Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T24 Event-Based Concurrency

Referência principal
Ch.33 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 24 de outubro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Modelos alternativos já eram assunto há 40 anos. . .

Programming Languages — T.A. Standish, Editor

Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra
Burroughs Corporation

So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be be shown.

Key Words and Phrases: programming languages, sequencing primitives, program semantics, programming language semantics, nondeterminacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology
CR Categories: 4.20, 4.22

Author's address: Burroughs, Plataanstraat 5, Nuenen—4565, The Netherlands.

1. Introduction

In Section 2, two statements, an alternative construct and a repetitive construct, are introduced, together with an intuitive (mechanistic) definition of their semantics. The basic building block for both of them is the so-called "guarded command," a statement list prefixed by a boolean expression: only when this boolean expression is initially true, is the statement list eligible for execution. The potential nondeterminacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact that programs can now be derived in a manner more systematic than before.

In Section 3, after a prelude defining the notation, a formal definition of the semantics of the two constructs is given, together with two theorems for each of the constructs (without proof).

In Section 4, it is shown how, based upon the above, a formal calculus for the derivation of programs can be founded. We would like to stress that we do not present "an algorithm" for the derivation of programs: we have used the term "a calculus" for a formal discipline—a set of rules—such that, if applied successfully: (1) it will have derived a correct program; and (2) it will tell us that we have reached such a goal. (We use the term as in "integral calculus.")

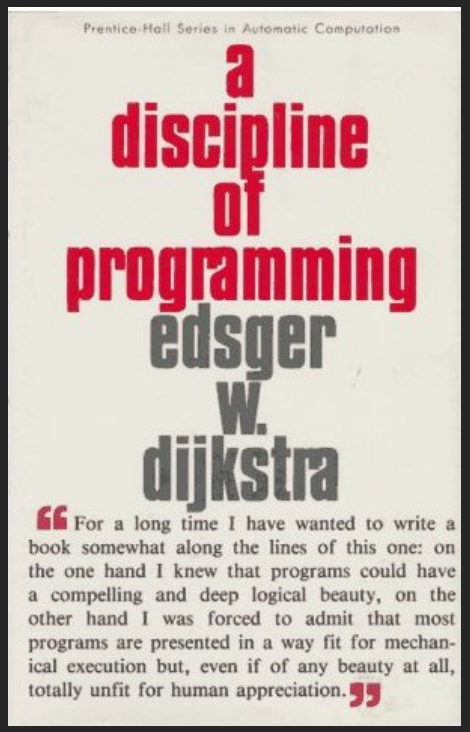
2. Two Statements Made from Guarded Commands

If the reader accepts "other statements" as indicating, say, assignment statements and procedure calls, we can give the relevant syntax in BNF [2]. In the following we have extended BNF with the convention that the braces {...} should be read as "followed by zero or more instances of the enclosed."

⟨guarded command⟩ ::= ⟨guard⟩ → ⟨guarded list⟩
⟨guard⟩ ::= ⟨boolean expression⟩
⟨guarded list⟩ ::= ⟨statement⟩ {; ⟨statement⟩}
⟨guarded command set⟩ ::= ⟨guarded command⟩ {[] ⟨guarded command⟩}
⟨alternative construct⟩ ::= **if** ⟨guarded command set⟩ **fi**
⟨repetitive construct⟩ ::= **do** ⟨guarded command set⟩ **od**
⟨statement⟩ ::= ⟨alternative construct⟩ | ⟨repetitive construct⟩ | "other statements"

The semicolons in the guarded list have the usual meaning: when the guarded list is selected for execution its statements will be executed successively in the order from left to right; a guarded list will only be

Communications of the ACM — August 1975, Volume 18, Number 8

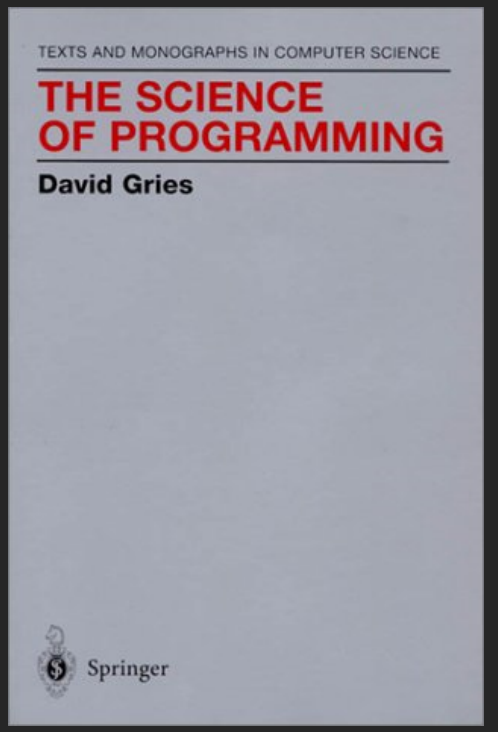


Prentice-Hall Series in Automatic Computation

a discipline of programming

edsger w. dijkstra

"For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation."



TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

THE SCIENCE OF PROGRAMMING

David Gries

Springer

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (Aug. 1975), 453-457.

Modelos alternativos já eram assunto há 40 anos. . .

On the Duality of Operating System Structures

- Hugh C. Lauer
  - Adjunct Prof., Worcester Polytechnic Institute
  - Xerox, Apollo Computer, Mitsubishi Electronic Research Lab, etc.
  - Founded a number of businesses:
    Real-Time Visualization unit of
    Mitsubishi Electric Research Labs (MERL)

- Roger M. Needham
  - Prof., Cambridge University
  - Microsoft Research, Cambridge Lab
  - Kerberose, Needham-Schroeder security protocol, and key exchange systems

Modelos alternativos já eram assunto há 40 anos. . .

Message vs Procedure oriented systems (i.e. Events vs Threads)

- Are they really the same thing?
- Lauer and Needham show
  - 1) two models are duals
    - Mapping exists from one model to other
  - 2) dual programs are logically identical
    - Textually similar
  - 3) dual programs have identical performance
    - Measured in exec time, compute overhead, and queue/wait times

Modelos alternativos já eram assunto há 40 anos. . .

SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (Welsh, 2001)

- 20 to 30 years later, still controversy!

- Analyzes threads vs event-based systems, finds problems with both
- Suggests trade-off: stage-driven architecture
- Evaluated for two applications
  - Easy to program and performs well

Modelos alternativos já eram assunto há 40 anos...

SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (Welsh, 2001)

- Matt Welsh
  - Cornell undergraduate Alum (Worked on U-Net)
  - PhD from Berkeley (Worked on Ninja clustering)
  - Prof. at Harvard (Worked on sensor networks)
  - Currently at Google
- David Culler
  - Faculty at UC Berkeley
- Eric Brewer
  - Faculty at UC Berkeley (currently VP at Google)

Event-Based Concurrency

- As we have seen, it is not easy to generate a thread-safe concurrent application using the traditional procedural programming style.
  - For example, missing locks, deadlocks and other problems can easily arise.
- Event-based concurrency is another style of concurrent programming which has become popular in some modern systems.
  - For example, it is used in GUI-based applications, some types of internet servers, and server-side frameworks like **node.js**.
- **The idea is** to replace the complexities of multi-threading with **a deterministic, sequential model** that is easier to understand and debug, while giving the developer greater control over what is scheduled at a given moment in time.

Can we build concurrent systems without threads?

- We would like to allow a single-threaded program to cope with high-latency I/ O devices by overlapping I/ O with processing and other I/ O.

  Thus,

  - Retaining control over concurrency
  - Avoiding some of the problems common to multi-threaded applications
- How could we build a concurrent server like that?

The Basic Idea: An Event Loop

- The basic approach is called **event-based concurrency**:
  - Wait for something (i.e., an "**event**") to occur.
  - When it does, check what type of event it is.
  - Do the small amount of work it requires.
- Example:

```
1.  while (1) {
2.      events = getEvents();
3.      for (e in events)
4.          processEvent(e);
5.  }
```

- While the event handler processes an event, this is the only activity taking place in the system.
- How can an event handler tell whether a message has arrived for it?

How to detect the occurrence of events

- Most systems provide a basic API, including `select()` or `poll()` system calls.
- They check whether there are incoming I/O requests to be serviced.
- They enable the developer to build a non-blocking event loop, which checks for incoming packets and reads from sockets with messages and replies as needed.
- With a single CPU and an event-based application, the problems found in concurrent programs are no longer present.
  - Since the event-based server cannot be interrupted by another thread, there is no need to acquire or release locks.
  - Thus, concurrency bugs common in multithreaded programs do not appear in the basic event-based approach.

Simple code using select()

```
1.  int main(void) {
2.      // open and set up a bunch of sockets (not shown)

3.      // main loop
4.      while (1) {
5.          // initialize the fd_set to all zero
6.          fd_set readFDs;
7.          FD_ZERO(&readFDs);

8.          // now set the bits for the descriptors
9.          // this server is interested in
10.         // (for simplicity, all of them from min to max)
11.         for (int fd = minFD; fd < maxFD; fd++)
12.             FD_SET(fd, &readFDs);

13.         // do the select
14.         int rc = select(maxFD + 1, &readFDs, NULL, NULL, NULL);

15.         // check which actually have data using FD_ISSET()
16.         for (int fd = minFD; fd < maxFD; fd++)
17.             if (FD_ISSET(fd, &readFDs))
18.                 processFD(fd);
19.     }
20. }
```

What if an event leads to a system call that might block?

- In a thread-based server, while the thread issuing the blocking I/O request suspends (waiting for the I/O to complete), other threads can run.
- This natural overlap of I/O and other operations is what makes thread-based programming quite natural and straightforward.
- With an event-based approach, however, there are no other threads to run: just the main event loop.
  - This implies that if an event handler issues a call that blocks, the entire server will also block until the call completes.
  - When the event loop blocks, the system sits idle, and thus is a huge potential waste of resources.
- We thus have a rule that must be obeyed in event-based systems: no blocking calls are allowed.

Why Simpler? No Locks Needed

- The event-based server cannot be interrupted by another thread.
  - With a single CPU and an event-based application.
  - It is decidedly **single threaded**.
  - Thus, *concurrency bugs* common in threaded programs **do not manifest** in the basic event-based approach.

A Problem: Blocking System Calls

- What if an event requires that you issue **a system call** that might block?
  - There are no other threads to run: *just the main event loop*
  - The entire server will do just that: block until the call completes.
  - Huge potential waste of resources

In event-based systems no blocking calls are allowed.

A Solution: Asynchronous I/O

- Enable an application to issue an I/O request and return control immediately to the caller, before the I/O has completed.
  - Example:

```
struct aiocb {
    int aio_fildes; /* File descriptor */
    off_t aio_offset; /* File offset */
    volatile void *aio_buf; /* Location of buffer */
    size_t aio_nbytes; /* Length of transfer */
};
```

  - An Interface provided on *macOS*
  - The APIs revolve around a basic structure, the `struct aiocb` or **AIO control block** in common terminology.

A Solution: Asynchronous I/O (Cont.)

- Asynchronous API:
  - To issue an asynchronous read to a file

```
int aio_read(struct aiocb *aiocbp);
```

    - If successful, it returns right away and the application can continue with its work.

  - Checks whether the request referred to by `aiocbp` has completed.

```
int aio_error(const struct aiocb *aiocbp);
```

    - An application can **periodically pool** the system via `aio_error()`.
    - If it has completed, returns success.
    - If not, EINPROGRESS is returned.

A Solution: Asynchronous I/O (Cont.)

- Interrupt
  - Remedy the overhead to check whether an I/O has completed
  - Using **UNIX signals** to inform applications when an asynchronous I/O completes.
  - Removing the need to *repeatedly ask the system*.

Another Problem: State Management

- The code of event-based approach is generally more complicated to write than *traditional thread-based* code.
  - It must package up some program state for the next event handler to use when the I/O completes.
  - The state the program needs is on the stack of the thread. → **manual stack management**

Another Problem: State Management (Cont.)

- **Example** (an event-based system):

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

  - First **issue** the read asynchronously.
  - Then, **periodically check** for completion of the read.
  - That call informs us that the **read is complete.**
  - How does the event-based server know what to do?

Another Problem: State Management (Cont.)

- **Solution:** continuation
  - **Record** the needed information to finish processing this event *in some data structure*.
  - When the event happens (i.e., when the disk I/O completes), **look up** the needed information and process the event.

What is still difficult with Events.

- Systems moved from a single CPU to multiple CPUs.
  - Some of the simplicity of the event-based approach disappeared.
- It does not integrate well with certain kinds of systems activity.
  - **Ex. Paging**: A server will not make progress until page fault completes (implicit blocking).
- Hard to manage overtime: The exact semantics of various routines changes.
- Asynchronous disk I/O never quite integrates with asynchronous network I/O in as simple and uniform a manner as you might think.

ASIDE: Unix Signals

- Provide a way to communicate with a process.
  - *HUP* (hang up), *INT*(interrupt), *SEGV*(segmentation violation), and etc.
  - **Example**: When your program encounters a *segmentation violation*, the OS sends it a *SIGSEGV*.

```c
#include <stdio.h>
#include <signal.h>
void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

A simple program that goes into an infinite loop

ASIDE: Unix Signals (Cont.)

- You can send signals to it with the **kill command** line tool.
  - Doing so will *interrupt the main while loop* in the program and run the handler code `handle()`.

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```