Birthday+Pid Name Analysis

Joe Meehean Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706
{jmeehean@cs.wisc.edu}

October 14, 2005

Process management is a problem often dealt with within the kernel. However, a sufficiently complex user-space system must also manage the processes that make up the system. In a highly structured system with with a single process at the root of the process hierarchy, strict system defined forking semantics along with help from the OS can allow relatively easy process management. However, in a more loosely defined system that allows arbitrary forking, and in which the parent of a process may not be the process responsible for managing the child, the OS does not provided sufficient support for process management. Namely, the OS will only notify a process its childrens deaths and process ids are not unique identifiers of a given process. Therefore, a user-level mechanism is required to determine the status of a given process. A simple layer of indirection could be applied to the forking functionality that allows a process management service to intervene between the user application and the system, essentially virtualizing processes and process ids. Our work is focused on ensuring that a particular service, the Condor master, has exited so that we can ensure that no Condor services are currently running on a given host. Therefore, adding another layer of services would be unacceptable because we would need be able to guarantee that this layer of services has also exited, and so on. We believe that the correct model for this is a simple program that queries and returns the status of the master process. We can ensure that the querying program has exited by returning the status of the master through the program's exit code.

To correctly query the status of a given process, we need a unique name for a process. Unfortunately, the OS-provided pids are not sufficient because they may be reused after a process exits. An obvious, and often used, solution to this problem is to pair process birthday with the pid to create a unique identifier for a process. Unfortunately, this technique is error prone due the precision of the birthday and variability of system time to due network time protocol (NTP). To better understand and to help alleviate the problems with a pid and birthday (bpid) identifier, we created a state machine that represents the pid usage process, and created a model of pid usage based on this state machine. We defined the precision of the birthdays returned by the OS as the 'range' of possible more precise birthdays for a process or r(P). Then we defined the bday function which returns true if birthday returned by the OS for the given pid is within the range of a that process, r(P), and false otherwise.

Next, we subdivided the time of a query into five possible states a pid could be in (from our state machine): unborn, alive, zombie, dead, and reused. We then created two cases based on two processes that share the same pid: process 1's (p1) range overlaps with process 2 (p2), and p1 and p2's ranges do not overlap. For both cases we determine the return value of the bday function when the pid is in each of the five possible states and when the OS provides a wide variety of birthdays. From this analysis we determined the states and cases in which the bday function will provide false positives and false negatives.

The false answers that the bday function returns fall into two broad categories, false positives due to the overlapping ranges of p1 and p2, and false answers due to birthdays failing outside of their designated ranges. To prevent the first category of false positives we need to ensure that r(p1) does not overlap with r(p2). Further, we should provide a buffer between the ranges to help reduce the false positives caused by the second category. The second category is more difficult because it is a problem with our assumptions, namely that we can accurately determine the precision of birthdays returned by the OS. If we cannot accurately determine the precision of birthdays returned by the OS, we cannot completely prevent the false answers caused by category two. However, we can attempt to reduce their occurrences by artificially increasing p1's range.

We can programmatically prevent overlapping birthday ranges by introducing a midwife program that creates a process and supervises its execution for a limited period of time. The OS will not reuse a pid until the process which previously had that pid is dead and has been reaped by the parent process. Therefore, we can prevent range overlap by using a midwife to fork our processes. The midwife will not reap a process until sufficient time has passed to ensure that birthday ranges for this pid will not overlap. To calculate what *enough time* is the midwife must fork the child and query the OS for its birthday. The midwife must assume that the queried value represents the smallest possible value in the range of available birthdays, and therefore add a full range size to the returned birthday. Then the midwife must provide a buffer between the ranges, and finally ensure that if the pid were immediately reused after the buffer, the range of the new pid would not overlap into the buffer. Therefore, the required wait time the midwife must supervise this process is: queried birthday + size(r(p1)) + buffer size + size(r(p2)). After which, if the process is dead the midwife can reap it, or if the process is still alive the midwife can exit and allow the OS to reap it later. It should be noted that this only establishes the range of p1, nothing more.

Increasing the range of p1 to reduce the problems of category two is more difficult. We can always err on the side of safety by setting the acceptable range to be larger than we think is necessary. However, this increases the time that the midwife must supervise a process. Additionally, we can artificially increase the range of p1 by setting the left most (smallest) value of the range to negative infinity. We can do this because, if the process existed with the given pid then no process can exist now with the same pid and a smaller birthday. In other words, reincarnation is not allowed in a computer system. Finally, we can allow the process status query program (undertaker) to act as a intermittent supervisor to the monitored process. If the undertaker checks the process and determines that it is still alive then we can extend the birthdays we accept to include most of the unsupervised run time of the process as well. To prevent overlapping ranges and provide a buffer, the range can only be extended to the time the undertaker checked the process minus the buffer range

and r(p2).

Unfortunately, this model does not include time anomalies caused by changes to the system clock due to NTP or administration. When possible NTP slews the clock into the right position by making individual seconds shorter or longer. However occasionally it must set the clock forward or backward, this is called stepping. When stepping occurs some OS's will adjust process birthdays to ensure that a process was not born before the system booted. For example, if a system booted at 45 and a process was born at 50 and at time 60 the system time was stepped to 70, the boot time would have to be modified to 55 to prevent a skew in the uptime, and therefore the process would be born before the system booted.

We, therefore, extended our model to include time anomalies. In this case we were able to simplify the model considerably because the only times that are recorded outside of the OS are process birthdays. The range around that birthday is computed rather than stored. Therefore, we only modeled queries that occur in the alive state and in the reused state. We created 4 cases: a forward step prior to reuse, a forward step after reuse, a backward step prior to reuse, and a backward step after reuse. This again creates two categories of false answers. False negatives occur in forward and backward stepping when the anomaly and the query occur before reuse. Essentially, the OS returns a birthday in *new* time that doesn't match our stored range. False positives occur in backward stepping with the anomaly before or after the reuse and the query is after reuse. Essentially, the OS returns p2's birthday in *new* time and p2's birthday falls into r(p1)'s old time.

Both of the categories can be prevented by shifting the stored birthday for the process into *new* time. The real difficulty occurs when trying to determine that an anomaly has occurred and by what amount has the time changed. We must be able to sample some relatively constant time that changes only when a anomaly occurs. In Linux, this is the the boot time, which is adjusted when stepping the clock to ensure that it doesn't appear like the system booted in the future. Storing the boot time of the system when a process's birthday is stored will ensure that we can detect an anomaly between a process's creation and the next query of its status. The difference between the stored boot time and the queried boot time will allows us to determine how much and in which direction to shift the stored birthday.

Using the techniques in this report we will implement a midwife that provides limited supervision of the early stages of a process's execution and an undertaker that can determine whether a process is dead with reasonable certainty. Additionally, the undertaker will be able to update the range of a process's acceptable birthdays and shift the birthdays to handle time anomalies.