# The Implementation of Newsqueak

*Rob Pike*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

The implementation of the concurrent applicative language Newsqueak has several unusual features. The interpreter, `squint`, uses a copy-on-write scheme to manage storage honoring Newsqueak's strictly applicative (by-value) semantics for data. There is no explicit scheduler. Instead, the execution of processes is interleaved very finely, but randomly, by an efficient scheme that integrates process switching into the interpreter's main loop. The implementation of `select`, the non-deterministic, multi-way communications operator, exploits details in the implementation of processes.

This paper describes much of the interpreter but explains only small aspects of the language. Further detail about the language may be found in the references.

**Storage**

Since the management of storage is central to the implementation of any language, it is a good starting point for the description of the Newsqueak interpreter. But it is especially pertinent for Newsqueak because the design of the language hinges on its strictly by-value (applicative) management of data.

Concurrent and applicative programming complement each other. The ability to send messages on channels provides I/O without side effects, while the avoidance of shared data helps keep concurrent processes from colliding. Newsqueak is an applicative concurrent language based on the concurrent composition of processes communicating on synchronous channels [Pike 89, McIl 89]. Two computations share a value only if they share a variable. All assignment of values to objects — including function return, binding of parameters to functions and processes, and passing values on channels — is done by making a copy of the assigned value. Although Newsqueak permits global variables, programming without them is convenient and implicitly encouraged. The language promotes independent functions and processes operating in environments defined exclusively by their parameters. Since those parameters may include communication channels, the language feels considerably different from traditional applicative languages.

Newsqueak does not require that its implementation copy values in an assignment, but the behavior must be as though the value were copied. The Newsqueak interpreter, `squint`, combines reference-count garbage collection with a lazy copying scheme to defer copying as long as possible. The result is similar to copy-on-write page management schemes in operating systems.

Reference counting was chosen because it is easy to implement [Knuth 73]. It was (correctly) expected that storage management would not dominate performance. Moreover, when the language was being designed there were several candidates for the CPU it would eventually be run on, so it seemed prudent to use simple, inherently portable techniques.

Each object in Newsqueak is represented using a single word containing the address of a data structure describing the object, except for integers, which are just stored in the word. (The language itself has no pointer types, but the implementation uses pointers extensively.) The data structure begins with a header that includes a reference count and a description of the size and layout of the object, which simplifies copying the structures. (The static type system requires no run-time checking of the type of objects, but the interpreter uses a single routine to copy all objects.) The data contained in arrays of characters and integers are stored immediately after the header. Arrays of non-integral objects are represented recursively by storing an array of pointers to the component objects after the initial header. Records, defined using the `struct` keyword, are stored using a hybrid scheme; a bit array at the beginning of the data part of the object indicates which elements of the structure are represented by pointers.

An object is freed (collected) when the number of references to it goes to zero, which can occur when a link to the object is broken by assignment or when a variable pointing to the object is released at the end of an executing block. When an object is freed, pointers contained within it are followed and their reference counts are decremented. When these counts go to zero, the algorithm continues recursively.

When an object is assigned to a variable, its reference count is increased. However, when a component of an object is updated, the resulting assignment may be done in place only if the containing object has a reference count of one. If not, the object must first be copied. Consider the isolated assignments

```
A = somearray
B = A
A[i] = p.
```

After the second assignment, the array pointed to by `A` and `B` has a reference count of at least two. To assign `p` to `A[i]`, `A` is copied, the old array's reference count is decremented (`A` no longer points to it, but `B` still does), all its component objects' reference counts are incremented (the new array points to the same components), and `A` is made to point to the new object. This new object has a reference count of one. The original object pointed to by `A[i]` and `B[i]` has its reference count decremented (after assignment `A[i]` will not point to it but `B[i]` will), and `A[i]` may finally be pointed at `p`, whose reference count increments. All these operations occur as a single atomic action. Care must be taken in the implementation to guarantee that assignments such as

```
A[i] = A[i]
```

work properly, even when disguised, for example by communication on a channel. But the method is not hard to implement, and is beneficial for many common cases such as passing an array to a function that examines it but does not change it.

**Processes and scheduling**

The unusual handling of processes and scheduling in `squint` is most easily approached the same way it was developed: by successive refinement of a basic interpreter. Many interpreters represent the target program as an array of function pointers, each of which represents pseudo-instructions in a simulated CPU, and use a program counter to step through those functions. The end of the program is indicated by a pseudo-instruction that returns zero (false); non-terminal pseudo-instructions return one (true). The code in C looks like:

```
typedef int     (*Inst)(void);

Inst    *pc;
Inst    program[];

compile();
pc=program;

while((**pc++)())
        ;
```

As well, there are often some global pseudo-registers, such as a stack pointer, and some associated data, such as a stack. These global variables are manipulated by the pseudo-instructions to push variables on the stack and perform other low-level operations.

Newsqueak needs processes. Since the implementation is a single (real) process in a C program, we need to simulate processes by interleaving the execution of various Newsqueak programs in a single instance of the basic loop. When a (simulated) process is not actively executing, its state can be held in a data structure such as — schematically at least —

```
typedef struct{
    Inst    *pc;        /* program counter */
    int     *sp;        /* stack pointer */
    int     stack[N];   /* stack storage */
}Proc;
```

A typical operating systems approach at this point would be to use the `Proc` structure to hold the `pc` and `sp` of a suspended process and to swap them with the global variables when the process is enabled again. (The stack storage in the `Proc` structure could be used as is, without copying.) The execution loop might then become

```
while(a process can run){
        while(randomtimer()!=0 && (**pc++)())
                ;
        swap(schedule());
}
```

where `schedule` selects a process (i.e., a `Proc` pointer) to run and `swap` exchanges the globals with the named process. Again, this is a CPU-like model; the timer represents some sort of clock that drives the preemptive scheduling algorithm. It might be implemented by having a software interrupt set a flag, or just by running a counter.

Of course, these processes should be communicating, so some scheduling can be tied to communication. For example, when a sending process, say A, detects (through a data structure representing a channel, which will be described in the next section) that another process, say B, is ready to receive its message, A can execute

```
swap(B)
```

thereby passing the flow of control to the receiving process. In general, though, this approach does not obviate the need for preemptive scheduling. If a process does not communicate often, it may never get run. Worse, tying the scheduling to communication makes the system very deterministic. Concurrent languages, particularly those originating with Hoare's Communicating Sequential Processes (CSP) [Hoare 78], have a tradition of non-determinism derived from a combination of Dijkstra's guarded commands [Dijk 76] and distributed computation. Non-determinism also has the advantage of avoiding certain classes of livelock that can occur when communicating with a chatty process. Newsqueak therefore should be non-deterministic when scheduling and when choosing between multiple potential communicators. To provide non-deterministic scheduling without interrupting timers, we need a different structure for the interpreter loop.

`Squint` has no scheduler and no global program counter or stack pointer. Instead, the state of all processes is described only by the `Proc` structures, using a model related to hardware microtasking [Thack 79] and the HUB miniature operating system [ODell 87] [Mas 76]. Rather than running the pseudo-instructions by executing

```
(**pc++)()
```

`squint` executes

```
(**proc->pc++)(proc)
```

where `proc` points to the head of a queue of active processes. Each pseudo-instruction needs the `Proc` pointer of the current process to access the appropriate stack and registers. These indirections may cost some execution time, of course, but the hope is to gain some back by not having to save and restore process state when scheduling. By not saving state when scheduling, all that is required is to change `proc` to run the new process. That is inexpensive enough to do after every pseudo-instruction, if we can cycle through the process queue cheaply. Given a process queue and a `next` pointer in each `Proc` the code becomes:

```
typedef int     (*Inst)(Proc*);

Proc    *proc;  /* head of process queue */
Proc    *ptail; /* tail of process queue */

while(proc){
        while((**proc->pc++)(proc)){
                ptail->next = proc;
                ptail = proc;           /* append proc to tail */
                proc = proc->next;      /* delete proc from head */
        }
}
```

The queue manipulation succeeds even if the queue has only one process. Nonetheless, in the common case that only a single process is running, we can improve the loop by leaving the queue alone if `proc` equals `ptail`:

```
while(proc){
        while((**proc->pc++)(proc)){
                if(proc != ptail){
                        ptail->next = proc;
                        ptail = proc;           /* append proc to tail */
                        proc = proc->next;      /* delete proc from head */
                }
        }
}
```

If only one process is running, the scheduling overhead is one comparison per pseudo-instruction, plus the cost of accessing the simulated registers through a pointer (which may be negligible; it depends on the architecture of the real CPU). If several processes are active, though, `proc` is not equal to `ptail` and each execution of the loop accesses several global variables. We can do better by amortizing the cost over several instructions, by scheduling less often. By being statistical rather than absolute in deciding how often, we have an opportunity to introduce non-determinism into the scheduler.

To randomize the interleaving of the processes, we need an extremely cheap way to decide how to interleave. The first requirement is a cheap random number generator. It does not have to be good, it just needs to be good enough that programs cannot exploit any correlations. The following generator, courtesy of Jim Reeds, uses a 31-bit linear feedback shift register to derive a random enough number in a handful of minor instructions on most 32-bit computers:

```
long x = 0xFFFFFFFFL;

x += x;
if(x < 0)
        x ^= 0x88888EEFL;
n = x&MASK;
```

(The & operator is bitwise and; ^ is bitwise exclusive or.) The resulting n is a random number between 0 and MASK; MASK is 15 in squint. With the generator in the loop, the result is:

```
while(proc){
        x += x;
        if(x < 0)
                x ^= 0X88888EEFL;
        n = x&MASK;
        while(--n>=0 && (**proc->pc++)(proc))
                ;
        if(proc != ptail){
                ptail->next = proc;
                ptail = proc;           /* append proc to tail */
                proc = proc->next;      /* delete proc from head */
        }
}
```

With x and n in registers, this loop runs insignificantly slower than the non-random one on a VAX-11 with a single process, and about 40% faster with several processes. It also offers non-determinism and requires no timer. The only remaining problem is to work in the scheduling requirements of inter-process communication, the subject of the next section.

Newsqueak provides a process creation operator (begin) but no explicit process destruction operator. When a process is instantiated, it is wrapped in an envelope that converts what would be its top-level function return into the sequence that releases the function's resources (by decrementing reference counts of the top-level data structures) and removes the process from the run queue. Garbage collection does the rest.

**Communication**

Consider a process S sending an integer i to a process R, using a channel c. S executes

```
c<- = i
```

and R executes

```
v = <-c
```

to save the value in v. Communication in Newsqueak is synchronous, which means that both S and R must be ready to communicate before either can do so; if, for instance, S tries to send on a channel when no receiver is ready, it suspends execution until a receiver, here R, tries to read from the same channel. If a process is prepared to communicate on a set of channels, it uses a `select` statement, which announces the possibility of communication and then executes a sequence of statements labeled by the single communication that finally proceeds. For example,

```
select{
    case i = <-c1:
        a = 1;
    case c2<- = i:
        a = 2;
}
```

sets `a` to 1 if the process receives a value from channel `c1`, or to 2 if the process sends a value on `c2`. If neither `c1` nor `c2` can communicate, the process suspends until one can. If both can, a non-deterministic choice is made. The semantics of these operations, but not the syntax, is taken from CSP [Hoare 78] and Occam [INMOS 84].

Assume that `S` reaches the communication point (rendezvous) first. `S` must wait — suspend execution — until `R` arrives, and when `R` does arrive `S` must resume execution. The channel `c` has a pair of queues of processes, `sendq` and `rcvq`, to which processes append themselves while awaiting rendezvous.

The following sequence, structured as pseudo-instructions executed by `S` and `R`, ensues. Each numbered step is a single pseudo-instruction, labeled by the process that executes it, except that steps 1, 5, and 7 may be many pseudo-instructions in a more complex example. Steps $S_3$ and $R_3$ are executed implicitly and described in the text. Some of the details in the sequence allow `select` to work, and will be explained later. As described, these actions assume that no other processes are executing, that the queues are initially empty, and so on, but the sequence works (implements first-in-first-out rendezvous and communication) when arbitrarily many processes are communicating in arbitrary order. Assume `S` arrives at the rendezvous first:

1. $S_1$: -Evaluate `c`.
   The channel is now on `S`'s stack.

2. $S_2$: -Examine `c`, notice that `c.rcvq` is empty.
   -Append `S` to `c.sendq`.
   -Suspend execution.
   If `c.rcvq` had an entry, `R` would have arrived first, and the order of execution between `S` and `R` would be mirrored. But since the queue is empty, `S` announces in `c.sendq` that it is waiting for a receiver, and suspends execution by returning zero (false) from this pseudo-instruction. At some later time, `R` reaches the rendezvous.

3. $R_1$: -Evaluate `c`.
   The channel is now on `R`'s stack.

4. $R_2$: -Examine `c`, notice that `c.sendq` has an entry.
   -Therefore remove the head (that is, `S`) from `c.sendq.`
   -Execute $S_3$ for `S` (see text).
   -Place `R` and `c` on `S`'s stack.

-Skip $R_3$ by incrementing `R->pc`.

-Enable `S` by placing `S` in the queue of running processes.

-Suspend `R`.

`R` finds `S` in `c.sendq` and prepares to communicate with it. `R` must tell `S` who is receiving its data; the channel data structure (to identify which communication succeeded in a `select`) and the pointer to `R` are now placed on `S`'s stack.

Because `R` arrives second at the rendezvous, it must be the process to remove the queued process (`S`) from the channel's queue. It does this by executing $S_3$ (that is, `(**S->pc++)(S)`) which is a single pseudo-instruction the interpreter has generated to remove `S` from `c.sendq`. This must be done before allowing any other process to execute, to avoid conflicts should another process be attempting to communicate using `c`. Rather than just unqueueing `S`, `R` temporarily calls upon `S` because `S` may be in a `select` and have more bookkeeping to do. The next pseudo-instruction in `R`'s stream, $R_3$, would remove `R` from `c.rcvq`, and would be called by `S` had `S` arrived second. Since `R` did, the operation is just skipped.

5. $S_4$: -Evaluate `i`.

`S` now wakes up and evaluates `i`, placing it on its stack. The value to be sent may, of course, be an arbitrary expression, even one involving communication. We will return to this subject below.

6. $S_5$: -Remove `R` from stack.

-Place `i` on `R`'s stack.

-Resume `R`.

`S` now completes its half of the exchange. With the receiving process and value to be sent in hand, all that remains is to hand off the value and resume `R` by appending it to the queue of running processes.

7. $R_4$: -Continue.

`R` now has the value on its stack and can proceed with that value by normal execution.

This sequence is an expansion of the independently developed, abstract model by Cardelli [Card 84], which does not integrate the communications operations into an interpreter.

If a process, say `S`, is executing a `select`, the sequence remains essentially the same but the individual operations `S` executes are more involved. Whether `S` is selecting is invisible to `R`.

Imagine that `S` is executing a select. In step $S_1$, the process evaluates all channels involved, pushing all of them on its stack and recording which are being used to send and which to receive. (Although our process is `S`, it may be sending in a `select` that also has receiving communications.) This may, of course, take many pseudo-instructions. In step $S_2$, `S` looks at all channels at once. There are two possibilities: no communication may proceed, or some may. In this example, we assume none may, so `S` atomically appends itself to all appropriate queues for channels mentioned in the select, and suspends execution. When `R` arrives at the rendezvous, it finds `S` in `c.sendq`. When `R` then executes `(S->pc++)(S)`, `S` removes itself from all queues on which it has attempted communication. `R`'s arrival chooses which communication proceeds.

Newsqueak allows `select` to be applied to arrays of channels. Given an array

```
a: array[N] of chan of int;
```

the statement

```
select{
    case <-a[]:
        i = 1;
}
```

is identical in behavior to the statement

```
select{
    case <-a[0]:
        i = 1;
    case <-a[1]:
        i = 1;
...
    case <-a[N-1]:
        i = 1;
};
```

in other words, each element of the array participates equally in the selection. Array selections are easy to implement; all that needs to be done is to post each element as a potentially communicating channel, that is, to perform at run-time the rewriting above.

A complicating factor is that `S` needs to know *which* channel communicated, so it can execute the appropriate subsequent statements. This is why the channel is passed from `R` to `S` during the rendezvous.

The rest of the execution is straightforward.

Some of the details of the execution sequence are necessary to make `select` work, and in particular to keep a process unaware of its partner's participation in a `select` statement. The largest effect of this constraint is in the order of evaluation. A selecting, sending process must not evaluate the communicated value before the rendezvous, as the evaluation may involve side effects that would be inappropriate if a different communication in the `select` proceeded. Newsqueak therefore specifies that the rendezvous occurs before the transmitted value is evaluated, which has some subtle effects. For example, in the expression

```
c1<- = <-c2
```

(send on `c1` the value received on `c2`), the `c1` rendezvous happens before the `c2` rendezvous.

Selection brings up the possibility of nondeterminism: a process must choose which of a set of ready channels should communicate and, similarly, which of a set of cases corresponding to the same channel should be executed. This latter issue is exemplified in the code fragment:

```
select{
    case <-c:
        i = 1;
    case <-c:
        i = 2;
}
```

Both of these problems may be addressed efficiently by applying the following little-known single-pass choice algorithm. Given an array of integers `a` of known length, but containing an unknown number of non-zero entries, the problem is to choose, fairly, one non-zero element. The following algorithm leaves `c` set to the index of the chosen element. If `c` is `−1` after the loop, no non-zero elements exist.

```
int a[N]
n = 1
c = −1
for(i=0 to N){
        if(a[i] ≠ 0){
                if((random() mod n) == 0)
                        c = i
                n = n+1
        }
}
```

Proof by induction: When the first non-zero element is found, `n` is one so `(random() mod n)` is zero, and `c` records the current element. If `n−1` elements have been found, when the `n`th element is found, the probability that the current element should replace the choice so far is $\frac{1}{n}$, which is simulated by `(random() mod n)==0` where `random()` generates integers much larger than `n`. For this algorithm, `squint` uses the simple congruential random number generator from the ANSI C standard [K&R 88], except that it includes the modulus calculation:

```
int
nrand(int n)
{
        static unsigned long next=1;
        next = next*1103515245 + 12345;
        return (next/65536) mod n;
}
```

A better generator would be overkill, and there are advantages of testability and portability to including the generator in the program rather than calling upon a library function.

**Discussion**

The implementation of processes used in the Newsqueak interpreter allows for fine-grained non-deterministic interleaving without resort to interrupts or timers. Because process switching is almost free, there is negligible penalty for using many processes in an application. As McIlroy demonstrates [McIl 89], the interpreter's performance is acceptable for the jobs the language was intended to handle. Applications that may profitably be written as sets of communicating processes run in time comparable to the same programs expressed using more traditional methods. On a VAX 8550, a simple test program executes about 8000 transactions per second on a `chan of int`. This is creditable but not spectacular performance. Because the system is an interpreter, however, communication is very cheap relative to more traditional calculations. If the interpreter were instead a compiler, simple computations would execute more quickly but the communications operators would probably not run much faster, which might cancel some of the charm of using communications in programs (there would be temptations to improve performance by optimizing out communications).

Although the interpreter uses reference-count garbage collection for reasons of simplicity and portability, the greatest benefit of reference counting is that it makes copy-on-write storage management possible. The philosophical advantages of efficient array management in a completely by-value language are clear and worthwhile. For Newsqueak programs, which can contain no pointer cycles and which will tend to use arrays rather than lists, sophisticated collection methods are unnecessary and may in fact not gain much performance over copy-on-write reference counting. The ease of porting the interpreter was also an issue. Fancy collection schemes often involve low-level implementations. Partly because storage management, like the rest of the Newsqueak interpreter, is implemented entirely in C, the system has been compiled and run successfully without change on half a dozen architectures.

Finally, a word about size. The complete program, including parser, type checker, interpreter and run-time libraries, is fewer than ten thousand lines of C code. This is a comfortable size for an experimental language, small enough to encourage experimentation with the language and its implementation.

**References**

[Card 84] Cardelli, L., ''An implementation model of rendezvous communication,'' *Proc. of NSF-SERC Seminar on Concurrency,* CMU, 1984. Published by Springer-Verlag.

[Dijk 76] Dijkstra, E.W., *A Discipline of Programming,* Prentice Hall, Englewood Cliffs, NJ, 1976.

[Hoare 78] Hoare, C.A.R., ''Communicating Sequential Processes,'' *Comm. ACM* **21**(8), pp. 666-678, 1978.

[INMOS 84] *Occam Programming Manual,* Prentice Hall International, Englewood Cliffs, NJ, 1984.

[Knuth 73] *The Art of Computer Programming, Volume 1, Second Edition,* pg. 412, Addison-Wesley, Reading, MA, 1973.

[K&R 88] *The C Programming Language, Second Edition,* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Mas 76] Masamoto, K., ''Implementation of HUB Processor,'' Master's Thesis, University of Illinois at Champagne-Urbana, 1976.

[ODell 87] O'Dell, M. D., ''The HUB: A Lightweight Object Substrate,'' *Proc. of EUUG Conference,* Dublin, Autumn, 1987

[McIl 89] McIlroy, M. D., ‘‘Squinting at Power Series,’’ Bell Labs, 1989

[Pike 89] Pike, R., ‘‘Newsqueak: A language for communicating with mice,’’ *Computing Science Technical Report 143,* AT&T Bell Laboratories, Murray Hill, New Jersey, 1989

[Thack 79] Thacker, C. P., McCreight, E. M., Lampson, B. W., Sproull, R. F., and Boggs, D. R., ‘‘Alto: A Personal Computer,’’ Xerox Palo Alto Research Center, Palo Alto, CA, 1979.