CS:APP2e Web Aside ECF:GRAPHS: Process Graphs*

Randal E. Bryant
David R. O'Hallaron

July 14, 2014

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Third Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2016. In this document, all references beginning with "CS:APP3e " are to this book. More information about the book is available at* `csapp.cs.cmu.edu`.

1 Process Graphs

The process graph is a handy tool for understanding the behavior of programs that use the `fork` and `wait` functions. Section CS:APP3e-8.4.2 mentions this idea in passing. In this note, we give a more thorough treatment and show some examples from the textbook.

A *process graph* is a simple kind of precedence graph that captures the partial ordering of program statements. Each vertex, a, corresponds to the execution of a statement in a C program. A directed edge $a \rightarrow b$ denotes that statement a "happens before" statement b. Edges can be labeled with information such as the current value of a variable or the output of a preceding `printf` statement. Each graph begins with a vertex that corresponds to the parent process calling `main`. This vertex has no inedges and exactly one outedge. The sequence of vertices for each process ends with a vertex corresponding to a call to `exit`. This vertex has one indedge and no outedges.

For example, Figure 1 shows the process graph for the example program in Figure CS:APP3e-8.15. Initially, the parent sets variable `x` to 1. The parent calls `fork`, which creates a child process that runs concurrently with the parent in its own private address space.

```
1  int main()
2  {
3      pid_t pid;
4      int x = 1;
5
6      pid = Fork();
7      if (pid == 0) {  /* Child */
8          printf("child : x=%d\n", ++x);
9          exit(0);
10     }
11
12     /* Parent */
13     printf("parent: x=%d\n", --x);
14     exit(0);
15 }
```

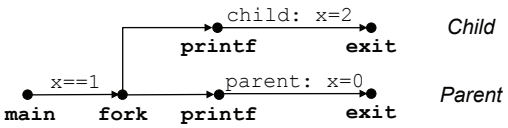

Figure 1: **Process graph for Figure CS:APP3e-8.15.**

For a program running on a single processor, any *topological sort* of the vertices in the corresponding process graph represents a feasible total ordering of the statements in the program. Here's a simple way to understand the idea of a topological sort: Given some permutation of the vertices in the process graph, draw the sequence of vertices from left to right, and then draw each of the directed edges. The permutation is a topological sort if and only if each edge in the drawing goes from left to right. Thus, in our example program in Figure 1, the `printf` statements in the parent and child can occur in either order because each of the orderings corresponds to some topological sort of the graph vertices.

The process graph can be especially helpful in understanding programs with nested `fork` calls. For example, Figure 2 shows a program with two calls to `fork` in the source code. The corresponding process graph helps us see that this program runs four processes, each of which makes a call to `printf`, and which can execute in any order.

```
1  int main()
2  {
3      Fork();
4      Fork();
5      printf("hello\n");
6      exit(0);
7  }
```

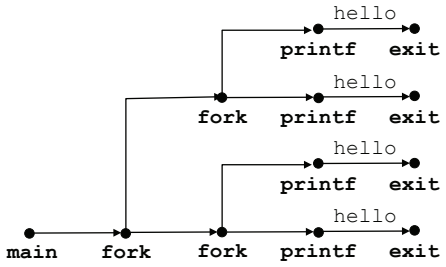

Figure 2: **Process graph for Figure CS:APP3e-8.16(c).**

As another example, Figure 3 shows the process graph for the program in Practice Problem CS:APP3e-8.2, where the child process executes two `printf` statements, while the parent executes only one.

The process graph can also help you to understand programs that use `wait` to synchronize with child processes. For example, Figure 4 shows the process graph for the program in Practice Problem CS:APP3e-8.3. The sequences $acbc$, $abcc$, and $bacc$ are possible because they correspond to topological sorts of the

```
1 int main()
2 {
3     int x = 1;
4
5     if (Fork() == 0)
6         printf("printf1: x=%d\n", ++x);
7     printf("printf2: x=%d\n", --x);
8     exit(0);
9 }
```

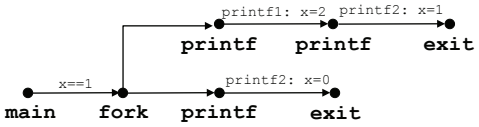

Figure 3: **Process graph for Practice Problem CS:APP3e-8.2**

process graph. However, sequences such as $bcac$ and $cbca$ do not correspond to any topological sort and thus are not feasible.

```
1 int main()
2 {
3     if (Fork() == 0) {
4         printf("a");
5     }
6     else {
7         printf("b");
8         waitpid(-1, NULL, 0);
9     }
10    printf("c");
11    exit(0);
12 }
```

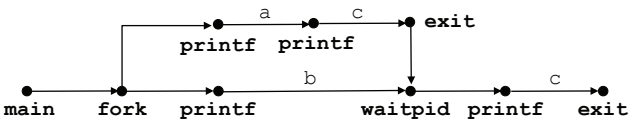

Figure 4: **Process graph for Practice Problem CS:APP3e-8.3**

Figure 5 shows another example, from Practice Problem CS:APP3e-8.4. For part A, we can determine the number of lines of output by simply counting the number of `printf` vertices in the process graph. In this case, there are six such vertices, and thus the program will print six lines of output. For part B, any output sequence corresponding to a topological sort of the graph is possible. For example: “Hello”, “1”, “0”, “Bye”, “2”, “Bye”.

```
1  int main()
2  {
3      int status;
4      pid_t pid;
5
6      printf("Hello\n");
7      pid = Fork();
8      printf("%d\n", !pid);
9      if (pid != 0) {
10         if (waitpid(-1, &status, 0) > 0) {
11             if (WIFEXITED(status) != 0)
12                 printf("%d\n", WEXITSTATUS(status));
13         }
14     }
15     printf("Bye\n");
16     exit(2);
17 }
```

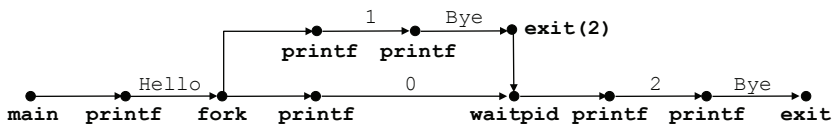

Figure 5: **Process graph for Practice Problem CS:APP3e-8.4**