Testing a Multiprogramming System[*]

(1973)

A central problem in program design is to structure a large program such that it can be tested systematically by the simplest possible techniques. This paper describes the method used to test the RC 4000 multiprogramming system. During testing, the system records all transitions of processes and messages between various queues. The test mechanism consists of fifty machine instructions centralized in two procedures. By using this mechanism in a series of carefully selected test cases, the system was made virtually error free within a few weeks. The test procedure is illustrated by examples.

1 Introduction

This paper describes the method used to test the RC 4000 multiprogramming system (Brinch Hansen 1970). The system was built with the following *test criteria* in mind:

1. A large program should be structured such that it can be tested by the simplest possible techniques.
2. The documentation of a large program should include a systematic set of reproducible test cases.

The nucleus of the RC 4000 system is an interrupt response program of 4,800 machine words called the *monitor*. The monitor multiplexes a single processor among concurrent processes and implements a set of procedures which these processes can call to create other processes and send messages to them. Monitor procedures are executed in a non-interruptable, privileged processor state; processes are executed in an interruptable, non-privileged state.

The monitor consists of five *programming layers* with the following tasks:

[*]P. Brinch Hansen, Testing a multiprogramming system, *Software—Practice and Experience 3*, 2 (April–June), 145–150. Copyright © 1973, John Wiley & Sons, Ltd.

processor multiplexing
message buffering
input/output operations
process creation and termination
file system

The layers were tested in that order starting with the bottom layer (processor multiplexing) working towards the top layer (file system).

2 Test Mechanism

The main difficulty in testing a (possibly erroneous) multiprogramming system is to prevent concurrent events from causing irreproducible, time-dependent test results.

During each test, the system was initialized with the monitor and a number of *test processes*. The simplest idea would have been to let a test process first call a monitor procedure and then examine various monitor variables (such as process descriptions and scheduling queues) to decide whether the call had the intended effect. Unfortunately, this idea does not work in a multiprogramming environment in which other events (caused by processor multiplexing) may change the internal state of the monitor before the result of a given monitor call has been recorded by a test process.

To make a test event well defined and reproducible, the execution of a monitor call and the recording of its result had to be an indivisible event. Mutual exclusion of test events was achieved by letting the monitor output test data on a typewriter in the non-interruptable processor state.

The hardest problem was to select a minimal set of monitor events that would give significant information about its handling of concurrent processes. It turned out to be sufficient to record all *transitions* of *processes* and *messages* among various queues. When a list element (representing a process or a message) is removed from or linked to a list (representing a queue), the monitor outputs the addresses of the list element and the head of the list plus a single character to distinguish removal from linking. The meaning of these addresses is defined by the assembly listing of the monitor program.

In the following, such test output is represented by more readable lines of the form:

take element from queue
put element in queue

This test mechanism, which adds fifty machine instructions to the monitor, is centralized in two local procedures, *take* and *put*. The following sections describe how it was used to test the monitor.

3 Processor Multiplexing

The processor is shared cyclically among all active processes. Every 25 msec, a clock interrupt causes the monitor to preempt a running process in favor of another process ready to run (Fig. 1).

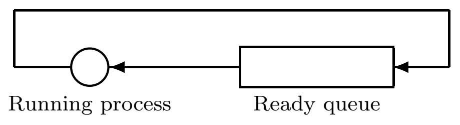

Figure 1 Processor multiplexing.

The monitor code for processor multiplexing can be represented as follows:

```
on clock interrupt do
begin
  put running process in ready queue;
  take another process from ready queue;
  continue that process;
end
```

To test processor multiplexing, the system was initialized with three processes P, Q and R in the ready queue (in that order). These processes cycled forever:

```
P: repeat until false;
Q: repeat until false;
R: repeat until false;
```

The clock was replaced by a manually operated *interrupt key.* During the test, the monitor produced the following output when processor multiplexing was correct:

```
    take P from ready queue
*   put P in ready queue
    take Q from ready queue
*   put Q in ready queue
    take R from ready queue
*   put R in ready queue
    take P from ready queue
    . . .
```

The lines marked $*$ are monitor responses to clock interrupts simulated by pushing the interrupt key.

As soon as processor multiplexing worked, the monitor procedures for process communication were tested.

4 Process Communication

Processes can exchange messages in buffer elements of fixed length stored within the monitor. A communication between two processes, S and R, takes place in four steps:

1. Process S sends a message M to process R in a buffer element B selected by the monitor by calling the procedure

   send message(R, M, B)

2. Process R receives the message by calling the procedure

   wait message(S, M, B)

3. Process R sends an answer A to process S in the same buffer element B by calling the procedure

   send answer(A, B)

4. Process S receives the answer by calling the procedure

   wait answer(A, B)

Figure 2 shows the life cycle of a buffer element. Available buffer elements are linked to a common *pool* within the monitor. The monitor also maintains a *message queue* for each process. A buffer element is linked to this queue when a message is sent to the corresponding process. The buffer element is removed from the queue when the message has been received. When the message has been answered, and the answer has been received, the buffer element is linked to the pool again.

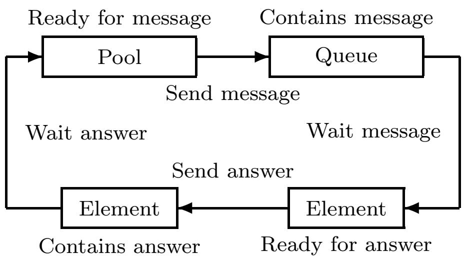

Figure 2 Message buffer states.

In a simplified form these monitor procedures can be represented as follows:

```
send message:
  take buffer from pool;
  **if** message expected **then**
    put receiver in ready queue
  **else** put buffer in message queue;

wait message:
  **if** message available **then**
    take buffer from message queue
  **else**
    **begin**
      indicate message expected;
      take another process from ready queue;
    **end**

send answer:
  **if** answer expected **then**
```

```
        **begin**
          put buffer in pool;
          put receiver in ready queue;
        **end**

    wait answer:
      **if** answer available **then**
        put buffer in pool
      **else**
        **begin**
          indicate answer expected;
          take another process from ready queue;
        **end**
```

During testing a process can ask the monitor whether a message or an answer is available for it without being forced to wait for its arrival. A process can also ask the monitor whether another process is expecting a message or an answer.

To test process communication, the system was initialized with two processes, R and S, in the ready queue (in that order). The processes exchange messages and answers in two buffer elements B and B':

```
R:  wait message($S$, $M$, $B$);
    **repeat until** answer expected($S$);
    send answer($A$, $B$);
    **repeat until** message available($R$);
    wait message($S$, $M'$, $B'$);
    send answer($A'$, $B'$);
    **repeat until** false;

S:  **repeat until** message expected($R$);
    send message($R$, $M$, $B$);
    wait answer($A$, $B$);
    send message($R$, $M'$, $B'$);
    **repeat until** answer available($B'$);
    wait answer($A'$, $B'$);
    **repeat until** false;
```

This test should produce the following output:

```
 1:     take R from ready queue
 2:     take S from ready queue
 3:     take B from buffer pool
 4:     put R in ready queue
 5:     take R from ready queue
 6:     put B in buffer pool
 7:     put S in ready queue
 8:  *  put R in ready queue
 9:     take S from ready queue
10:     take B' from buffer pool
11:     put B' in message queue(R)
12:  *  put S in ready queue
13:     take R from ready queue
14:     take B' from message queue(R)
15:  *  put R in ready queue
16:     take S from ready queue
17:     put B' in buffer pool
```

The output can be explained as follows:

Line 1: The monitor selects process R as the first process to run.

Line 2: While process R waits for nessage M, the monitor continues to run process S.

Lines 3–4: Process S sends message M to process R which in turn reenters the ready queue.

Line 5: While process S waits for answer A, the monitor continues to run process R.

Lines 6–7: Process R sends answer A to process S which in turn reenters the ready queue.

Lines 8–9: Process R continuously asks the monitor whether a message is available for it. A key interrupt preempts process R in favor of process S.

Lines 10–11: Process S sends message M' to process R.

Lines 12–13: Process S continuously asks the monitor whether an answer is available for it. A key interrupt preempts process S in favor of process R.

Line 14: Process R receives message M' and sends answer A' to process S.

Lines 15–16: Process R cycles indefinitely. A key interrupt preempts process R in favor of process S.

Line 17: Process S receives answer A'.

This test covers the eight relevant cases of process communication:

$$\left\{\begin{matrix} \textit{send} \\ \textit{receive} \end{matrix}\right\}\left\{\begin{matrix} \textit{expected} \\ \textit{unexpected} \end{matrix}\right\}\left\{\begin{matrix} \textit{message} \\ \textit{answer} \end{matrix}\right\}$$

5 Concluding Remarks

When the communication procedures worked, all possible interactions between processes and peripheral devices of various types were tested. Tests concerning dynamic process creation and termination then followed. Finally, the file system was tested.

The test output immediately revealed all serious synchronizing errors within the monitor. Quite often, the output also led to the discovery of errors in the test programs themselves. As a result of this systematic approach, the monitor was virtually error free after a test period of a few weeks.

It is worth mentioning that the monitor program was written *after* the test mechanism had been selected. If the test problem had been attacked after the monitor was finished, the relevant test events (*take* and *put*) might have been scattered all over the program as in-line code, thus making a centralization of the test mechanism impossible without extensive study and revision of the program text.

Acknowledgements

The author is indebted to Peter Naur (1963) for demonstrating the validity of simple, systematic testing techniques for large, sequential compilers.

References

Brinch Hansen, P. 1970. The nucleus of a multiprogramming system. *Communications of the ACM 13*, 4 (April), 238–250. *Article 2.*

Naur, P. 1963. The design of the Gier Algol compiler. *BIT 3*, 2–3, 124–140 and 145–166.