Safe Asynchronous Exceptions For Python*

Williams College Technical Note 02-2002, Dec. 2002

Stephen N. Freund
Department of Computer Science
Williams College
Williamstown, MA 01267

Mark P. Mitchell
CodeSourcery, LLC
9978 Granite Point Ct.
Granite Bay, CA 95746

Abstract

We demonstrate that the Python programming language is not signal-safe, due to Python's support for raising exceptions from signal handlers. We examine the approaches that various languages have used when dealing with the combination of asynchrony and exception handling, and propose a modification of the Python language that restores signal safety.

Keywords: programming languages, exceptions, asynchronous signals, resource management, Python

1 Introduction

Asynchrony is a fact of life in computer systems. Many programs must handle asynchronous events, such as keyboard interrupts, timer alarms, and sensor activity, to function properly. However, these events occur at unpredictable points in time.

Python employs the *signal* model to capture asynchronous events. Signals are used in both the UNIX operating system and the C programming language. When the operating system *raises* a signal, control is transferred to a *signal handler*, i.e., a special purpose routine designed to process the asynchronous event. Because the signal handler executes asynchronously, it can make only very limited assumptions about the integrity of data structures being manipulated by the remainder of the program.[1] When the signal handler returns, the main program resumes from the point at which it was interrupted.

Programmers often want to treat signals in the same way that they treat error conditions. For example, a program may recover from an error writing to an open file and process a "keyboard interrupt" signal raised during the write in much the same way. As we describe below, Python combines error handling with signal handling to enable them to be treated uniformly.

Python, like many recent languages, uses *exceptions* to deal with error conditions and other similar problems. Exceptions are non-local gotos that transfer control from the point at which the exception is *raised* to an *exception handler*. Exception handlers are dynamically scoped; control is transferred to the innermost handler of the appropriate type on the current call stack. Exceptions enable errors to propagate without the use of cumbersome and error-prone return codes. Exceptions also allow the authors of library modules to defer the handling of errors to clients, under the assumption that the client application is best able to decide how to deal with the error condition.

To treat signals as exceptions, Python provides a way to raise an exception within a signal handler, thereby turning an asynchronous signal into an *asynchronous exception*. Once raised, asynchronous exceptions are handled using the exception handler mechanism for normal (synchronous) exceptions.

Unfortunately, the Python language's transformation of signals into asynchronous exceptions is currently not robust. Subtle race conditions actually make it impossible to write reliable code that handles signals in a timely fashion. We propose a small modification to the Python programming language to correct this problem in a satisfactory way. Our extension, following the suggestions in [4] for Haskell, introduces statements that block and unblock asynchronous signal delivery in their dynamic scope. These statements are 1) a simple extension to the Python language, 2) easy to implement, and 3) intuitive.

The need for more a reliable asynchronous exception mechanism is not merely theoretical. One of the authors uncovered the problem while trying to build a robust software tool using Python. In addition, the Python standard library itself contains several hundred exception handlers to manage various resources, and all of them may behave incorrectly in the presence of asynchronous exceptions.

After showing how our extension solves the demonstrated problem in an elegant way, we give an overview of other ways in which various programming languages have combined signals and exceptions.

*This work was presented at the Lightweight Languages Workshop, Nov. 2002.

[1] For example, in C, it is not even valid to use `printf` in a signal handler.

2 Asynchronous Exceptions and Python

As we demonstrate in this section, Python does not enable programmers to write robust code in the presence of asynchronous exceptions. The existing language features to handle exceptions suffer from subtle race conditions when asynchronous exceptions are generated in signal handlers. This is a known pitfall that has also appeared in other languages [8, 4].

Python models signal handling similar to the C programming language. In particular, programs may register signal handlers (i.e., Python functions) that are then called when signals occur. For example, the following Python program installs the `handle_alarm` function as the handler for the SIGALRM signal.

Figure 1: Signal Handlers

```
def handle_alarm(signum, frame):
  print "received alarm signal"

signal.signal(signal.SIGALRM, handle_alarm)

...
```

The Python interpreter will call this function every time the operating system generates a SIGALRM notification.

The Python interpreter (written in C) implements this functionality by setting a flag when a signal occurs. Then, when iterating through the main byte code execution loop, the interpreter notices that the flag has been set, and begins interpreting the code that makes up the signal handler, as if the next bytecode instruction had been a subroutine call to the signal handler. Thus, the Python interpreter partially synchronizes the signal. While the signal handler executes asynchronously with respect to the main Python program, it executes synchronously with respect to the Python interpreter, i.e., it does not execute as part of a C signal handler.

Python explicitly permits signal handlers to raise exceptions. For example, we could replace the `print` statement in Figure 1 with `raise TimeOutError` to raise an exception when the signal occurs. Also, Python's default signal handler for `SIGINT` (the interrupt signal) raises the `KeyboardInterrupt` exception. This translation of asynchronous signals into exceptions is convenient; programmers can handle keyboard interrupts as they would "file not found" or "permission denied" errors.

As a result, however, Python programs must anticipate the fact that exceptions can be thrown *at any point*, i.e., exceptions are now asynchronous events. Common practice is to surround code in which the programmer wants to handle asynchronous exceptions with a Python exception handler. However, this leads to a number of race conditions, which are exemplified by the following code:

Figure 2: Python Race Condition

```
f = open(file_name)
try:
  s = f.read()
finally:
  f.close()
```

The semantics of Python's `try-finally` construct are that the code in the `try` block is executed and then, whether the `try` block is executed normally or via an exception, the `finally` block is executed. This construct is intended to be used, as shown above, to manage resources robustly. This code attempts to ensure that the file `f` will be closed even if an error occurs when reading from the file. (The `read` function raises an exception if data cannot be read.) Idioms similar to this code are used to handle many different resources, including mutex locks, network connections, and so on.

However, in the presence of signals, this code does not perform as intended. In particular, consider the case in which a signal occurs immediately after the call to `open`, but before the interpreter has begun to execute the `try` block. If the signal handler throws an exception, the `finally` block is never executed. Similarly, consider the case in which a signal occurs just as the interpreter enters the `finally` block, but before the call to `close`. If the signal handler throws an exception, the interpreter will never call `close`. In both cases, the program will leak the open file.[2]

2.1 Attempted Solutions and Their Pitfalls

There are no adequate solutions to avoid these problems. To illustrate the subtle interactions between signals and exceptions, we describe two suggested programming practices to prevent race conditions and demonstrate their weaknesses.

One approach is to explicitly block signals before the call to `open` and unblock them after the call to `close`. Although Python does not provide signal-blocking and signal-unblocking routines in its library, it is easy to write a C extension module that provides this functionality:

Figure 3: Python Race Condition

```
block_all_signals()
f = open(file_name)
try:
  s = f.read()
finally:
  f.close()
unblock_all_signals()
```

[2]When Python garbage collects the file object, it will close the file, but there is no guarantee that this will happen in a timely fashion. In general, the resource might be one that Python will never deallocate.

While this approach does indeed avoid the race conditions described above, it does not solve the problem in a satisfactory way. In particular, if the body of the `try` block contains a long-running computation, asynchronous events may be blocked for an arbitrarily long period of time, making the program unacceptably non-responsive. Also, note that the call to `open` is now made when signals are turned off. If the file is on a network file system, and cannot be opened quickly, the user can no longer interrupt the program. By moving the calls to different locations, we can improve responsiveness—but only at the cost of robustness.

Another approach is to explicitly reset the signal handler to a routine that will never throw an exception. Instead, the signal handler would queue the exception. Later, at an appropriate point, the exception could be thrown. This approach is not easy to get right, even if it can be made to work. Signals must be blocked while the signal handlers are being replaced, some provision must be made to deal with the case in which signal handlers are changed explicitly by code in the middle of the `try` block, and it is difficult to handle the possibility of multiple signals arriving during the `try` block.

Perhaps more importantly, both approaches suffer from the fundamental defect that the programmer must explicitly remember to insert code to block and unblock signals, or to register and unregister signal handlers. We also believe that requiring the programmer to match the two separate operations correctly is dangerous. Other paired operations, such as lock acquire and release, are frequently used incorrectly, and we wish to avoid adding a similar feature here.

3 Blocking and Unblocking Asynchronous Exceptions

We now present a simple language extension to enable safe use of asynchronous exceptions in Python programs. Our proposal is based on features explored in other programming languages, particularly in the context of the Haskell programming language [4]. We explore the full design space for handling asynchronous exceptions in the next section.

Our proposed modification to Python extends the syntax with two additional scoped constructs—`block` and `unblock`. The following program demonstrates the constructs:

Figure 4: `block` and `unblock`

s_0
`block:`
  s_1
  `unblock:`
    s_2
    s_3
  s_4

The `block` and `unblock` statements disable and enable asynchronous signal delivery in their dynamic scope. Thus, no asynchronous exceptions will occur during the execution of s_1 or s_4, but they may occur during the execution of s_0, s_2, and s_3. The `block` and `unblock` statements can be arbitrarily nested with the obvious meaning. Using `block` and `unblock`, we can rewrite the code in Figure 2 to ensure that the file resource is never left open while still enabling asynchronous exceptions while using the file:

Figure 5: Revised Example

```
block:
  f = open(file_name)
  try:
    unblock:
      s = f.read()
  finally:
    f.close()
```

With these extensions, the Python run time will deliver asynchronous exceptions only when control is in the `unblock` statement. Thus, the race conditions described above are eliminated. However, long-running computations in the `unblock` block can still be interrupted by an asynchronous event, and signals can still be transformed into exceptions as appropriate. If `open` may take a long time to complete, it too could be rewritten to use `unblock` to ensure that the user could interrupt the call at appropriate times.

Note that the operators nest naturally and match our intuition about when we would like to prevent asynchronous exceptions from occurring.

3.1 Implementation

As described in Section 1, the Python interpreter queues signals and only delivers them to the interpreted program at the top of the interpreter's main loop. This machinery makes it easy to embed implementations of the `block` and `unblock` statements in the Python run time with no measurable memory or performance penalty.

Activation records in the standard Python interpreter contain a stack of information about exception handlers in the dynamic scope of execution. We extended the interpreter to record entries and exits for `block` and `unblock` statements on this stack as well. We also modified the interpreter to deliver signals at the top of the main loop only after inspecting this stack for the current activation record and determining that signals are not currently blocked. If signals are blocked, the interpreter delays delivery until they become unblocked. When a function is called, the interpreter pushes onto the new activation record's stack an indication of whether signals are currently blocked in the caller. This allows us to avoid searching outside the current activation record when determining if signals are blocked.

3.2 Higher-level Extensions

The `block` and `unblock` statements are relatively low-level constructs, and we suggest providing "syntactic sugar" for common, higher-level programming idioms. Resource acquisition, usage, and release is one such idioms, and extending Python to better handle this common case is straightforward once `block` and `unblock` have been implemented. One possible way to provide support for this feature is with the following extension:

Figure 6: Resource Management

```
initially:
  f = open(file_name)
try:
  s = f.read()
finally:
  f.close()
```

This extension is merely syntactic sugar for the example above. The virtual machine blocks signal delivery during the `initially` and `finally` fragments, but not the `try` code fragment. All exits from the `try` section, exceptional or not, will execute the `finally`. We believe that most code in `initially` and `finally` blocks will run quickly and that latency in processing signals will therefore not be unduly affected. Nested `block` and `unblock`, or `initially-try-finally`, statements can provide better fine-grained control over asynchronous exception delivery when more subtle programming patterns are employed.

As illustrated in [4], the proposed language extension can also easily model many other idioms involving asynchronous signals or communication as well, including parallel logical-or and timeouts. Clearly, features must be added to a language with care, and it remains to be seen which combination of low-level and high-level constructs for dealing with asynchronous exceptions are most appropriate, given the overall design and intended use of Python.

3.3 Interruptible Operations

While the presented features are sufficient to prevent the types of race conditions that are of greatest concern to us, they still do not prevent all potentially bad behaviors. For example, an exception could be generated after the call to `f.read` completes but before the return value is assigned to `s`. In that case, the value just read would be lost. As in the Haskell, we can address this problem by permitting some low-level blocking operations, such as `read` to be interruptible, even when inside a `block` statement. With this extension, one could write the following safer version of the code:

Figure 7: Interruptible `read` Operation

```
block:
  f = open(file_name)
  try:
    s = f.read()
  finally:
    f.close()
```

Either the call to `read` will be interrupted before any input is actually read, or it will complete, in which case we are guaranteed not to be interrupted before assigning the result to `s`. Our initial prototype does not support interruptible operations, although adding them would be trivial. The more difficult issue would be identifying which operations in the Python libraries should be interruptible in this way, and annotating them as such.

4 Related Work

We have based our Python language extension on the scoped `block` and `unblock` combinators designed for Concurrent Haskell [4]. In some sense, the purely functional nature of Haskell makes `block` and `unblock` the ideal primitives. It is easy and natural to use the combinators in more complex ways to support higher-level idioms. We believe that these operations are appropriate for use in Python as well. They are a simple extension to the Python language, have a straightforward and efficient implementation, and are flexible enough to capture the most common idioms used in large systems.

The treatment of asynchronous signals and exceptions in other languages covers a broad-spectrum of possible solutions. We highlight representative language features to demonstrate approaches, starting with the most restrictive designs. In addition to language architectures that support signals generated by the underlying operating system, we look at languages providing mechanisms to send signals between threads.

We begin with languages that prevent asynchronous generation of exceptions. For example, the C++ language specification forbids signal handlers from raising exceptions [11]. Therefore, the programs must handle signals by setting a global variable that can be polled later to determine which signals have occurred.

Other languages support this type of polling discipline directly. In Modula-2+ [9] and Modula-3 [6], one thread may send an asynchronous signal to another thread, but delivery is delayed until the receiver indicates interest in receiving notification by calling either `AlertWait` or `TestAlert`. The `TestAlert` procedure returns a boolean value indicating whether or not another thread signaled an `Alert` for the currently executing thread. When a thread calls `AlertWait`, it will block until an `Alert` is raised, which causes the run time to generate an exception to be handled by the blocked thread. In effect, the alert polling process turns synchronous signals into synchronous events. Another example of this technique appears in Scsh, a dialect of Scheme for systems programming [10].

To use this type of polling, it is necessary to include calls to the polling routines in all libraries and code segments that may be used in conjunction with asynchronous events. This requirement makes it difficult to reason locally about asynchrony and requires coordination throughout the entire program, which is a severe handicap when building large systems.

A similar polling technique is used in Java to interrupt sleeping threads. A thread can wake up a sleeping thread by invoking the `interrupt` method on it. The method invocation generates an `InterruptedException` exception in the suspended thread, which is then caught with a standard Java exception handler. If `interrupt` is invoked on a running thread, no exception is generated. Instead, a flag that can be tested later is set to indicate the occurrence of the asynchronous interruption. Libraries providing thread support, most notably the POSIX threads interface (accessible from C and many other languages), often define polling mechanisms to handle thread cancellation [7].

Java originally provided methods to suspend, resume, and stop threads asynchronously. However, these features presented a number of significant safety problems [12]. For example, programs that suspend threads while they hold locks became prone to deadlock, and stopping threads outright prevents them from cleaning up any resources and ensuring that data structures are left in a consistent state. Sun deprecated these features after the initial release of Java. Although suspend and resume operations appear in other run-time architectures, such as the Microsoft .NET framework thread model and Ada tasks [2], separate threads cannot interact via asynchronous exceptions.

The definition of Standard ML [5] originally included an `Interrupt` exception that was generated on the SIGINT signal, which indicates a user interrupt. However, the designers removed it because of the difficulty modeling this behavior formally and problems similar to those illustrated in Section 2. SML/NJ introduced a more general signal handling mechanism [8] in its place. When a signal is received in the new scheme, the run time creates a continuation out of the currently executing thread. The continuation is then passed to the signal handler (which runs with further signals disabled), and the handler may either resume the original continuation or transfer control to different thread after it has finished its own processing. Although this technique provides some of the benefits of scoped `block` and `unblock` constructs, the reliance on continuations makes it inappropriate for Python.

Several Lisp dialects also provide support for limited asynchronous signals between concurrent threads, primarily to allow speculative execution. In PaiLisp [3], one thread may force a different thread to execute a continuation. In addition, QLisp provides a heavyweight mechanism that allows easy destruction of a whole tree of related threads, but this mechanism is too costly to use in the types of programs considered here [1].

5 Summary

Asynchrony creates subtle, but notoriously complex, problems for programmers. Programming languages can provide a foundation for tackling these problems via simple and effective features that handle asynchronous events reliably. In this paper, we have identified one aspect of Python where this is currently not the case, and we propose a straightforward extension to the language to enable safe use of asynchronous exceptions. We hope that the Python developers will seriously examine the situation and adopt either our proposed solution or another solution with similar properties.

The same (or closely related) problems with asynchronous exceptions have surfaced in many languages, including Haskell, ML, and Java, and we have shown how these other languages have attempted to provide safe asynchronous features. In addition to improving Python, we hope that this paper encourages language designers to deal with the subtle interaction between signals and exceptions more proactively in the next generation of languages.

Acknowledgments

We thank Mike Burrows, John Mitchell, and the anonymous reviewers for comments on a draft of this paper.

References

[1] R. P. Gabriel and J. McCarthy. Queue-based multiprocessing lisp. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 25–44, 1984.

[2] International Organization for Standardization. *Ada 95 Reference Manual.* January 1995.

[3] T. Ito and M. Matsui. A parallel lisp language pailisp and its kernel specification. In *Proceedings of the US/Japan workshop on Parallel Lisp on Parallel Lisp: languages and systems*, pages 58–100. Springer-Verlag New York, Inc., 1990.

[4] S. Marlow, S. L. Peyton Jones, A. Moran, and J. H. Reppy. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[5] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[6] G. Nelson. *System Programming in Modula-3.* Prentice Hall, 1991.

[7] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming.* O'Rielly, 1996.

[8] J. H. Reppy. Asynchronous signals in Standard ML. Technical Report Technical Report TR90-1144, Cornell University, August 1990.

[9] P. Rovner, R. Levin, and J. Wick. On extending Modula-2 for building large, integrated systems. Research Report 3, Digital Equipment Corporation Systems Research Center, 1985.

[10] O. Shivers. Automatic management of operating-system resources. In *Proceedings of the ACM International Conference on Functional Programming*, pages 274–279, 1997.

[11] B. Stroustrop. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[12] Sun Microsystems. Why are Thread.stop, Thread.suspend, Thread.resume, and Runtime.runFinalizersOnExit deprecated?, 2002. At `http://www.javasoft.com/`.