Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 224 (2009) 89–95
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Using Graphviz as a Low-cost Option to Facilitate the Understanding of Unix Process System Calls

Miguel Riesco[1] Marián D. Fondón[2] Darío Álvarez[3]

Dept. of Computing
University of Oviedo
Oviedo, Spain

Abstract

Unix system calls to create and execute processes are usually hard to understand for novice students. Using graphics for visualizing the behaviour of these system calls can be useful both for the teacher to explain and for the students to understand them. The problem here is that there is no software specifically addressed to generate graphical representations of these kind of programs, and to develop it would be costly. Instead of developing a complete system to visualize programs that use Unix system calls, we turned to a "cheaper" alternative solution. In this paper we show how we have used the open source graphviz tool to develop a simple way of generating graphical representations of the behaviour of these system calls, thus facilitating the comprehension of this important part in the learning of the Unix operating system.

Keywords: Unix Operating System, process system calls, program visualization.

1 Introduction

When teaching Operating Systems, it is usual to employ graphics to better illustrate the different aspects involved. Thus, graphs representing the modules of the operating system, the lifecycle of processes, or the message-based process communication are common.

One of the topics appearing frequently in this subject is the Unix operating system, from diverse points of view: internal structure, command-line user, or system programmer. In the systems programmer view, the API calls of the system are studied, and students develop programs using these calls. Here it is also usual to resort to graphics supporting the explanation of how the system calls work as the

[1] Email: albizu@uniovi.es
[2] Email: fondon@uniovi.es
[3] Email: darioa@uniovi.es

doi:10.1016/j.entcs.2008.12.052

program using them is running, to show the dynamic behaviour of a program using system calls, and to visualize the evolution of the data structures involved.

These graphics are generally created manually by the teacher, or taken from text books (where, in turn, they were created by the author manually). Although these static graphs are useful, some kind of animation where the evolution of processes could be seen would be better.

As far as learning the Unix system calls is concerned, we teachers have been explaining how processes are created by using drawings in the blackboard, providing interactive animation by drawing and erasing as we explain. It is important to have some kind of graphic to help students comprehending this part of the API, as this is not something students assume as natural: to the peculiar behaviour of these calls we have to add the fact that there are a number of processes running concurrently, thus making the comprehension more difficult.

This was detected before [5], but the solution presented is not readily accessible, needs specific systems to generate and visualize animations of processes. A different aproach was done by [4]. However, it is aimed at studying the interaction between *fork* and *dup* system calls. Apart from these, we have not found more systems deemed adequate for our needs. In this paper we present the work we are developing to automatically generate, from a real program that uses process-related system calls, graphics illustrating its behaviour, supporting the teachers explanation of the topic.

2 POSIX process services

The POSIX API offers a reduce set of process-related system calls. There are other ways to get information about processes, but most of the functionality lies in four system calls:

- *fork*: clones the process making the call (the clone is a child process).
- *exec*: changes the program executing the process (does not create a new process, it just substitutes the code the process is running for another one).
- *wait*: waits for the termination of a child process.
- *exit*: Terminates the execution of a process.

Students have problems grasping the behaviour of these system calls, as the natural way would be to have a call creating a new process running the program passed as an argument. To clarify it, graphics such as those in [1] or [3], similar to what is show in figure 1, try to represent the behaviour of the program.

```
main() {
 pid_t pid;
 int i, n=4;
   for (i=0; i<n ; i++ ) {
     pid=fork();
     if (p==0) break;
  }
}
```

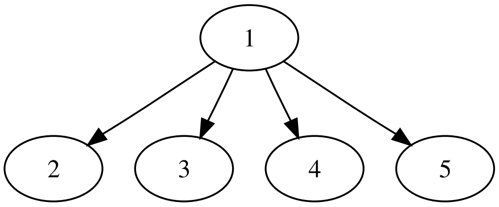

Figure 1 Graphic representing the creation of 4 processes

Even though for simple examples it is easy to create illustrative graphics, for complex programs it is not the case. Besides, trying to correctly represent the dynamic behaviour of processes can take lots of time.

Trying to solve this problem, and having experience with the graphviz tool [2] for other reasons, we applied it to the generation of graphics representing process-related system calls.

3 Instrumenting programs to generate graphviz graphics

We have developed a function library with names analogous to the original Posix calls (*myfork*, *myexec*, *mywait*, *myexit*). These functions maintain the functionality of the original system call (*myfork* does *fork*, *myexec* does *exec*, etc.), but they also generate one or more lines in graphviz *dot* format to graphically represent the system call.

Figure 2 shows the implementation of *myfork* function which, besides calling the original *fork* function, generates a line of graphviz code representing a pair of nodes (containing the PIDs of both parent and child processes) linked by an arrow representing their relationship.

Similar functions have been developed for the rest of the services. Every function includes a call to the original service and the necessary sentences to generate the graphviz code to represent its behaviour.

```
int myfork() {
 pidf=fork();
 if (pidf==0) {
  ppid=getppid();
  pid=getpid();
  sprintf(cad,"%d␣->␣%d;\n",ppid, pid);
  store(cad);
 }
 return (pidf);
}
```

Figure 2 Implementation of *myfork* function

Figure 3 shows the graphical representation of the four system calls. Each call gets this representation:

- *fork* is represented with a node for each process containing the process PID, with an edge indicating the father-child relationship.
- *exec* is represented in a similar way, adding the name of the execed program after the process PID.
- *wait* adds a dotted line from the child process to its father (showing the SIGHLD signal sent when the child process ends).
- *exit* uses a dotted node to represent the terminated process.

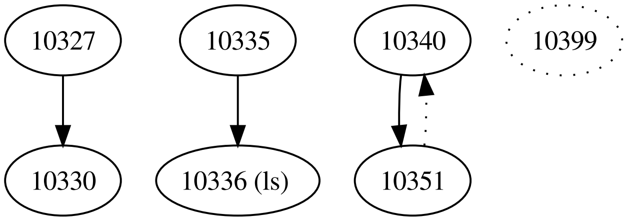

Figure 3 Graphical representation of the fork, exec, wait and exit system calls

To generate the graphical representation of the dynamic behaviour of a real program, the original system calls are replaced with the instrumented library functions mentioned before. A simple program or script does this. The original functionality is preserved and we get its graphical representation.

```
main() {
 int i,p;

   for (i=0; i<3; i++)
     p=fork();
}
```

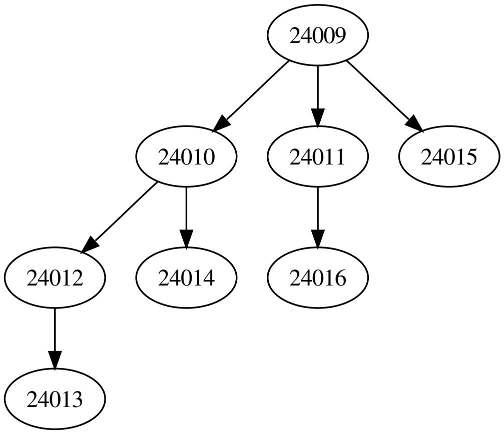

Figure 4 Creating processes recursively and their representation

Once the process ends execution, a graphviz *dot* file is created with the representation of all the executed system calls. This is fed into the graphviz tool, which in turn will generate the corresponding graphic file. Figure 4 shows the graphical representation of a typical program that creates processes recursively.

This kind of graphic is useful in some situations, but we realized that it would be more interesting to analyze how the program evolves as each call is done, and not only at the end of the execution. We developed a second version that allows to visualize this dynamic behaviour. Using the same file generated after the execution of the program, we do the following process:

(i) The global *dot* file is divided into as many files as lines are in it. The first one has the first line of the original file, the second one the first two lines, and so on. That is, the ith file has the behaviour of the first i calls (each call has a line in the global *dot* file).

(ii) n graphics files are created using the n files of the previous step using graphviz.

(iii) These graphics can be uploaded and then visualized using a web application. With simple controls, the user is able to see the graphical representation of the execution sequence, while showing the source code at the same time. With this support, the teacher can develop an detailed explanation of how the creation and destruction of processes is evolving. This is of great help for the student to grasp the topic.

Figure 5 shows a program that uses the four POSIX services we want to study. The nine graphics that have been generated to analize the dynamic evolution of the programa are shown too. Each picture represents the state of the processes after the execution of one system call. Using this sequence of pictures a student can better understand how each system call works.

```
void hdler (int a){
 int cr, pid;
 pid=wait (&cr);
 signal(SIGCHLD, hdler);
}

main() {

 signal(SIGCHLD, hdler);
 pid=fork();
 if (!pid)
   exec("ps");
 pid=fork();
 if (pid)
  do
   pidr=wait(&cr);
  while (pidr!=pid);
 else {
   exec("ls");
   exit(-1);
 }
 exit(0);
}
```

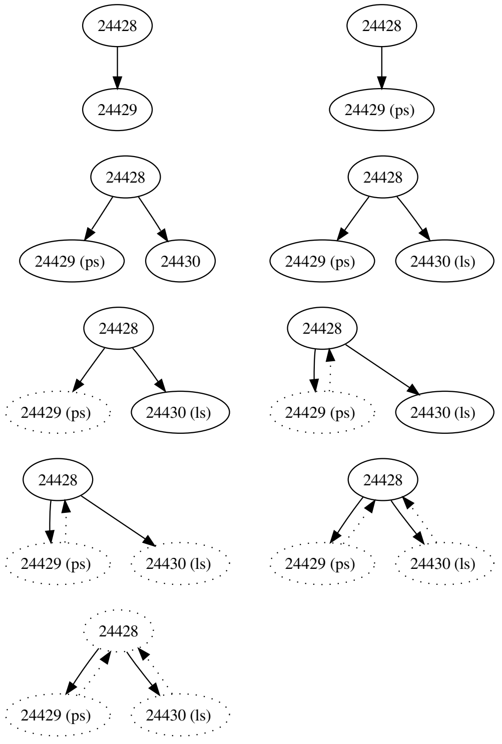

Figure 5 Showing the execution sequence of a program

4 Observed results

To date, we have used this method and tool mainly to support the teachers explanation of the topic, although it could also be used by students to analyze the execution of their own programs.

We do not have yet an exhaustive analysis of the impact in learning the topic. However, we can state two facts:

(i) The teachers that have used the method are very satisfied, as they believe it facilitates the construction of examples, as well as the students comprenhension of the topics.

(ii) The students have accessed the web pages were the teacher-created examples are stored. In the first week we had 874 page loads. We think this is quite a success, as there are 72 students and 8 different examples.

Therefore, we think this experience is positive. We have the intention of going more deeply and to apply these ideas to other topics in the subject.

5 Future Work

We plan to develop future work along these lines:

The priority is to improve the process of creating and publishing the graphics. Currently each step (creating the original file, dividing it into iterative files, generating the graphics, publishing graphics into web pages) is done independently and semi-manually. So, the first thing to do will be the packaging of the independent steps into one program that automatically generates and publishes graphics in one step from the data of the execution of a process.

Another issue is to develop a tool to visualize the evolution of the program in real time, allowing the user to interact with the running program stepping back and forth (a kind of simple graphical debugger).

Apart from the technical aspect, we have the intention to apply similar ideas to other parts of the subject that could benefit from this kind of support for the explanations. File management is a good candidate. In this case we would visualize the evolution of the data structures involved in performing each system service. We are also studying how to apply this to other topics such as concurrent programming or input/output management, although it is not that obvious.

6 Conclusions

In this paper we have shown a method to graphically represent the behaviour of the POSIX system calls for process management.

Using a free tool such as Graphviz it is even possible to represent simulations of the dynamic behaviour of the processes using these system calls. So, it is not always necessary to spend a large amount of time developing complex program visualization software, but simpler solutions can be used instead.

The graphical representation of the behaviour help the teachers develop examples for a better explanation of the topic, while the students can analyze the behaviour of the programs in a more convenient way.

The experience has been a positive one. We are still working on the improvement of the graphics-creation process, and to apply the same ideas to other topics in the Operating Systems subject.

References

[1] Carretero, J., P. de Miguel, F. García and F. Pérez, "Sistemas Operativos, 2/e," McGraw-Hill Interamericana, Inc., Madrid, Spain, 2007.

[2] Ellson, J., E. R. Gansner, E. Koutsofios et al., *Graphviz and dynagraph static and dynamic graph drawing tools*, Technical report, AT&T Labs - Research, Florham Park NJ 07932, USA (2004), also available as http://www.graphviz.org/Documentation/EGKNW03.pdf.

[3] Robbins, K. and S. Robbins, "UNIX Systems Programming: Communication, Concurrency and Threads (2nd Edition)," Pearson Education, Inc., Upper Saddle River, New Jersey, USA, 2003.

[4] Robbins, S., *Exploration of process interaction in operating systems: a pipe-fork simulator*, SIGCSE Bull. **34** (2002), pp. 351–355.

[5] Vogt, C., *Visualizing unix synchronization operations*, SIGOPS Oper. Syst. Rev. **31** (1997), pp. 52–64.