Automatic management of operating-system resources

Olin Shivers
MIT AI Lab
Cambridge, Massachusetts, USA
shivers@ai.mit.edu

Abstract

One of the attractive features of functional programming languages is that they provide automatic management of the store, in the form of garbage collection. However, the benefits of automatic resource management can be applied to other resources as well. Scsh, a systems-programming dialect of Scheme, provides automatic resource management for operating-systems structures, notably processes, I/O channels, and signal events. This kind of automatic management extends the benefits of garbage collection—modularity, robustness, simplicity, and clarity—to new sets of objects in programming, and also gives us hints as to how operating systems should be structured from the perspective of functional programming languages.

1 Introduction

Functional programming languages have provided garbage collection (GC) of user-allocated data structures for decades [Wilson]. There are many benefits to garbage collection. GC eliminates, at one stroke, several classes of run-time errors, chiefly those involving dangling pointers and storage leaks. It also simplifies source code by eliminating the storage-reclamation machinery from the application program. This makes programs simpler and clearer. Finally, GC promotes modularity—in systems that require explicit storage management, a module that exports a data structure to clients must also provide some way to tell when client modules no longer require the data structure.

Scsh is a dialect of Scheme [R4RS] that is designed for systems programming [scsh, scsh-man]. As such, it provides complete interfaces to the operating system. By far the most complex and difficult parts of the design involved providing interfaces to the underlying OS resources that harmoniously integrated the OS functionality with the programming language. For a programming language like Scheme, this typically meant allowing for automatic management of the exported resources. The design of the I/O channel interface, for example, required nine months and four redesigns before I was satisfied with it.

As users gained experience with scsh, certain classes of problem with the design began to show up. These problems were typically fixed by taking some operating-system resource, making it abstract, and providing automatic management for it. It was striking how quickly serious systems use turned up these design flaws. In the end, the principle of providing the OS resources in a form that admitted automatic management turned out to be an important guiding heuristic in developing the system to its current state. In this paper, I would like to describe the techniques employed and the lessons learned from this experience.

Scsh provides automatic management of three types of OS resource: I/O channels, processes, and signal events. In the next three sections of the paper, we'll examine in detail the interface scsh provides for each of these three types of resource. {Note Unix system calls}

To appear in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, June 1997, Amsterdam. Also available as `http://www.ai.mit.edu/~shivers/os-gc.ps`

2 Managing I/O ports

Perhaps the most difficult part of the design of scsh was the integration of Scheme ports and Unix file descriptors. Dealing with Unix file descriptors in a Scheme environment is difficult. In Unix, open files are part of the process state, and are referenced by small integers called *file descriptors*. Open file descriptors are the fundamental way I/O redirections are passed to subprocesses, since file descriptors are preserved across system calls that create subprocesses.

Scheme, on the other hand, uses ports for specifying I/O sources. Ports are anonymous, garbage-collected Scheme objects, not integers. When a port is collected, we can deallocate any underlying OS resources on which it depended. By contrast, because file descriptors are just integers, it's impossible to garbage collect them—in order to close file descriptor 3, you must prove that the process will never again pass a 3 as a file descriptor to a system call doing I/O, and that it will never invoke a program that will refer to file descriptor 3. This is difficult at best.

If a Scheme program only used Scheme ports, and never directly used file descriptors, this would not be a problem. Many Scheme implementations provide this kind of simple automatic closure of file descriptors on port GC. But Scheme code that does serious systems programming in Unix must violate the abstraction and descend to the file-descriptor level in at least two circumstances:

- when interfacing to foreign code;
- when interfacing to a subprocess.

This causes problems. Suppose we have a Scheme port constructed on top of file descriptor 2. We intend to fork off a C program that will inherit this file descriptor. If we drop references to the port, the garbage collector may prematurely close file 2 before we exec the C program.

Scsh's Unix I/O interface is intended to address the problems that arise from the mismatch between ports and file descriptors. The fundamental principle is that in scsh, ports are attached to open files, not to particular file descriptors. For most Scheme I/O ports, the user is completely unaware which underlying file descriptor has been allocated for that port. This means that the scsh run-time system is free to shuffle file descriptors around underneath the ports with the `dup` system call should it become necessary to make a specific file descriptor available.

The scsh run-time system keeps careful track of which I/O ports are completely abstract by associating a *revealed* count with each port. If the *revealed* count is zero, then the port's underlying file descriptor has not been exported to the user program. Ports created with the standard procedures such as `open-input-file` have their *revealed* count set to zero. If the programmer uses the `port->fdes` procedure to expose a port's underlying file descriptor, its *revealed* count is incremented. Once a port's file descriptor has been revealed, it is the programmer's responsibility to manage the resource explicitly. The programmer can use the `release-port-handle` procedure to decrement the port's *revealed* count when he no longer needs to know the file descriptor.

Not all file descriptors are allocated by requests to make ports. Some are inherited on process invocation via `exec`, and are simply part of the global environment. Subprocesses may depend upon them, so if a port is later allocated for these file descriptors, it should be considered as a revealed port. For example, when a scsh process starts up, it creates ports on top of the standard file descriptors 0, 1, and 2 for the initial values of `(current-input-port)`, `(current-output-port)`, and `(error-output-port)`. These ports are initialised with *revealed* set to 1, so that stdin, stdout, and stderr are not closed even if the user drops the port.

Keeping track of which ports have not had their file descriptors revealed allows the run-time system several important freedoms:

- When an unrevealed port becomes eligible for garbage collection, its underlying file descriptor can also be de-allocated with a `close` system call. If the Scheme user doesn't know what file descriptor is associated with a given port, then he can't possibly retain an "integer handle" on the open file after dropping pointers to the port itself, so the garbage collector is free to close the file.

- For similar reasons, the file descriptor can be closed when the Scheme process uses the `exec` system call to overlay itself with a new program. All current memory is being dropped, so the port structure itself is being deallocated. If no one knows the file descriptor associated with the port, then the file descriptor is also eligible for deallocation.

- The run-time system is free to shift an unrevealed port from one file descriptor *fd* to a new one *fd*$'$ should the user's program specifically request file descriptor *fd* for some purpose.

This last freedom is an interesting one, and addresses a long-standing problem with Unix system programming. Some programs need to perform I/O operations that explicitly allocate a specific file descriptor, *e.g.*

```
(fork (λ ()
        (dup pwd-iport  3)  ; Read password on fd 3.
        (dup data-iport 0)  ; Read plaintext on fd 0.
        (exec encryption-program)))
```

It might be the case that, unbeknownst to the programmer, Unix had already allocated file descriptor 3 on a previous operation, perhaps when it allocated the pipe for the Scheme port `data-iport`. If this is the case, our `(dup pwd-iport 3)` operation will clobber our `data-iport`. When our encryption program executes, it will try to read both password and data from the password source—not what we wanted at all!

However, if scsh knows that the original port built on top of file descriptor 3 is unrevealed, it is free to shift this port to some new file descriptor with an internal `(dup 3)` operation, freeing up file descriptor 3 to be allocated by the `(dup pwd-iport 3)` call. This is exactly what the scsh run-time system does—all system calls that allocate specific file descriptors first check to see if there is an unrevealed port already associated with the target descriptor. If there is, the port is first shifted to a new file descriptor with a `dup` call before performing the requested operation.

Note that this is not a Scheme-specific problem; it arises just as easily in C programs that manipulate file descriptors. Scsh' solution is the first one known to myself that solves the problem.

Scsh's port facility preserves the transparent close-on-collect property for file ports that are used in straightforward ways, yet allows access to the underlying Unix substrate without interference from the garbage collector. This is critical, since systems programming absolutely requires access to the actual file descriptors, as their numerical values are a critical part of the inter-process interface.

Under normal circumstances, all this machinery just works behind the scenes to keep things straightened out. The only time the user has to think about it is when he starts accessing file descriptors from ports, which he should almost never have to do. If a user starts asking what file descriptors have been allocated to what ports, he has to take responsibility for managing this information.

3 Managing processes

The Unix operating system represents processes with integer names called "process ids" or "pids." A process is created by the `fork` system call, which returns the process id of the new process. A process can synchronise with a child process by applying the `wait` system call to the child's process id. The parent process blocks until the child process exits, at which point the `wait` procedure returns the child's exit status.

Issues of resource management affect the definition of the `wait` procedure in an interesting way: The `wait` call is the mechanism whereby the OS reclaims the resources of the process named by the process id. After the `wait` call, the operating system is free to reuse the process id for a new process, and it is also free to reclaim the space in the kernel's process table that stores the process' externally visible state.

As a result, the parent must wait on the child process *exactly* once:

- *nwaits* < 1
  If the process fails to wait on children, then they accumulate in the kernel's process table. The kernel may not reuse the process id for new processes, nor may it reclaim this space as long as the child's parent is alive, since the parent might at any point decide to perform a `wait` on the child. Eventually, the space of process ids will fill up, or the process table will overflow, and the operating system will be unable to create new processes. As a trivial example, consider the following simple fragment of C code:

  ```
  while(1)
      if( !fork() ) exit();
  ```

  Although it consumes no resources asymptotically, it will eventually cause the process table to fill up with dead child processes, thus causing the kernel to be unable to create new processes.

- *nwaits* > 1
  If the process waits on a child more than once, all waits after the first will return an error—if we are lucky. An even worse failure mode lurks in store for the unwary, a classic dangling-pointer bug. After the first wait on the process id, the kernel is free to allocate the process id for a new process. It is possible that the original process id could be allocated for a new child of the original parent, causing the further waits on the original child to return the later child's exit status.

There are two points we can draw from this problem.{Note Prescient designers} First, the Unix solution is not modular. A module of code that waits on a child process must know that no other module will wait on the child. This is exactly analogous to the modularity problems with explicit storage management.

Second, as in our previous I/O channel case, small integer names are a poor substitute for abstract data types. This is a lesson that applies as much to operating-system resources as it does to user-level data structures. Because Unix uses sixteen-bit integers to name processes, the space is prone to overflow—an unnecessary failure mode, since the interface uses no interesting property of integers. It also makes it impossible to apply automatic storage management directly to the exported representation. As with the file descriptors we discussed earlier, it is impossible to garbage-collect objects named by integers.

The problem of process-table overflow is, unfortunately, a problem of practical import. The original release of the scsh system used the standard Unix model to represent processes: integer process ids on which the programmer was required to wait exactly once. This led to modularity problems that caused certain simple programs to overflow the process table.

The first bug report that led us to this problem concerned capturing process output in scsh. Scsh contains a procedure `run/port*` that forks a child process and returns an input port connected to the child's standard output. This procedure can be defined very simply:

```
(define (run/port* thunk)
  (call-with-values pipe
    (λ (r w)
      (fork (λ ()
              ;; Pipe -> stdout (fd 1) in child process
              (let ((w1 (dup w 1)))
                (with-current-output-port* w1 thunk))))
      r)))
```

Reading from the returned port `r` will return whatever the child process prints out.

Another simple procedure, `port->string`, takes an input port, and reads from it until it encounters end-of-file, at which point it returns the accumulated data as a single string. The composition of these two procedures

$$\texttt{run/string*} \equiv \texttt{port->string} \circ \texttt{run/port*}$$

forks a child and returns its output as a string. Notice that the `run/string*` procedure needs automatic storage management in order to reclaim *two* different OS resources: the child's process id (which is not revealed by the `run/port*` procedure), and the parent's end of the parent/child I/O pipe (which is not revealed by the `run/string*` procedure).

A programmer had written a loop in a program that created a time-stamp string on each pass through the loop. The time stamp was made by using the `run/string*` procedure to capture the output of the `date` program:

```
(let loop ()
  (let ((ts (run/string*
              (λ () (exec "/bin/date")))))
    ...)
  (loop))
```

Unfortunately, because the program never waited for the dead children, they inexorably filled up the kernel's process table. After twenty or so iterations, the program would fail by being unable to fork new children.

The programmer couldn't be advised to insert a `wait` call into his code, since the actual process id was hidden down inside the `run/string*` procedure. He couldn't simply wait for any child, since that might cause him to accidentally wait on a child process upon which he intended to wait at a later time. Changing `run/string*` and `run/port*` to export the child's process id would complicate the interfaces, damage simple composability, and violate modularity.

The solution was to redesign scsh so that integer process ids were replaced with an abstract data structure, the *process object*. Process objects are returned by the `fork` procedure, passed to the `wait` procedure, and finally reclaimed by automatic garbage collection. Now we are free to wait on a process zero, one, or more times:

- *nwaits* < 1
  A process object is garbage collected by the scsh run-time system after all pointers to it are dropped and the child process exits. The garbage collector is also free to perform a `wait` system call on the dead child, causing it to be deallocated from the kernel's process table.

- *nwaits* > 1
  The scsh run-time system caches the exit status of the child process within its process object on the first wait. Subsequent waits use the cached value.

This is modular.

Because a child process' exit status is saved within the child's process object, the scsh run-time system is free to eagerly wait on a process any time between the child's death and the first time the user requests it. This is called *reaping* the process, and it has two benefits:

- It frees up the child's entry in the kernel's process table.
- It makes the process object eligible for GC.

What we have really done is migrate an OS resource (process table entries) from the kernel into the scsh process, which makes them eligible for automatic reclamation.

4 Managing external signals

Our final example concerns handling the reception of external, asynchronous signals in scsh programs.[1] Handling signals in Unix implies constructing a concurrency model—asynchronous signals are essentially a mechanism for notifying a process of the actions of an external agent, such as the user striking the interrupt character on his keyboard, or a modem line being dropped by a remote program. {Note Synchronous exceptions}

It was an early design decision in scsh's concurrency model not to provide interrupts. Rather, scsh provides only serial threads. The interrupt-handler model was not provided because the interrupt model has several problems. First, interrupts confound several distinct computational elements, *e.g.*, concurrency, mutex, and communication. It is better to factor these elements into their individual components, allowing the programmer to use only those which are needed by the given application.

[1] The mechanisms described in this section are not available in the currently released versions of scsh.

Second, the interrupt model doesn't parallelise—it is fundamentally a uniprocessor mechanism. Because the main process is suspended while an interrupt handler runs, there is a mutex "built in" to the handler. This means that if the program is executed on a parallel processor, the system can't run the handler in parallel with the main program—to do so might violate some assumption of mutex made by the programmer.

Third, interrupt handlers are not modular. If one module of code installs a handler for an interrupt, it will clobber the handler installed by any other module. Global resources are always a potential source of conflict.

Finally, there is the argument of economy of mechanism: the thread model functionally dominates the interrupt model, so it is superfluous to provide both mechanisms.

In the scsh serial-thread model, if a program wishes to attend to some set of external signals, it blocks and waits for an occurrence of a signal. If it wishes to compute concurrently while attending to the external signals, it spawns a sub-thread to wait for the signal and performs the concurrent computation itself. As a simple example, let us suppose we wish to perform some computation while simultaneously printing "Ouch!" every time the user strikes the interrupt key on his keyboard. We introduce a blocking operator

(`wait-signal` *signal-set*) $\longrightarrow$ *signal*

that causes a thread to block until a signal from the specified set occurs. Our application would then contain the following code:

```
;;; The singleton set SIGINT:
(define ctl-c (signal-set signal/int))

(fork-thread (λ () (let loop ()
                     (wait-signal ctl-c)
                     (display "Ouch!")
                     (loop))))
```
...perform main computation...

In this model, interrupts are no longer necessary, interrupt handlers are no longer a global resource, and there is no spurious serialisation or mutex implied between the handler and the main computation—they are completely concurrent. However, our example exposes a race condition in the design: only signals that arrive while the sub-thread is blocked waiting in its `wait-signal` call get caught. If a signal arrives while the handler thread is actively processing a prior signal, it will be dropped.

We can fix this in our concurrency model by introducing a new data type, the *event*, that represents a single occurrence of some signal. An event has two fields: the Unix signal that occurred, and a pointer to the next event that occurred. That is, events are kept in a linked list in increasing-time order. The scsh run-time system keeps track of the most recent event; whenever a signal is delivered to a scsh process, the run-time system catches the signal, constructs a new event structure for it, and links it onto the tail of the list (fig. 1). Given this ordering, an event e provides an index into the time-line of signal events. Note that if no pointers to other events are retained by the user program, events are GC'd away as they occur, as the scsh run-time system only retains a pointer to the most recent event, *i.e.*, the *end* of the event queue. This value is exported to user code by the `most-recent-event` procedure.

Now we modify our `wait-signal` procedure to take a second argument, which is an event that serves as an index into the portion of the event queue in which we are interested:

(`wait-signal` *signal-set event*) $\longrightarrow$ *event*

This procedure is now defined to block until an event whose signal is a member of *signal-set* occurs—but only events occurring after the *event* parameter are eligible. The occurring event is returned as the value of the procedure call.

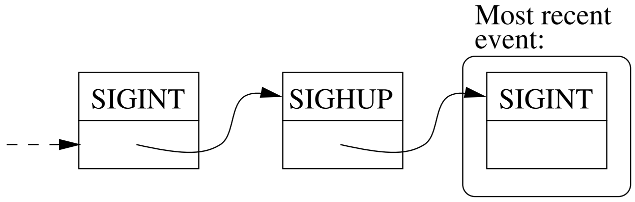

Figure 1: The event queue

The implementation is simple: `wait-signal` simply starts with *event*, and scans down the event chain searching for an event that satisfies the *signal-set* argument. If it finds one, it returns it immediately, without blocking. If it reaches the tail of the chain without finding an acceptable event, the thread is blocked by the thread scheduler; when the proper signal arrives, it will be made runnable again.

Our previous example can now be rewritten to use events to avoid the race condition:

```
;;; Catch all signals:
(fork-thread
  (λ ()
    (let loop ((last (most-recent-event)))
      (let ((event (wait-signal ctl-c last)))
        (display "Ouch!")
        (loop event)))))
```

If we wish to discard pending signals while processing a signal, we simply drop the triggering event after handling it:

```
;;; Discard pending signals:
(fork-thread
  (λ ()
    (let loop ()
      (wait-signal ctl-c (most-recent-event))
      (display "Ouch!")
      (loop))))
```

What our event mechanism does is give an on-demand event-buffering policy. Because retaining a pointer to an event prevents subsequent events from being reclaimed by the GC, the system only retains as many events as the program requires. That is, the garbage collector mediates the amount of signal queuing required by an application.

This signal-handler model is not without certain limitations. It can cause the system to retain large amounts of storage. If a program retains a pointer to an event that is never freed, then all subsequent events will be retained by the system. This means that the storage consumption of the program is dependent upon the behaviour of external agents—the signal generators. Hence this model is most likely not appropriate for systems where signals occur at extremely high rates. We would not want, for instance, to have an event occur upon each rotation of the computer's disk. However, for typical Unix programs, signal rates should be low enough to make this model practical.

This event model requires all events to be totally ordered. This could be a central bottleneck for a system with a large number of events occurring at high rates in a distributed machine. Again, for simple Unix signals, it should not be a problem.

5 Conclusion

We've seen how the scsh dialect of Scheme provides automatic resource management for three different operating-system resources. The techniques used to reflect these resources into the programming language are novel.

- With I/O channels, scsh provides mechanisms for breaking the abstraction in controlled ways that do not hinder the management of the simple, abstract uses of the resource. This design is the first of which I am aware to provide both of these features, in any programming language.
- With processes, scsh uses techniques to migrate process-table entries into the scsh process to make them eligible for automatic storage management. Again, scsh's design is novel, the first to provide the benefits of automatic resource management to managing the OS's process resources.
- With signal events, scsh provides a fully interrupt-free concurrency model, and allows the GC to mediate the amount of event queuing required by an application according to the dynamic needs of the program. The facilities traditionally provided by interrupt handlers in other programming languages are provided by an alternative mechanism which is modular and parallelisable.

While other operating-system and language designs have provided interfaces to OS resources from advanced languages, such as the Spin project's use of Modula-3 [Spin], or the Standard ML of New Jersey's interface to Posix [SML/NJ], none of these systems have managed to exploit automatic storage management to manage the operating system's resources in this manner.

Although we've considered Scheme and three different Unix resources as our examples, the ideas behind these examples are not specific to Scheme or Unix. Rather than use Scheme, we could just as easily use any modern programming language with abstract data types and automatic resource management, such as ML or Dylan. Nor are we confined to the bounds of Unix; the problems and limitations of explicit resource management and non-abstract data types named by small integers are commonly found in today's operating systems. The techniques presented here can be adapted to these systems as well.

Looking beyond current operating systems, we might consider how the lessons learned in providing for automatic management of Unix's resources could be applied to the design of new operating systems. For example, rather than resort to the specialised trickery we employed in the process-object case to migrate OS resources into user space to enable GC, we might consider how to make all the OS's resources directly visible to automatic management policies. As another example, we are using the insights derived from scsh's signal-handler system to guide the design of an ML-based OS kernel being constructed at MIT.

The programming language determines the programmer's view of the operating system. So in some sense, the language *is* the OS. The message we can take away is that the principles of modern programming languages can be directly applied to the view they provide of the machine and operating-system resources.

References

[R4RS] J. Rees and W. Clinger (editors). The revised[4] report on the algorithmic language Scheme. *Lisp Pointers* IV(3):1-55, July–September 1991.

[scsh] Olin Shivers. A Scheme shell. To appear in the *Journal of Lisp and Symbolic Computation*. (Also available as technical report TR-635, Laboratory for Computer Science, MIT; and technical report TR-94-10, Department of Computer Science, University of Hong Kong.)

[scsh-man] Olin Shivers. *The scsh manual.* November 1995, scsh release 0.4. MIT Laboratory for Computer Science. (Also available as URL `ftp://www-swiss.ai.mit.edu/pub/su/scsh/scsh-manual.ps`.)

[SML/NJ] The Posix API for SML/NJ is currently available only in a pre-release form; its manual also is not available. Interested parties may, however, contact Dr. John Reppy, AT&T Laboratories, `jhr@research.att.com`.

[Spin] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian Bershad. Language support for extensible operating systems. In Workshop on Compiler Support for System Software, February 1996. (Also available as URL `http://www.cs.washington.edu/research/projects/spin/www/papers/WCS/language.ps`)

[Wilson] Paul R. Wilson. Uniprocessor garbage collection techniques. In International Workshop on Memory Management, St. Malo, France, September 1992. (The Proceedings has been published as Springer-Verlag Lecture Notes in Computer Science no. 637. The individual paper is also available as URL `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`.)

Notes

{Prescient designers}

Beyond the perhaps generous observation that the Unix system designers appear to have anticipated the current fashion in linear languages by several decades.

{Synchronous exceptions}

Synchronous exceptions, such as floating-point overflow, or divide-by-zero traps, are fundamentally different entities from asynchronous interrupts, and are better handled by different linguistic mechanisms. Scsh provides interfaces to these exceptions using dynamically-scoped exception handlers. We will not consider them further.

{Unix system calls}

Here is a brief overview of the Unix system calls relevant to sections 2–4.

A disk file is opened by applying the `open` system call to a file name, returning an integer "file descriptor" that names the open file. After using this integer in other system calls to read and write the file, the file is closed by passing the integer to the `close` system call. Closing a file also frees the file descriptor for reuse. A valid open file descriptor `fd` can be replicated with the `dup2(fd, new_fd)` system call, which causes file descriptor `new_fd` to refer to the same open file as `fd`. Similarly, the `dup(fd)` call allocates the smallest unused file descriptor, causing it to refer to the same open file as `fd`, and returns it.

By convention, file descriptors 0, 1, and 2 are taken by most programs to be the standard input, standard output, and standard error sources and sinks.

A child process is created with the `fork()` system call, which duplicates the calling process. The call returns in both the parent and child process. In the parent process, it returns the child's non-zero process id; in the child, it returns zero. The child's process state, such as its open file descriptors, is a duplicate of the parent's. A process (typically a newly-forked child process) may overlay itself with a new program by applying the `exec` system call to the file name of the program to be executed, *e.g.*, `exec("/bin/ls")`. The `exec` call does not return; instead the process commences executing the named program in a fresh address space. Process state, such as open file descriptors, is preserved across calls to `exec`. A process terminates by calling the `exit(status)` system call, where `status` is a small integer code. The process's parent may synchronise with the process' termination by calling the `wait(child_pid)` system call, which blocks until the child terminates, returning the child's exit status.

The typical scenario for using these primitives is for a program to call `fork`, then in the child process to arrange the child's process state as it wishes (*e.g.*, setting the standard I/O file descriptors to any desired redirections), and then to invoke the desired child program with an `exec` call.