REMARKS ON REPEATED FORKING AND VISUALIZATION

EVAN RUSSENBERGER-ROSICA

1. Understanding Repeated Forking

Consider the following:

Algorithm 1 Fork a program 4 times

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
void main()
{
    for (int i=0;i<4;i++)
    {
        pid_t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0)
        {
            /* error occurred */
            fprintf(stderr, "Fork Failed");
        }
        else if (pid == 0)
        {
            /* child process */
            printf("%s%d%s%d\n", "child: ", getpid(), " of ", getppid());
        }
        else
        {
            /* parent process */
            /* wait for the child to complete */
            wait(NULL);
        }
    }
}
```

What is the output of this program? From [1, 117], we have that:

When a process creates a new process, two possibilities for execution exist:

(1) The parent continues to execute concurrently with its children.
(2) The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

(1) The child process is a duplicate of the parent process (it has the same program and data as the parent).
(2) The child process has a new program loaded into it.

Further, we have that "A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process."[1, 117]. It is only after we use the `exec` system call that we replace the process's memory space with a new program. Since we have not invoked `exec` in Algorithim 1, each forked process is a duplicate of the parent process. This duplication includes all of the parent's variables, including the loop counter `i`. For example, if the child was created in the 0th iteration of the parent's loop, the child's loop will initialize at `i=1`. Thus, as the parent's loop counter gets closer to its termination condition `i<4`, child processes are created with initialization conditions approaching the termination condition. For instance, the child forked when the parent's loop is at`i=3`, will have its loop counter set to `i=4` the first time it runs, thus it will not enter the for loop, and will not have any children itself. Thus we see that as the parent's loop counter approaches its termination state, the children generated will fork fewer times, and therefore will have fewer (and eventually 0) children of their own.

This program will output something similar to the following. We have underlined all the children of the main process `7630`.

```
child: 7631 of 7630
_____________________

child: 7632 of 7631
child: 7633 of 7632
child: 7634 of 7633
child: 7635 of 7632
child: 7636 of 7631
child: 7637 of 7636
child: 7638 of 7631
child: 7639 of 7630
_____________________

child: 7640 of 7639
child: 7641 of 7640
child: 7642 of 7639
child: 7643 of 7630
_____________________

child: 7644 of 7643
child: 7645 of 7630
_____________________
```

Observe that the root process `7630` has 4 children; `7631,7639,7643,7645`. One of its children, `7645` has no children. Another child, `7643` has only one child: `7644`. Child `7639` has 2 children, and

one grandchild. Lastly, child `7631` has 3 children, 3 grandchildren, and 1 great grandchild. Making sense of this output is hard, so in the next section we will discuss how we can visualize this output.

2. Visualizing Repeated Forking

In this section, we discuss how to visualize the output of Algorithim 1. We only use the last two digits of the process id for aesthetic reasons. The book visualizes the forking of 4 processes similar to figure 2.1. We read this tree as follows:

(1) The number of child processes a given process has is equal to the number of child nodes the corresponding node of the tree has
(2) The total number of processes equals the total number of nodes in the tree
(3) The depth of a given node gives the distance of a process from the root process

However, this method of visualization has a few shortcomings

(1) Forking inputs one process and outputs two (the original one, plus a duplicate), thus we would expect the tree to have a branching factor of two, however it does not.
    (a) Because of this, the relationship between the number of `fork` operations and the number of generated processes is unclear.
(2) Nodes of the same depth are created at different times (e.g: `7645` is created when `i=3` in the root program, but it has the same depth as `7631`, which is created when `i=0`). This makes it unclear which nodes were created when, unless we manually label each edge.
(3) It is not clear why different nodes have a different amount of children.

To address these points, we suggest an alternative tree visualization, which more closely parallels the actual operation of the program. This is shown in figure 2.2. Observe:

(1) This tree has a constant branching factor of two. This makes it clear that the number of processes generated by forking n times is 2^n.
(2) Forks performed on the ith iteration have the same depth in the tree. This shows us when, and in what order processes were forked. This makes it clear why, e.g. `45` has no children - it is because when it was forked from `30`, the loop counter `i` was already equal to `3`. Therefore, when `45` ran, the loop counter was initialized to `4`. Thus it failed to enter the for loop, and never encountered the `fork` statement. Therefore it has no children.

While these are major advantages, there are costs:

(1) This tree has more nodes and is thus less compact than the book's representation
(2) It is slightly harder to see how many children a given process has. The number of children of process `x` is equal to the number of (non-`x`) nodes which are adjacent to an `x` node. E.g, there are 4 non `30` nodes which are adjacent to `30` nodes. Thus `30` has 4 children.

In conclusion, while students can use either visualization method (or neither), we believe that using both provides the most complete picture of what occurs when we repeatedly fork a process.

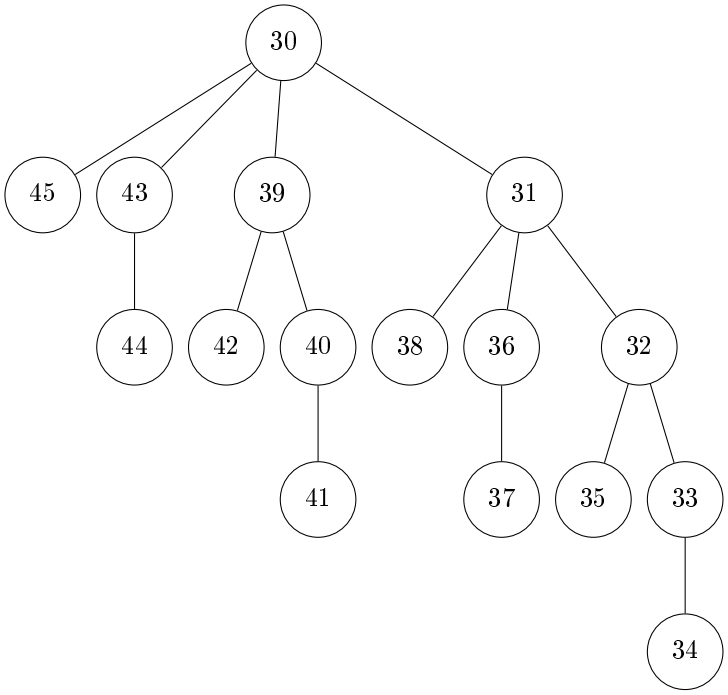

FIGURE 2.1. Forking 4 Processes in a Loop

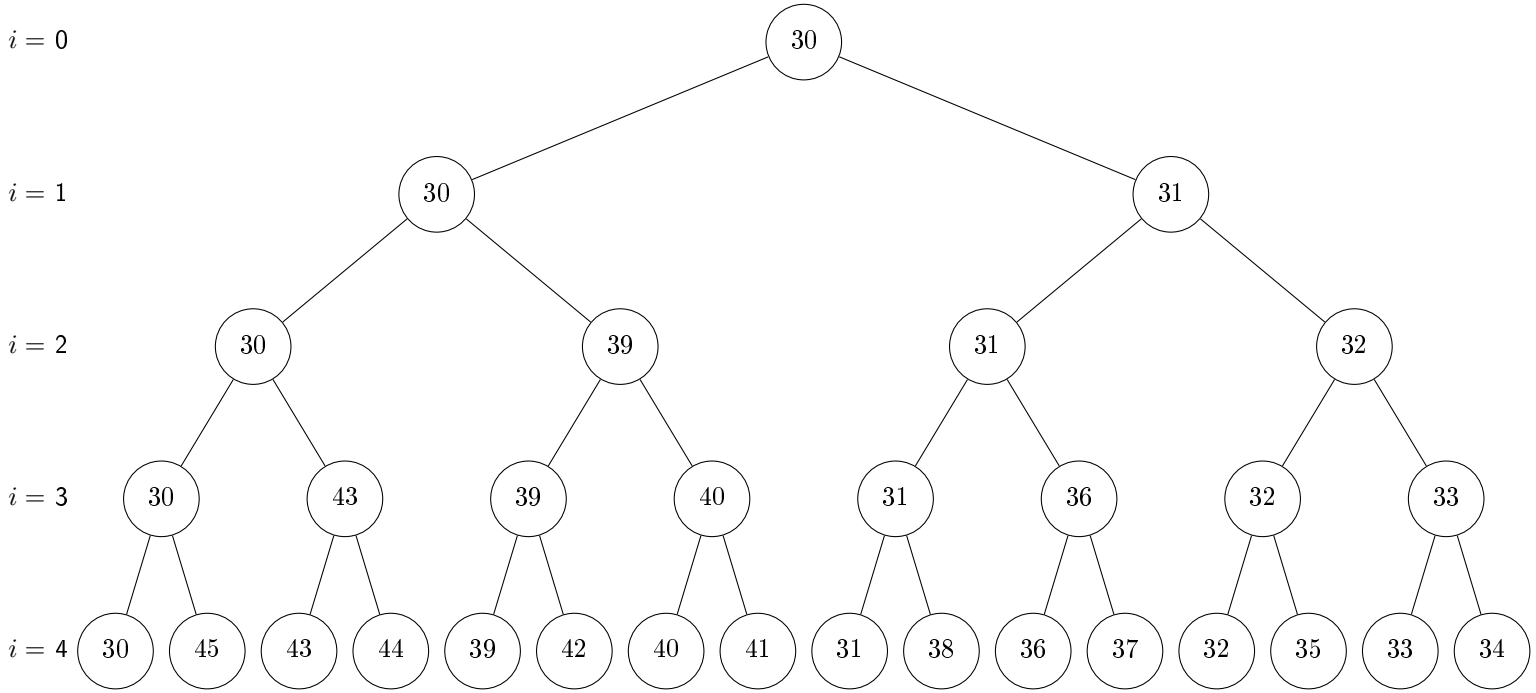

FIGURE 2.2. Forking 4 Processes in a Loop

References

[1] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John WIiley & Sons INC, 2012.