Paper 019-2009

The SAS® Terminator: An Ultimate Killer of UNIX Processes

Houliang Li, Frederick, MD

ABSTRACT

As a UNIX SAS user, you may not realize that when your SAS program terminates, whether voluntarily or by external force, it may leave behind one or more UNIX processes spawned by the now dead top level process. In fact, a SAS program can create multiple generations of processes. Merely killing the patriarch does not stop its children or grandchildren or great-grandchildren from running wild. This paper shows you how to safely and efficiently eliminate the entire lineage of UNIX processes started by your SAS program. In effect, you become The SAS Terminator. Now which killing mode gives you more satisfaction: methodically iterative or decisively recursive?

INTRODUCTION

In the UNIX environment, a process can spawn, or create, new processes. These new processes are called child processes of the first process, which is now a parent process. Each of the child processes can in turn spawn their own child processes, thus resulting in three generations of processes. Such reproductive activities can continue indefinitely subject only to resource limitations, much like in the natural world. You can use the UNIX **ps** command to manually track the relationships between different processes. Assuming the user name is lih, if you type the following command at the command prompt:

```
   prompt% ps –e –o user –o pid –o ppid –o args | grep lih
```

You will get something similar to the following listing (**bold** heading added by author):

```
user   pid    ppid   args (program name)
lih    24131  24128  /usr/lib/ssh/sshd
lih    24145  24142  /SAS_9.1/SAS/sasexe/sas
lih    24154  24145  /SAS_9.1/SAS/utilities/bin/motifxsassm -title SAS: sessionid
lih    24184  24142  grep lih
lih    24183  24142  ps -ef
lih    24142  24133  csh
lih    24133  24131  –ksh
```

If you look closely, you can see the following relationships among the processes, as arranged by time of creation:

$$\begin{tikzcd}
 & & & & 24145 \arrow[r] & 24154 \\
24128 \arrow[r] & 24131 \arrow[r] & 24133 \arrow[r] & 24142 \arrow[ur] \arrow[r] \arrow[dr] & 24183 \\
 & & & & 24184
\end{tikzcd}$$

Clearly, the process 24128 is the patriarch of the whole family, having started everything in the first place. On the other hand, the process 24142 is the most prolific because it created three child processes of its own.

Similarly, when you launch an interactive SAS session from the command line, you typically create multiple UNIX processes. For example, in Solaris 9, typing **sas &** at the command prompt will give you two processes. The first process is a SAS application instance, and the second process is a session manager, which is actually a child process created by the first process to handle interactive display. Using the example above, the SAS instance (process id 24145) creates the session manager (process id 24154).

Please note that a particular value will appear only once in the pid column because the Solaris operating system does not allow duplicate process ids on the same server box at the same time. However, any value can appear multiple times in the ppid column, signifying that, as a parent process, it has spawned multiple child processes. Once a child is born, it will either be childless for the rest of its life or become a parent at some point.

For various reasons, we sometimes need to terminate an active UNIX process and all the processes it has spawned. Unfortunately, killing the top-level process often does not do anything to the child processes and their descendants. For example, we can terminate the patriarch process 24128 above, but the two SAS-related processes will live on if the SAS instance (process 24145) was started in the background. Similarly, if the SAS process has spawned other child processes in addition to the session manager (process 24154), merely terminating the SAS process does not do much to those other child processes. In most cases, you need to manually track down each child process and its descendants until the entire lineage is identified, before you can actually start the killing spree. If you kill a process before killing its descendants, you may never be able to positively identify those descendants anymore. To make matters worse, if you run multiple SAS sessions at the same time along with other UNIX processes, it becomes increasingly difficult to wade through generations of similar processes.

THE ULTIMATE SOLUTION

Actually, there are at least three kinds of solutions. If you are good at UNIX shell scripting, you can write a shell script to identify a target process's descendants. If you enjoy thinking recursively, you can also use a recursive SAS macro to solve the problem. Finally, if you like to keep it simple and straightforward, you can use an iterative approach to do the slaughter. This paper will focus on the third approach, while also touching on the second method with sufficient detail to get you going.

Before we start to build our ultimate killing machine, we need to know how to find our target, just as the Terminators did in the movies. In addition, our killer should be more flexible than its cinematic cousins. Whereas the movie Terminators had one particular target preprogrammed before being dispatched into the past (or our present), our killer should be field-programmable, i.e., it should be able to target any UNIX process in the future, including SAS programs and non-SAS programs. That means it should be able to accept a parameter and use it to identify the target process. A typical scenario is using the **ps** command to find the program name or process id of the patriarch of the family of processes to be terminated. As the mastermind behind the ultimate killer, it is assumed that you know what UNIX program you want to kill before dispatching your lethal weapon. If you like, you can make things a little simpler by allowing a partial program name as long as it uniquely identifies the target process. However, to avoid killing innocent processes, using the full program name is highly recommended.

It must be noted that when we resort to The SAS Terminator, it is usually because there is no other way to quickly and efficiently identify and eliminate a whole family of problematic processes. With interactive SAS sessions, we can at least easily get rid of the SAS process and the session manager by using the menu, AFTER we have identified and killed their descendants. So by default, The SAS Terminator will be used more often on background SAS and other UNIX processes. That makes our killing mission a little easier because we can now rely on the distinct names of our SAS programs, not just the sas executable which is always named **sas**.

IMPLEMENTATION - ITERATIVE

Suppose we want to build a SAS killer that can track down the entire lineage of processes at the drop of a full program name or sufficiently unique partial name. We can define the macro like this:

```
%macro sasterminator (programname= );
   /* details */
%mend;
```

Our first step is finding the target process among all the UNIX processes owned by the mastermind, i.e., you. (It is possible to kill other UNIX users' processes if you have special privileges such as root or sudo.) We can use the following macro statement (all typed on one line) to collect the owner's processes into a text file:

```
%sysexec ps –e –o ppid –o pid –o user –o args | grep lih | grep –v grep | sort >
         /killingfield/activeprocesses.txt;
```

Next we use a DATA step to separate them into two data sets. The first data set contains all pid and ppid pairs, meaning each observation contains a parent process and its child. This data set is already sorted by ppid upon creation and will never change. The second data set holds only the target pids, i.e., the processes to be killed, so it may be expanded as new target processes are identified.

```
data pids (keep=pid ppid)
     pidstokill (keep=pid rename=(pid=ppid));
   infile "/killingfield/activeprocesses.txt" truncover;

   input ppid pid @1 line $200.;
   output pids;

   if index(line, "&programname") then
      output pidstokill;
run;
```

Logically, the target data set should start with only one observation. If it has none, that means there is no process matching the program name (or partial name) you provided, either due to a typo or because the intended program has somehow ended. On the other hand, if it has more than one observation, you have probably provided a program name (or partial name) that is not unique enough. Extreme caution is required to avoid collateral damages! We want to be the good kind of terminator, the kind Arnold worked hard to portray.

The key to the iterative approach is just that – iterative, meaning one by one, until everything is processed. In this case, we need to examine every observation in the pids data set and see if any ppid has the same value as the target pid, contained in the pidstokill data set. If we find a single observation, that means the target pid has one child process. If we find multiple observations, the target pid has created multiple children. In either case, we need to add the child process(es) to the target data set.

```
data newtargets (keep=pid rename=(pid=ppid));
   merge pids (in=all) pidstokill(in=target);
   by ppid;

   if all and target;
run;

data pidstokill;
   set pidstokill newtargets;
run;
```

It is tempting to think that we are done at this point, but we are not! Each of the child processes just identified can become parents themselves, so their children, if any, need to be tracked down and added to the hit list, and the same cycle starts anew with each generation. This is where looping comes into play. Here is the modified code:

```
%let targetobs = 1;
%let totaltargets = 1;

%do %while (&targetobs <= &totaltargets);

   data newtargets (keep=pid rename=(pid=ppid));
      merge pids (in=all)
            pidstokill(in=target firstobs=&targetobs obs=&targetobs);
      by ppid;

      if all and target;
   run;

   data pidstokill;
      set pidstokill newtargets end=last;

      if last then
         call symput("totaltargets", _n_);
   run;

   %let targetobs = %eval(&targetobs + 1);

%end;
```

There are two crucial elements to the success of the looping operations. First, we use the macro variable **`targetobs`** as a pointer to iterate over the target data set, one observation (ppid) at a time and starting from the first one. Because all newly identified targets are appended to the data set, we will never repeat any ppid, as the pointer is successively incremented until no more target is left. Secondly, we use the macro variable **`totaltargets`** to keep track of the number of observations (ppids) in the newly updated target data set. It sets the current upper boundary to the iteration of the pointer. When no new targets can be identified in the pids data set and all the existing targets in the pidstokill data set have been scrutinized, the job is done.

Now that we, or The SAS Terminator, have rounded up all the right suspects, it is time for a final showdown. This is how dramatic it gets in the programming theatre:

```
proc sql noprint;

   select ppid
      into : johnconnorandgang
      separated by " "
   from pidstokill;

quit;

%sysexec kill -9 &johnconnorandgang;
```

Essentially, we line the suspect processes up and shoot them down, all in one rapid burst of our powerful, future-proof SAS weapon. Mission accomplished. You may not see blood splatter on your screen, but there are definitely lifeless remains lying around. You may want to call up the specialized SAS body collector, the **`cleanwork`** utility, to clean up the battlefield.

IMPLEMENTATION - RECURSIVE

Like any decent Terminator, good or bad, we need to have a backup weapon. This is where the recursive approach enters the scene. Recursion is simply a programming feature whereby an object, such as a piece of code, calls itself inside its definition. It requires the support of the programming language involved. SAS has offered this programming capability in its MACRO language for a long time, and is also making it available to DATA step programming in BASE SAS beginning with SAS 9.2.

As a general rule, recursion is best suited for problems that can be broken down into smaller problems, which must be structurally identical to the original problem. The only difference between the original problem and the new, smaller problems is simply scope. The smaller problems are then further broken down into even smaller problems, and the dividing process continues until the newest problem at hand can no longer be divided. At this point, we have come to the smallest problem, and a solution emerges in the form of either an obvious answer to the newest problem or a clear proof of no answer. The process then reverses itself by backtracking from the smallest problem to successively bigger problems, while also collecting all the partial answers along the way, until it returns to the original problem with a full solution. The factorial calculation is often used as an example of recursive programming.

Having built a lethal terminator using the iterative approach, we already know that UNIX processes are just like family trees. Our iterative terminator goes down each branch of the tree until no more branches are left in search of family members. Using the recursive approach, we can simply ask that each UNIX process (like a member, or node, in a family tree) identify its own child(ren). This process recurs, i.e., is repeated, with each child as soon as it is identified, until no more child(ren) can be found, and the recursion stops.

Of course, implementing a recursive solution is never really easy. Once a UNIX program starts to spawn child processes, we will never be able to predict, or know beforehand, what names those child processes will have. The same is true of further generations down the lineage. In addition, each original UNIX program can have an entirely different lineage of programs and processes, so we have to rely exclusively on the process id instead of the program name to implement our recursion. On the other hand, we do need to first identify the process id of the target program by using the program name. Here is one solution to this dilemma:

```
%macro recursive_terminator (programname=, pid= );

%if &programname ~=  %then %do;
   /* Step I:   Identify the process id of the target program               */
   /* Step II:  Collect the target process (pid)                            */
   /* Step III: Call the macro with only the target pid and no program name: */
   /*           %recursive_terminator (pid=&targetpid);                     */
   /* Step IV:  Terminate all the collected processes (pids)                */
%end;
%else %do;
   /* Step I:   Use the current target pid to identify its child processes   */
   /* Step II:  Collect the child processes (pids)                          */
   /* Step III: Call the macro on each of the child pids just identified:    */
   /*           %recursive_terminator (pid=&targetpid1);                    */
   /*           %recursive_terminator (pid=&targetpid2);                    */
   /*           ...                                                         */
   /*           %recursive_terminator (pid=&targetpidN);                    */
%end;

%mend;

%recursive_terminator (programname=johnconnor);
```

By providing a non-blank value to the **`programname`** parameter at the initial macro call and not to any subsequent invocations, we can easily determine whether a particular macro execution is a recursive call (where the **`programname`** parameter is blank) or not. Similarly, only recursive calls will have a non-blank value for the **`pid`** parameter. This distinction enables us to perform appropriate processing steps and complete the mission successfully. Please note that the term “recursive call” in this paragraph refers to all the macro calls AFTER the initial macro invocation. Strictly speaking, every invocation of a recursive object is a recursive call.

The details of the recursive implementation are not provided in this paper, but you should be able to fill them in using the framework above. It is important to remember that recursive or not, a SAS macro is still an old-fashioned macro. If you know how many times you need to call a non-recursive macro in your SAS program to solve your particular problem, you will have no difficulty whatsoever. Now just pretend that your SAS program has been renamed to the non-recursive macro inside the same program. Suddenly, you have gone recursive! And you no longer need to worry about how many times you actually call this previously non-recursive macro because the SAS macro facility will now take care of it. As long as the DATA steps and procedures work correctly in your non-recursive macro, they should work equally well in your recursive macro. The rules on macro variable scopes, including global and local, are important in any macro programming. They deserve your special attention in a recursive macro.

DISCUSSIONS

Once you have completed, or become, The SAS Terminator, whether it is iterative or recursive, you need to think of a way to put it to good use. Of course, you can always use it in an interactive SAS session, such as an X Window client, or through the SAS Enterprise Guide if the **`allowxcmd`** option is set. You can certainly use it as a background SAS job after supplying the necessary target program name and resaving the code. However, it is most useful, and most convenient, when you utilize a UNIX shell script to launch the macro. Please see the attached sample code for detailed instructions.

Powerful as it is, The SAS Terminator can still use some improvements. For example, you can add some error checking in the macro or the script. Isn't it nice if the terminator promptly reports any errors back to you before, or even without, starting its killing spree? You may overlook them when checking the **`ps`** output, but the terminator can be made to alert you about multiple programs sharing the same target name you have thrown its way. You can also add another parameter so that you decide whether you want all processes associated with a target program to be terminated immediately, or you just want to know who they are by way of displaying them on the command line or through a timely email.

Another potential improvement is to make the terminator more flexible. The current implementation, whether iterative or recursive, requires a target program name. Providing a process id does not work. What about allowing both types of input? No self-respecting terminator should choke on such a small integer, right?

Finally, the way to dispatch the terminator via a script can also be simplified. Instead of specifying the full or relative pathname of the script, you can create an alias in your UNIX environment, for example, **`kp`** for **k**ill **p**rocesses, either in the **`.cshrc`** or **`.profile`** file. From then on, typing **`kp`** followed by a (partial) program name at the command prompt gets the job done. What a befitting image for The SAS Terminator!

If you have a philosophical disagreement with the general approach as embodied by either the movie Terminators or our SAS equivalent, you can certainly make the SAS version a little gentler. For example, instead of killing the target processes right away without as much as a blink of the eye, you can choose a less drastic mode of termination. Please check the UNIX manual page for the **`kill`** command for more details. In addition, you may want to verify that the target processes are safe to terminate, i.e., without causing unintended consequences. If a target process is killed abruptly while accessing another innocent file, that file may become your collateral damage. However, don't get carried away by your feminine side. If creating The SAS Nanny is really what you want, by all means, be my guest. Just don't call it a Terminator. Arnold calls it "a girly man," so does every version of my SAS hero.

CONCLUSION

In the Terminator movies, the ultimate hero comes in different models. Similarly, our SAS terminator can be either iterative or recursive. The implementation details can vary greatly even within the same type. In fact, our terminator does not even have to be a SAS program. Using a UNIX shell script can be just as lethal. Regardless of the implementation approach or details, building a SAS UNIX process killer will provide multiple benefits. While you are building and testing the terminator, you will gain a deeper understanding of the relationship between the SAS application and the UNIX platform it runs on, not to mention enhanced expertise on SAS macro programming and potentially recursion. Inspired by this experience, you may come up with new ideas for various utilities that save you time and efforts, even money. Once it is completed successfully, you will have a valuable and powerful tool at your disposal. The SAS Terminator will undoubtedly play an important role in maintaining a healthy and productive UNIX environment for you and your company. An added bonus is a boost to your ego. Think about it. How often do you get away with rampant killing and yet are still regarded as an ultimate hero for all that violence?

Since the movie Terminators have not succeeded yet (not sure about Terminator IV), our SAS terminator can not guarantee a certain future for the entire humankind either. What it can do is making your professional future look just a little brighter.

REFERENCES

Li, Houliang. "Building Your Own Real-Time SAS® Server Monitor under Unix" *Proceedings of the SAS Global Forum 2007*. April 2007. <http://www2.sas.com/proceedings/forum2007/010-2007.pdf>

Li, Houliang. "The Pegboard Game: A Recursive SAS® Macro Solution" *Proceedings of the Nineteenth Annual NorthEast SAS Users Group Conference*. September 2006. <http://nesug.org/proceedings/nesug06/ap/ap03.pdf>

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Houliang Li
2556 Carrington Way
Frederick, MD 21702
(301) 682-6832
houliang_li@yahoo.com

APPENDIX

```
/* The following SAS macro is a fully functioning utility. It accepts a UNIX
   program name, tracks down all the processes associated with the program, and
   terminates them all.
*/

/* Define the SAS macro */

%macro sasterminator (programname= );

/* Assign some macro variables for convenience */

%let killer = %sysget(USER);
%let killingfield = sasdisc;


/* Find and sort all the processes owned by the user and put them in a file */

%sysexec ps -e -o ppid -o pid -o user -o args | grep &killer | grep -v grep | sort >
         /&killingfield/activeprocesses.txt;


/* Put all the pid-ppid pairs into a data set and identify the target pid */

data pids (keep=pid ppid)
     pidstokill (keep=pid rename=(pid=ppid));
   infile "/&killingfield/activeprocesses.txt" truncover;

   input ppid pid @1 line $200.;
   output pids;

   if index(line, "&programname") then
      output pidstokill;
run;


/* Look for target pids by iterating over all the pid and ppid pairs */

%let targetobs = 1;
%let totaltargets = 1;

%do %while (&targetobs <= &totaltargets);

   /* Identify the pids whose ppid is the current target pid */

   data newtargets (keep=pid rename=(pid=ppid));
      merge pids (in=all)
            pidstokill(in=target firstobs=&targetobs obs=&targetobs);
      by ppid;

      if all and target;
   run;

   /* Append the new pids to the target pids data set */

   data pidstokill;
      set pidstokill newtargets end=last;

      if last then
         call symput("totaltargets", _n_);
   run;

   %let targetobs = %eval(&targetobs + 1);

%end;
```

```
/* Put the target pids into a macro variable */

proc sql noprint;

   select ppid
      into : johnconnorandgang
      separated by " "
   from pidstokill;

quit;


/* Let the killing begin */

%sysexec %str(kill -9 &johnconnorandgang;
              echo “Terminated processes (&programname): &johnconnorandgang”;
              echo “%upcase(&killer) is The SAS Terminator!”);

%mend;


/* Invoke the macro by passing in the sysparm macro variable, which will be
   supplied by a simple shell script. If you want to use the macro in an
   interactive SAS session, simply pass in the target program name.        */

%sasterminator (programname=&sysparm);


/* Here is the shell script, called kp.scr:

#! /bin/csh
# Example:  kp.scr  unique_program_name_or_partial_name

if ($#argv != 1) then
   echo "Usage: kp.scr unique_program_name_or_partial_name"
   echo "       Please try again!"
else
   sas /killingfield/sasterminator.sas -sysparm $1
endif

*/
```