specialized. The following discussion on socket routines indicates for each routine, whether it is used by a client application, or a server application.

To use socket I/O facilities in a C program, include the file:

```
#include <uFile.h>
```

at the beginning of each source file. This file also includes the following system files: `<sys/types.h>`, `<sys/file.h>`, `<sys/un.h>`, `<socket.h>`.

13.1 Socket Creation

Both client and server applications must create a socket with the `uSocket` routine.

socket-id = `uSocket(` *address-format*, *communication-type*, *protocol* `);`

socket-id is a `uFile` identifier to the socket. This value must be retained as it is used to subsequently specialize the socket as a client or server.

address-format is an address format for interpreting subsequent addresses in socket operations. See the man entry for `socket` for the formats.

communication-type is a type which indicates the semantics of communication. See the man entry for `socket` for the types.

protocol is a particular protocol to be used with the socket. See the man entry for `socket` for the different protocols.

`uSocket` creates a cluster and places one virtual processor on that cluster. Then `uSocket` migrates the calling task to the new cluster, which performs an actual UNIX socket operation, creating the UNIX socket descriptor on the virtual processor for that cluster. The calling task is then migrated back to its original cluster. The following example shows the creation of a socket:

```
uFile socket;
int af, type, protocol;
...
socket = uSocket( af, type, protocol );
```

13.2 Server Socket Routines

The following routines are used by applications using sockets for the server side of the model.

13.2.1 Binding a Name to a Socket

A server application will bind a name to a socket with the `uBind` routine.

`uBind(` *socket-id*, *socket-name*, *socket-name-length* `);`

socket-id is a tt uFile identifier of a socket.

socket-name is an address for a sockaddr structure in which the socket name will be placed.

socket-name-length is a the length of the new socket name.

This socket is now a server. The following example shows the binding of a socket with a socket name making it a server:

```
uFile server;
struct sockaddr *name;
int namelen;
...
uBind( server, name, namelen );
```

13.2.2 Listening to a Socket

A server application must set a limit on the number of incoming connections from clients that will be buffered with the uListen routine.

uListen(*server-id*, *max-queue-length*);

server-id is a uFile identifier of a socket that is now a server from a call to uBind.

max-queue-length is the maximum length the queue of pending connections.

The following example shows the setting of a maximum number of connections to a server socket:

```
uFile server;
int logsize;
...
uListen( server, logsize );
```

13.2.3 Accepting a Connection

A server can accept multiple connections with the uAccept routine.

connection-id = uAccept(*server-id*, *socket-name*, *socket-name-length*);

connection-id is a uFile identifier to the connection through which communicate can occur with a client.

server-id is a uFile identifier of a server that is managing connections on a socket.

socket-name is an address for a sockaddr structure containing the socket name.

socket-name-length is a the length of the socket name.

When a client arrives, a connection is established between client and server and uAccept returns (unless the server is marked nonblocking). The *connection-id* is use in transfer data through the connection between the client and the server. After the connection is created, the server is available to establish more connections with other clients. The following example shows how to accept a connection on a socket.

```
uFile server, connection;
struct sockaddr *name;
int namelen;
...
connection = uAccept( server, name, namelen );
```

13.3 Client Socket Routines

The following routines are used by applications using sockets for the client side of the model.

13.3.1 Making a Connection

A client application makes a connection to a server with the uConnect routine.

uConnect(*socket-id*, *server-socket-name*, *server-socket-name-length*);

socket-id is a uFile identifier of a socket.

server-socket-name is an address for a sockaddr structure containing a server socket name.

server-socket-name-length is a the length of the server socket name.

uConnect returns when the socket has connected with a server socket. This socket is now a client and it communicates with the server through the connection that was created by the server's accept. The following example shows the connection of a socket with a server making the socket into a client:

```
uFile client;
struct sockaddr *server_name;
int server_namelen;
...
uConnect( client, server_name, server_namelen );
```

13.4 Communicating on a Socket

The following routines are used to communicate among clients and connections.

13.4.1 Reading and Writing from a Socket

After a connection has been established between a client and a connection for a server, communication between client and connection is performed with the uRead and uWrite routines. Both uRead and uWrite have the same parameters.

count = uRead({*client,connection*}-*id*, *buffer-address*, *number-of-bytes*);
count = uWrite({*client,connection*}-*id*, *buffer-address*, *number-of-bytes*);

count is the number of bytes actually read from the socket by uRead or written to the socket by uWrite. This number may not be the same as the number of bytes requested if the requested amount of bytes had not yet arrived. uRead operations do not always block until the socket receives the requested amount of bytes. Rather, when some bytes have arrived, and a significant delay has passed, the uRead routine will return with only those bytes. Therefore, an application may have to poll the socket until it receives the requested number of bytes.

{*client,connection*}-*id* is a uFile identifier to a client or a connection.

buffer-address is the address of an area into which the bytes are read.

number-of-bytes is the number of bytes to be read from the socket into the buffer.

The following example shows bidirectional communication from a client to some connection:

```
uFile client;
char *buf;
int len, count;
...
count = uRead( client, buf, len );
count = uWrite( client, buf, len );
```

13.5 Closing a Socket

A client application can close a socket with the `uClose` routine. A server application can close either a connection or close the socket with the `uClose` routine.

`uClose(` {*client,server,connection*}*-id* `);`

{*client,server,connection*}*-id* is a `uFile` identifier to a client, server or connection.

There is a significant difference between closing a server or closing a connection. Closing a server causes the entire socket to be destroyed, and no more communication is possible. Closing a connection causes only that connection to be terminated, and the server remains available for further communications from clients. The following is an example of closing a server.

```
uFile server;
...
uClose( server );
```

Appendix E contains a complete socket program.

14. Errors

Errors in the μSystem are divided into three categories:

- A user task detects an error and wants to abort execution. The preferred way for a user's program to stop execution while running within the μSystem is to call routine uError. The UNIX routines `exit` and `abort` are designed for single process programs and will not work as expected in the multikernel.
- The μKernel discovers that some error has occurred and it calls `uError`. Examples of such errors are running out of memory or sending a message that is too long to be received.
- A user task executes some code that causes the UNIX process representing the virtual processor to fault. The death of the UNIX process will be caught by a task executing on the parent process of the terminating process. In general, this is a task in the system cluster, which calls routine `uError`. For example, if a task tries to divide by zero or access memory out of the address space currently available to the application, these errors will be trapped. In such situations, the UNIX signal number of the terminating process is displayed in the error message. Hence, when the μSystem displays a message saying that a UNIX process died, the cause of that UNIX process's death can be determined.

The following is a list of the possible errors that the μKernel may report as the result of a user task request.

- task deadlock
- stack overflow
- death message too long
- send message too long
- reply message too long
- reply not to sender
- forward not from sender
- out of memory
- out of processors
- processor death

The list of UNIX errors that may be reported as the result of processor death may be looked up in `/usr/include/signal.h`.

14.1 Error Handling

The current μSystem error handling facilities are simple and result in immediate program termination. Currently, there is no way for a user task to deal with errors in a programmatic way. Error handling facilities will be extended in the immediate future by an exception handling mechanism.

The routine `uError` prints a user specified string which is presumably a message describing the error, and then prints the identity of the task calling the routine and the current value of the UNIX signal number.

`void uError(` *format*, *arguments ...* `)`

format is a format string containing text to be printed and format codes which describe how to print the following variable number of arguments.

arguments ... is a list of arguments to be formatted and printed on standard output. The number of elements in this list must match with the number of format codes.

14.2 Symbolic Debugging

The symbolic debugging tools (e.g. `dbx`) do not necessarily work well with the μSystem. This is because each coroutine and task has its own stack, and the debugger does not know that there are multiple stacks. When a program terminates with an error, only the stack of the coroutine or task in execution at the time of the error will be understood by the debugger. Further, in the multiprocessor case, there are multiple UNIX processes that are not necessarily handled well by all debuggers. Nevertheless, it is possible to use many debuggers on programs compiler with the uniprocessor μKernel. At the very least, it is usually possible to examine some of the variables, externals and ones local to the current coroutine or task, and to discover the statement where the error occurred. The `gdb` debugger works well in uniprocessor form, but time-slicing must be turned off if breakpoints are to be used.

15. Pre-emptive Scheduling and Critical Sections

In general, the μKernel and UNIX library routines are *not* written to allow multiple tasks to execute them. For example, many random number generators maintain an internal state between successive calls and there is no mutual exclusion on this internal state. Therefore, one task that is executing the random number generator can be pre-empted and the generator state can be modified by another task. This can result in problems with the generated random values or errors. One solution is to supply cover routines for each UNIX function, which guarantees mutual exclusion on the call. In general, this is not practical as too many cover routines would have to be created.

Our solution is to allow pre-emption only in user code. When a pre-emption occurs, the handler for the interrupt checks if the interrupt location is within user code. If it is not, the interrupt handler resets the timer and returns without rescheduling another task. If the current interrupt point is in user code, the handler causes a context switch to another task. In the unikernel case, this means that μSystem cover routines like `uOpen` are not necessary; however, in the multikernel case `uOpen` is necessary to deal with the blocking I/O problem. To ensure portability between unikernel and multikernel, μSystem supplied cover routines, like `uOpen`, should always be used.

Determining whether an address is executing in user code is done by relying on the loader to place programs in memory in a particular order. μSystem programs are compiled using a program that invokes the C compiler and includes all necessary include files and libraries. The program also brackets all user modules between two precompiled routines, `uBeginUserCode` and `uEndUserCode`, which contain no code. We then rely on the loader to load all object code in the order specified in the compile command. This results in all user code lying between the address of routines `uBeginUserCode` and `uEndUserCode`. The pre-emption interrupt handler simply checks if the interrupt address is between the address of `uBeginUserCode` and `uEndUserCode` to determine if the interrupt occurred in user code.

16. Installation Requirements

The μSystem runs on the following processors:

- 68000 series
- NS32000 series
- VAX
- MIPS
- Intel 386

The μSystem runs on the following operating systems:

- BSD 4.{2,3}
- UNIX System V that has BSD system calls `setitimer` and a `sigcontext` passed to signal handlers which contains the location of the interrupted program
- Apollo SR10 BSD
- Sun OS 4.0
- Tahoe BSD 4.3
- Ultrix 3.0
- DYNIX

The uniprocessor μSystem runs on the following vendor's computers: DEC, Apollo, Sun, MIPS, Sequent, SGI. The multiprocessor μSystem runs on the following vendor's computers: Sequent Symmetry and Balance, SGI.

The μSystem requires at least GNU C 1.35 [Sta89] for all computers except the MIPS, which requires at least GNU C 1.36. This compiler supports both K&R C and ANSI C [KR88] (see `man gcc` for information) and can be obtained free of charge. The μSystem will NOT compile using other compilers due to the inline assembler statements that appear in the C machine dependent files and the use of structure constructors for initialization. The Sequent versions is setup so that GNU C always uses the Sequent assembler because the GNU assembler does not handle the assembler directives generated from GNU C when the `-fshared-data` flag is used. This allows the uSystem to function when GNU C is installed using the GNU assembler.

17. Reporting Problems

If you have problems or questions or suggestions, you can send e-mail to `usystem@maytag.waterloo.edu` or mail to:

μSystem Project
c/o Peter A. Buhr
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
CANADA

References

[Che82] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability.* American Elsevier, 1982.

[Cor88] G. V. Cormack. A Micro Kernel for Concurrency in C. *Software–Practice and Experience*, 18(4):485–491, May 1988.

[Dij68] E. W. Dijkstra. The Structure of the "THE"–Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Pentice Hall Software Series. Prentice Hall, second edition, 1988.

[Mar80] Christopher D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis.* Springer-Verlag, 1980.

[Sta89] Richard Stallman. The Free Software Foundation's Gnu C Compiler. Free Software Foundation, 1000 Mass Ave., Cambridge, MA, U. S. A., 02138, 1989.

A Coroutine Example

```
/* Producer-consumer problem, full coroutines */

#include <uSystem.h>

long random( void );

void Producer( uCoroutine *cons, uCoroutine creator, int NoOfItems ) {
    int i, product;

    uSuspend( U_NULL, 0, U_NULL, 0 );                    /* wait for consumer to be created */
    uPrintf( "Producer will produce %d items for the consumer\n", NoOfItems );
    for ( i = 1; i <= NoOfItems; i += 1 ) {
        product = random() % 100 + 1;
        uPrintf( "Producer:  %d\n", product );
        uResume( *cons, U_NULL, 0, &product, sizeof(product) );
    } /* for */
    product = -1;                                         /* terminal value */
    uResume( *cons, U_NULL, 0, &product, sizeof(product) ); /* terminate consumer */
    uResumeDie( creator, U_NULL, 0 );                     /* return to creator */
} /* Producer */

void Consumer( uCoroutine *prod ) {
    int product;

    uSuspend( &product, sizeof(product), U_NULL, 0 );    /* wait for producer */
    while ( product >= 0 ) {
        uPrintf( "Consumer:  %d\n", product );
        uResume( *prod, &product, sizeof(product), U_NULL, 0 );
    } /* while */
} /* Consumer */

void uMain() {
    uCoroutine prod, cons;

    prod = uCocall( U_NULL, 0, Producer, &cons, uThisCoroutine(), 10 ); /* create producer */
    cons = uCocall( U_NULL, 0, Consumer, &prod );        /* create consumer */

    uResume( prod, U_NULL, 0, U_NULL, 0 );                /* start producer */

    uPrintf( "successful completion\n" );
} /* uMain */
```

B P/V Example

```
/* Producer and Consumer Problem using P/V with a Bounded Buffer */

#include <uSystem.h>
#define QueueSize 10

extern long int random( void );

struct shrqueue {
    int front, back;                              /* position of front and back of queue */
    uSemaphore full, empty;                       /* synchronize for full and empty buffer */
    int queue[QueueSize];                         /* queue of integers */
}; /* shrqueue */

void Producer( struct shrqueue *q, int NoOfItems ) {
    int i, product;
    uPrintf( "Producer will produce %d items for the consumer\n", NoOfItems );
    for ( i = 1; i <= NoOfItems; i += 1 ) {       /* produce a number of items */
        product = random() % 100 + 1;             /* generate random product */
        uPrintf( " Producer:  %2d\n", product );
        uP( &(q->empty) );                        /* wait if queue is full */
        q->queue[q->back] = product;              /* insert element in queue */
        q->back = ( q->back + 1 ) % QueueSize;    /* increment back index */
        uV( &(q->full) );                         /* signal consumer */
    } /* for */
    product = -1;                                 /* terminal value */
    uP( &(q->empty) );                            /* wait if queue is full */
    q->queue[q->back] = product;                  /* insert element in queue */
    q->back = ( q->back + 1 ) % QueueSize;        /* increment back index */
    uV( &(q->full) );                             /* signal consumer */
    uDie( U_NULL, 0 );
} /* Producer */

void Consumer( struct shrqueue *q ) {
    int product;
    for ( ;; ) {
        uP( &(q->full) );                         /* wait for producer */
        product = q->queue[q->front];             /* remove element from queue */
        q->front = ( q->front + 1 ) % QueueSize;  /* increment the front index */
        uV( &(q->empty) );                        /* signal empty queue space */
      if ( product < 0 ) break;
        uPrintf( "Consumer :  %2d\n", product );
    } /* for */
    uDie( U_NULL, 0 );
} /* Consumer */

void uMain( ) {
    struct shrqueue queue = { 0, 0, U_SEMAPHORE( 0 ), U_SEMAPHORE( QueueSize ) };
    uTask Prod = uEmit( Producer, &queue, 10 );   /* create producer */
    uTask Cons = uEmit( Consumer, &queue );       /* create consumer */

    uAbsorb( Prod, U_NULL, 0 );                   /* wait for completion */
    uAbsorb( Cons, U_NULL, 0 );
    uPrintf( "successful completion\n" );
} /* uMain */
```

C Message Passing Example

```
/* Producer-consumer problem with send/receive/reply communication. */

#include <uSystem.h>

long random( void );

void Producer( uTask Cons, int NoOfItems ) {
    int i, product;

    uPrintf( "Producer will produce %d items for the consumer\n", NoOfItems );

    for ( i = 1; i <= NoOfItems; i += 1 ) {
        product = random() % 100 + 1;
        uPrintf( " Producer:  %2d\n", product );
        uSend( Cons, U_NULL, 0, &product, sizeof(product) );
    } /* for */
    product = -1;                                          /* terminal value */
    uSend( Cons, U_NULL, 0, &product, sizeof(product) );   /* terminate consumer */
    uDie( U_NULL, 0 );
} /* Producer */

void Consumer( ) {
    int product;

    for ( ;; ) {
        uReply( uReceive( &product, sizeof(product) ), U_NULL, 0 );
      if ( product < 0 ) break;
        uPrintf( "Consumer :  %2d\n", product );
    } /* for */
    uDie( U_NULL, 0 );
} /* Consumer */

void uMain( ) {
    uTask Prod, Cons;

    Cons = uEmit( Consumer );                              /* create consumer */
    Prod = uEmit( Producer, Cons, 10 );                    /* create producer */

    uAbsorb( Prod, U_NULL, 0 );                            /* wait for completion */
    uAbsorb( Cons, U_NULL, 0 );

    uPrintf( "successful completion\n" );
} /* uMain */
```

D File Example

```
#include <uSystem.h>

/* This application simply reads a file, and prints its contents on standard output. */

void uMain( int argc, char *argv[] ) {

    uStream input;
    int ch;

    switch ( argc ) {
    case 2:                                             /* program takes file name as argument */
        break;
    default:
        uError( "usage:  %s file-name\n", argv[0] );
        break;
    } /* switch */

    input = uFopen( argv[1], "r" );                     /* open file */

    for ( ;; ) {
        ch = uGetc( input );                            /* read characters from file */
      if ( ch == EOF ) break;                           /* end of file ? */
        uPutchar( ch );                                 /* write character on standard output */
    } /* for */

    uFclose( input );                                   /* close input file */
}

/* Local Variables: */
/* compile-command: "concc -quiet -work -O File.c" */
/* End: */
```

E Socket Example

E.1 Client Socket

```
#include <uSystem.h>
#include <uFile.h>
#include <sys/un.h>

void uMain( int argc, char *argv[] ) {

    uFile sd;
    struct sockaddr_un server;
    int c;
    void strcpy( char *, char * );

    switch ( argc ) {
    case 2:
        break;
    default:
        uError( "usage: %s socket-name", argv[0] );
    } /* switch */

    sd = uSocket( AF_UNIX, SOCK_STREAM, 0 );            /* create a socket */

    server.sun_family = AF_UNIX;                        /* specify socket domain */
    strcpy( server.sun_path, argv[1] );                 /* specify destination socket name */
    uConnect( sd, &server, sizeof( server ) );          /* connection to destination socket */

    for ( ;; ) {
        c = uGetc( uStdin );                            /* get a byte */
            if ( c == EOF ) break;                      /* no more bytes? */
        uWrite( sd, &c, sizeof( c ) );                  /* write byte to socket */
    } /* for */

    uWrite( sd, &c, sizeof( c ) );                      /* write end of file marker to socket */

    for ( ;; ) {
        uRead( sd, &c, sizeof( c ) );                   /* read byte back from socket */
            if ( c == EOF ) break;                      /* no more bytes? */
        uPutc( c, uStdout );                            /* put a byte */
    } /* for */

    uClose( sd );                                       /* close socket */
} /* uMain */

/* Local Variables: */
/* compile-command: "concc -quiet -work -multi -O -o Client SocketClient.c" */
/* End: */
```

E.2 Server Socket

```
#include <uSystem.h>
#include <uFile.h>
#include <sys/un.h>

void uMain( int argc, char **argv ) {

    int c;
    uFile sd;
    uFile fd;
    struct sockaddr_un server;
    void strcpy( char *, char * );

    switch ( argc ) {
    case 2:
        break;
    default:
        uError( "usage:  %s socket-name", argv[0] );
        break;
    } /* switch */

    sd = uSocket( AF_UNIX, SOCK_STREAM, 0 );

    server.sun_family = AF_UNIX;
    strcpy( server.sun_path, argv[1] );
    uBind( sd, &server, sizeof( server ) );

    uListen( sd, 5 );

    for ( ;; ) {
        fd = uAccept( sd, 0, 0 );                    /* accept a connection */
        for ( ;; ) {
            uRead( fd, &c, sizeof( c ) );            /* read byte from socket */
            uWrite( fd, &c, sizeof( c ) );           /* write byte to socket */
          if ( c == EOF ) break;                     /* no bytes left? */
        } /* for */
        uClose( fd );                                /* close the connection */
    } /* for */
} /* uMain */

/* Local Variables: */
/* compile-command: "concc -quiet -work -multi -O -o Server SocketServer.c" */
/* End: */
```

Index