Using Interaction Networks for Visualisation of Message Passing

Paul Ashton
Department of Computer Science
University of Canterbury

TR-COSC 04/96, Aug 1996

Using Interaction Networks for Visualisation of Message Passing

Paul Ashton
Department of Computer Science
University of Canterbury
Christchurch, New Zealand

paul@cosc.canterbury.ac.nz

TR COSC 04/96, August 1996

Abstract

Understanding software structured as separate programs that communicate via message passing requires understanding how the various programs interact with each other, as well as understanding the individual programs in isolation. An interaction network display shows in a visual way message passing within and between programs. We have implemented an interaction network monitor for the MINIX operating system, for use in showing students how the system programs and processes of MINIX interact to provide operating system services. We give three examples to demonstrate the insights into message-based systems that can be gained from studying interaction network displays.

Introduction

Many pieces of complex software are structured as processes running different programs that communicate via message passing. Most recently-developed operating systems have this structure, particularly distributed operating systems. An advantage of structuring software in this way is that the individual programs are simpler and easier to understand than a single *monolithic* program that provides the same functionality. However, understanding such a piece of software requires considerable knowledge about the message passing patterns between component programs, in addition to knowledge about the internals of each program. Few tools exist that allow students to visualise message passing patterns, an activity that will help them understand how the various programs cooperate in solving problems.

The *interaction network* was developed as a way of representing interactive processing carried out in a distributed system. A strength of the interaction network is that it captures *all* server processing required to carry out a user request. Although interaction networks were developed originally to support performance measurement, experience with interaction network monitors developed for SunOS [2] and Amoeba [1] showed that displays of interaction networks gave considerable insights into the behaviour of message passing systems.

An interaction network monitor has been developed for MINIX, an operating system implemented as three separate programs that communicate via message passing. The monitor was developed so that students could be presented with interaction networks so as to help them understand how the various MINIX programs and processes collectively provide operating system services.

In this paper we present examples of interaction networks and use them to illustrate how MINIX carries out common operating system tasks. These example show that displays of interaction networks can contribute to increased understanding of message passing systems and their dynamic behaviour. Before presenting the examples, we give brief introductions to MINIX and to interaction networks.

MINIX

MINIX is a complete operating system developed for use in operating systems teaching [4, 5]. While the user and application-programmer interfaces are heavily based on Unix, the internal structure of MINIX is quite different to the monolithic structure of traditional Unix kernels.

In MINIX, the operating system consists of three programs and around ten processes. The `kernel` program contains code for handling traps and interrupts (which includes support for the three message passing system calls) and several processes: the device drivers, and the system task which acts as an agent for the file system and memory manager. The file system (`fs`) and memory manager (`mm`) run as separate processes in their own address spaces.

Nearly all communication between the various operating system processes, and between users processes and system processes, is through messages. There is a library function for each of the standard Unix system calls (`fork`, `exec`, `open` and so on), which sends a request message to either the memory manager (for `fork`, `exec` and so on) or the file system (for `open` and so on), then waits for a reply message.

All operating system processes are structured as servers. That is, they sit in a loop in which they receive a message, process the message then (usually) send one reply. This request/reply protocol is common in MINIX, but not universal. Sometimes there is no reply, or the reply is triggered by a later event. Sometimes there are two replies. Most of the complex message passing patterns in MINIX (including all of the examples we present) involve some message passing that does not follow the simple request/reply protocol.

To understand MINIX, it is essential to appreciate how the system processes interact to provide system services.

Interaction networks

An interaction network is an acyclic digraph that represents the *system reaction* to a single *user action* (such as a single keystroke or mouse event). System reactions are carried out by communicating processes, and consist of periods of process execution, and messages sent between processes. A system reaction is very different to the execution of a single program because the system reaction involves complete and partial execution of many programs. The fact that interaction networks contain records of all processing, including server processing, that occurs in response to a user input means that they record the message passing between operating system components that occurs during processing of interactive requests.

We have implemented an interaction network monitor for Solaris MINIX, a version of MINIX that runs hosted under Solaris 2.5[1]. A major motivation for developing the monitor was to give students the ability to observe message passing patterns that occur in MINIX. The events recorded by the monitor relate to: reception of keyboard input, sending and receiving of messages, and creation and termination of processes.

One way of analysing an interaction network is to browse a display of it. Figure 1 shows an interaction network as displayed by the browser for MINIX interaction networks. Each vertex represents an event in a process or message, and each edge represents a period of process execution (or time spent blocked), or the transmission of a message. The vertices are arranged in columns, with one column for each process that performed processing in response to the user input. Figure 1 shows three processes performing processing in response to a single keystroke.

In an interaction network display, time increases as you go down the network, with the Y-coordinate of each vertex in direct proportion to the time at which the event represented by the vertex occurred. The elapsed time between the events represented by the initial (top-most) and final (bottom-most) vertices in Figure 1 is 2.8ms. Each vertical edge represents an activity carried out by a process, and each non-vertical edge represents a message. Each light edge represents a period during which a process was blocked waiting for a message. Each dark edge represents a period during which a process was runnable (though not necessarily running), or a message being sent. The length of each edge in the Y direction shows the duration of the activity that it represents. The interaction network in Figure 1 is explained in the next section.

The MINIX interaction network browser supports panning and zooming, can provide extra details on a selected event vertex (such as the type of message being sent or received), and provides some ability to tailor the way in which the network is displayed.

[1]For details on downloading the current Solaris MINIX release, see `http://www.cosc.canterbury.ac.nz/~paul/smx.html`. The interaction network monitor is part of the Solaris Minix release.

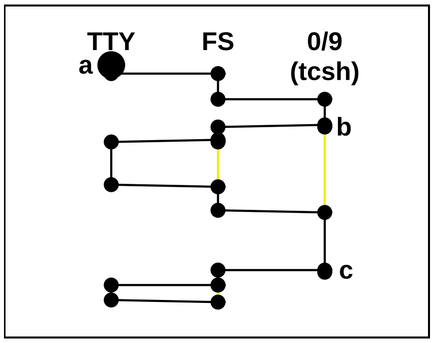

Figure 1: Interaction network showing reaction to input of a single character that is part of a shell command line.

The interaction network displays presented in the next three sections show how displays of interaction networks provide considerable insights into the internal operation of MINIX. The information needed to write the commentaries on each interaction network was gained by looking at interaction network displays, determining message types using the browser and, where necessary, reading relevant source code fragments.

Example 1: Input of a Single Character

The interaction network in Figure 1 shows the processing that occurs as the result of input of a single character that is part of a shell command line. The shell in use, `tcsh`, reads command lines character-by-character to allow for command line editing. The character whose input triggered the interaction network did not terminate the command line, so all that occurred was that `tcsh` echoed the character and added it to a buffer, before asking for the next character.

Vertex `a` in Figure 1 represents the moment at which the terminal task (one of the processes that run in the `kernel`) was notified that a character had been typed. The terminal task sends a `REVIVE` message to the file system, causing it to reply to an earlier `READ` message from `tcsh`. `tcsh` sends a `WRITE` message (`b`) to the file system so as to echo the character just entered. The file system sends a `DEV_WRITE` message to the terminal task, which then replies that the character has been echoed successfully. The file system then relays this fact to `tcsh` in a reply message.

`tcsh` has not got a complete command line, so it sends a `READ` message (`c`), to the file system which sends a `DEV_READ` message to the terminal task. There is no terminal input, so the terminal task replies to that effect.

As noted earlier, message exchanges involving one request message being sent to a server, and the server sending back one reply are common in MINIX. Three such request/reply message pairs are evident in Figure 1.

Notice that when the terminal task discovers it has no characters to return to the file system, it replies anyway. This allows the file system to continue processing requests. If the terminal task did not reply until it had input, then the file system could be suspended for a long period, during which all active processes would soon find themselves waiting to send a message to the file system.

The file system does not in turn reply to tcsh, which keeps tcsh suspended until there is further input. This is the standard MINIX method for suspending a user process—a reply is withheld until the request can be completed. In the meantime, the operating system process is free to handle requests from other processes.

When subsequent input does occur, the terminal task sends a REVIVE message to the file system (which is the second reply to the file system's DEV_READ request) that allows the file system to (finally) reply to the earlier READ.

The message received by tcsh just before vertex b is a reply to the message sent at vertex c in the previous interaction network recorded for this user. It should be clear how multiple instances of this interaction network would occur one after the other as the characters of a command line were entered, with the message passing towards the end of one interaction network setting the scene for the message passing that occurs at the start of the next.

Finally, in MINIX messages are not buffered by the operating system. A message is copied from sender to receiver only when both are ready to communicate. One consequence of this is that the message passing activities shown in Figure 1 are very short (9 to 10 microseconds), as all that is involved is copying the message.

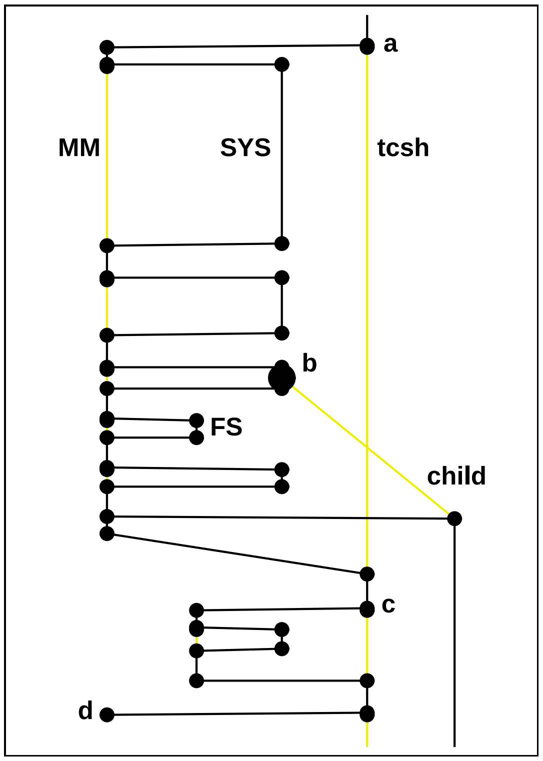

Figure 2: Interaction network showing the message passing that occurs during process creation.

Example 2: Process Creation and Termination

The message passing that arises from a request to create a new process is shown in the interaction network fragment in Figure 2.

At vertex a, tcsh sends a FORK message to the memory manager, asking it to create a new (child) process. The memory manager allocates two new memory areas for the child, to hold its data and stack segments (the text segment is shared), then sends two SYS_COPY messages to the system task in the kernel; the first to copy the data segment, and the second to copy the stack segment. The memory manager is an ordinary user process, with no ability to access memory outside its own address space, which is why it must ask the system task to do the copying on its behalf.

The memory manager then sends FORK messages to the system task and the file system to inform them of the creation of the new process. In MINIX, the process table is split between the kernel, the memory manager and the file system. The FORK messages are sent so that the system task and file system add an entry for the new process to their respective pieces of the process table. The child process is shown as being created at vertex b, with the diagonal edge leaving vertex b representing time during which the child process is blocked.

The memory manager determines the memory map for the child process (which records the sizes and virtual and physical addresses of the text, data and stack segments), and sends a SYS_NEWMAP message to the system task so that the memory map is recorded in the kernel's process table.

Process creation is now complete. The semantics of the UNIX fork operation mean that the child is created in the same state as the parent. In MINIX, this means that the child is created waiting for a reply from the memory manager. The memory manager therefore sends two reply messages: one to the child with a return value of 0, and one to the parent with a return value of the process id of the child process. Thirteen messages have been sent to carry out the FORK request.

When tcsh resumes executing, it sends a TIMES message (c) to the file system to determine CPU time used to date. The file system simply forwards the request to the system task in the kernel, which is where the accounting records are kept. User processes can send messages to the file system and memory manager only, which is why tcsh cannot send the TIMES message directly to the system task.

Finally in Figure 2, tcsh sends a WAIT message (which is received at d) to the memory manager to wait for one of its child processes to terminate. The memory manager does not reply immediately, thereby suspending tcsh. Again, a user process is blocked because a reply is withheld.

Figure 3 shows a second fragment of the interaction network from which Figure 2 is taken. In it we see the message passing that occurs when the process whose creation is shown in Figure 2 terminates. At vertex a, date (which is the program now being executed by the child process) sends an EXIT message to the memory manager. The file system and system task are informed that date is terminating as, once again, the file system and kernel process

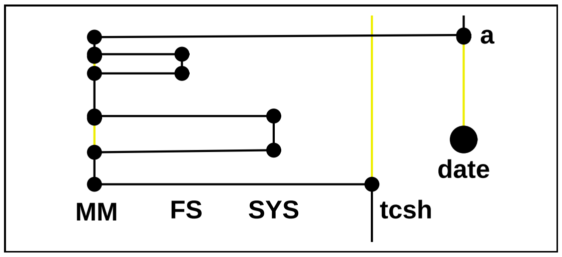

Figure 3: Interaction network showing the message passing that occurs during process termination.

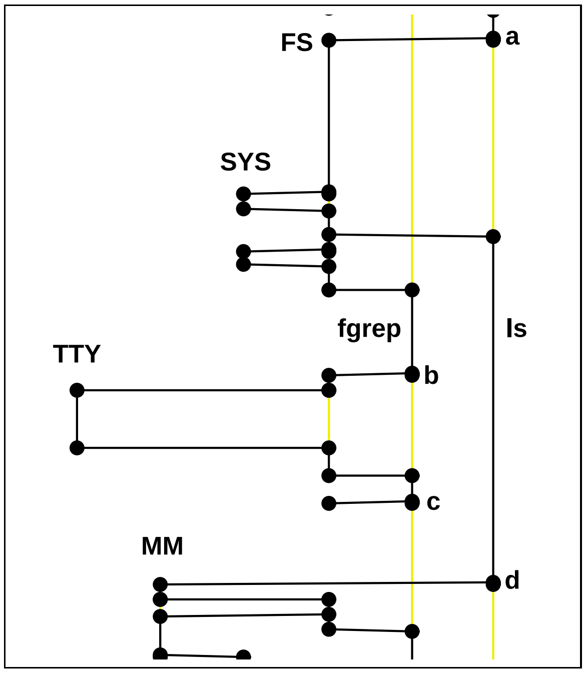

Figure 4: Interaction network showing communication between two user processes through a pipe.

tables must be updated. We regard the final termination as being done by the system task. The memory manager then sends a reply message; not to `date` (it not longer exists) but to `tcsh`. This reply is sent in response to the `WAIT` message shown at the bottom of Figure 2, which was sent 209ms earlier.

Example 3: Communication Through a Pipe

The final example shows the message passing that occurs when two user processes communicate through a traditional Unix pipe. In the example, the output of `ls` (list directory) is being sent to `fgrep` (extract lines matching a specified string) through a pipe. The output of `fgrep` is going to the terminal.

At vertex `a`, `ls` sends a `WRITE` message to the file system to request that data be written to the pipe. Because the file system is a user process, it must ask the system task to copy the data into one of the file system's buffers, after which the file system replies to `ls`. The file system then finds that there is a process waiting to read from the (no longer empty) pipe, so uses the system task to copy data to `fgrep`, then sends a reply to an earlier `READ` message from `fgrep`. This is another example of one request message resulting in two replies.

The data read from the pipe is processed by `fgrep`, and at vertex `b` a `WRITE` message is sent to the file system to write some output to the terminal. The file system forwards this request message to the terminal task. Notice that because the terminal is running within the `kernel` it can copy output from `fgrep` without needing to invoke the system task.

With the terminal write complete, `fgrep` sends a `READ` message at vertex `c` to ask the file system for more data from the pipe. The pipe is currently empty, so the file system suspends `fgrep` by withholding the reply message. Notice that `fgrep` would have been suspended in the same way prior to the sequence of events shown in Figure 4. The pattern of messages from the sending of the `WRITE` message at vertex `a`, to the reception of the `READ` message sent at vertex `c` will have occurred repeatedly as data is written to the pipe by `ls` and read and processed by `fgrep`. A total of eight messages is needed to carry out the transfer of a single buffer through the pipe.

When `ls` resumes execution, it sends an `EXIT` message to the memory manager at vertex `d`. The memory manager then notifies the file system that the `ls` process is terminating (as described in example 2). Notice that after the file system replies to the memory manager, it also sends a message to `fgrep`. This is because the termination of `ls` has removed the last process that can write to the pipe, so an end of file indication must be returned to the waiting `fgrep` process.

Experiences

To date, we have had some experience with teaching students about MINIX using interaction networks. In 1995, students of COSC805 (a post-graduate operating systems course) were asked to prepare a report that used interaction network displays to illustrate aspects of the internal operation of MINIX. Students were given the monitoring software, and selected appropriate networks themselves. They had no difficulty in coming up with many interesting examples, including ones similar to those presented in this paper.

In 1996, a report based on the 1995 class work was given to the COSC805 class, and figures from the report were discussed in lectures. Informal feedback indicated students thought that message passing patterns were well captured by the interaction network displays. In reading the report, they commented that they were constantly switching attention between the figure and the supporting text, which they found distracting.

Related work

Our approach is not unique in its use of digraph displays to show recorded message passing activity. Similar displays can be found for both parallel and distributed programs (see, for example, [6, 7]). These other displays are, however, based on capturing message passing within a *single* parallel or distributed program; the monitors have no ability to follow execution outside the processes that execute the program.

Because an interaction network records all processing that results from a user input, interaction network monitors are capable of following execution into processes executing other programs. This makes interaction network monitors very well suited to visualisation of message passing in systems, such as message-based operating systems, that are made up of communicating processes executing more than one program. We are not aware of other monitors that can do this.

Tanenbaum seems to regard graphical displays of message passing patterns as a useful teaching device, as he uses them to explain MINIX message passing patterns [4]. In his figures, each process is represented by a single vertex, with each message shown as an edge. The edges are numbered to show the order in which the messages are sent. The figures presented in [4] rely on the availability of a MINIX expert to produce them, whereas an interaction network monitor lets students explore any aspect of the system that catches their interest. Also, interaction network displays can capture much more complex message passing patterns.

Other visualisation methods for operating systems provide detailed information on a single algorithm, and so have very different objectives from our own. A typical example is Hartley's use of an algorithm animation system to illustrate operation of a solution to the dining philosophers problem [3].

Conclusions

Many complex pieces of software are structured as separate programs that communicate by message passing. Students seeking to understand such software must learn about the (often complex) interactions between the various programs, as well as learning (using traditional tools and techniques) about the internal operation of each program. Monitors that can record and display interaction networks provide valuable information on interactions between programs because (unlike program monitors) they are able to track requests for processing from one program to another.

The examples presented show that interaction networks are a rich source of information on the behaviour of message-based systems. The figures in this paper show examples of several system processes, running in three disjoint address spaces, cooperating to carry out requests from user processes. The figures also show several examples of unusual message passing patterns, such as a reply being withheld to suspend a user process, two replies being sent in response to a single request, and a reply being sent to a process different from the requesting process.

Displays of MINIX interaction networks show clearly interactions between processes executing communicating programs, and provide insight into the overall operation of such pieces of software. Interaction network monitors are not MINIX-specific, and can be applied to a wide variety of message based systems, as shown by previous implementations for SunOS [2] and Amoeba [1].

Acknowledgements

Thanks to Tim Bell and Michel de Champlain for their comments on drafts of this paper, and to the COSC805 class of 1995 for providing initial versions of many of the examples.

References

[1] ASHTON, P. An interaction network monitor for Amoeba. Tech. Rep. TR-COSC10/95, University of Canterbury, Department of Computer Science, Oct. 1995.

[2] ASHTON, P., AND PENNY, J. A tool for visualising the execution of interactions on a loosely-coupled distributed system. *Software—Practice and Experience 25*, 10 (Oct. 1995), 1117–1140.

[3] HARTLEY, S. Animating operating systems algorithms with XTANGO. In *Proceedings of the 25th SIGCSE Technical Symposium* (New York, Mar. 1994), pp. 344–348.

[4] TANENBAUM, A. S. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[5] TANENBAUM, A. S. A UNIX clone with source code for operating systems courses. *Operating Systems Review 21*, 1 (Jan. 1987), 20–29.

[6] TSAI, J. J. P., FANG, K.-Y., CHEN, H.-Y., AND BI, Y.-D. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering 16*, 8 (August 1990), 897–915.

[7] ZERNIK, D., SNIR, M., AND MALKI, D. Using visualization tools to understand concurrency. *IEEE Software 9*, 3 (May 1992), 87–92.