Zombie Process In Linux

Posted on 2016-01-16 | In OS | Visitors:

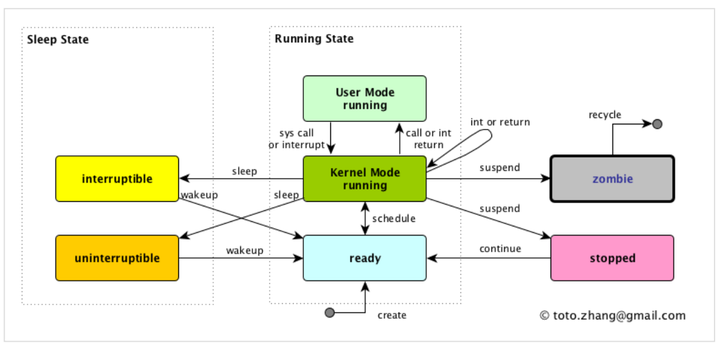
In Linux, any process that ends up running becomes a zombie process within a certain period, so a single zombie process is not inherently harmful. Only when the number of zombie processes in the system continues to accumulate and not disappear, the safety of the system will be threatened, especially in important server systems, the potential harm of the zombie process requires our special attention. So why does zombie processe exist? How does this process occur? What will happen to the system when zombie processes accumulate in large Numbers? How to avoid the potential harm of zombie process? This article discusses the above issues and briefly summarizes the process control of Linux operating system. The theory and approach to zombie processes are applicable to Solaris, BSD, and the Linux family of operating systems that conform to POSIX standards.

The life cycle of Linux Process

In the Linux operating system, any Process is created by a previous existing process, which is called the parent process of the newly created process, and the newly created process is a child process. The only exception here is the init process, which is the first process loaded by the OS kernel. Init is the root of the process tree and the status of the init process can be seen using the pstree command.

```
toto@guru:~$ pstree
init─┬─ModemManager───2*[{ModemManager}]
     ├─NetworkManager─┬─dhclient
     │                ├─dnsmasq
     │                └─3*[{NetworkManager}]
     ├─accounts-daemon───2*[{accounts-daemon}]
     ├─acpid
     ...
     ├─cron
     ├─cups-browsed
     ...
     ├─kerneloops
     ├─lightdm─┬─Xorg───{Xorg}
     │         ├─lightdm─┬─init─┬─at-spi-bus-laun─┬─dbus-daemon
     │         │         │      │                 └─3*[{at-spi-bus-laun}]
     │         │         │      ├─at-spi2-registr───{at-spi2-registr}
     ...       ...       ...    ...
     └─wpa_supplicant
```

There are several states throughout the lifecycle of a process, including **running state, sleep state, pause state, and zombie state**. Among them, **running state** is subdivided into ready state, kernel running state and user running state. **sleep state** is divided into interruptible sleep state and uninterruptible sleep state. The state transitions are as follows. Any process that creates another process needs to apply to the operating system, and after the application is approved, the new created process enters the ready state. The kernel loads and runs the new process. The process switches to the kernel running state and the user running state. When the process terminates, the process enters the zombie state and stays in the zombie state until its parent process recycles it. Other states are not covered in this article.



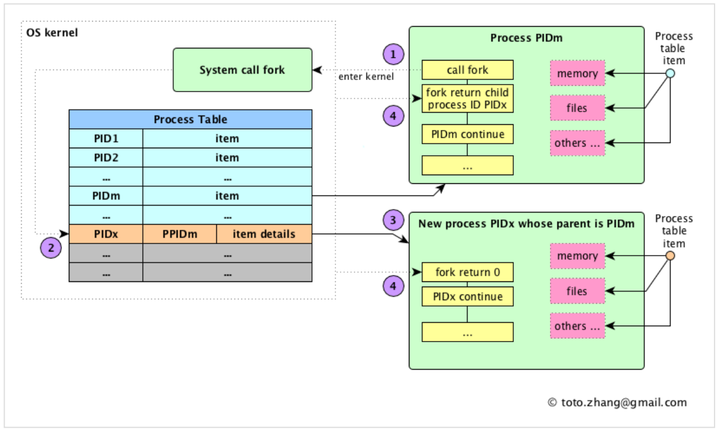
It can be told from the state transition that the zombie state is a mandatory path that a process must go through, and the zombie process is the process in the zombie state.

Process creation

As mentioned earlier, any process that creates another processe requires an application to the operating system through a fork system call. When a process calls fork, the operating system kernel adds a new item to its progress table and allocates resources for the new item, including memory resources, file descriptors, and so on. From the user’ s perspective, a new process is born. The new item in the process table describes all the information about the new process, and each field of the new item is described in the current Linux kernel using a struct named task_struct, where the field pid represents the process ID and is the unique identification of the process in the kernel process table. Usually fork() is used in conjunction with the exec() family of functions. Refer man manual or APUEv3 for details. The following figure describes the procedure of how a process called PIDm creates PIDx.

- Step 1. (1) process PIDm calls fork and enters the kernel for execution.
- Step 2. (2) the kernel assigns the contents of the task_struct structure to the new process and adds this to the process table.
- Step 3. (3) the new item describes a newly assigned process PIDx.
- Step 4. (4) the fork call returns the process PIDm from the kernel with a PIDx value.
- Step 5. (4) fork the call returns the process PIDx from the kernel with a return value of 0 to distinguish between the parent process PIDm and the child process PIDx.

PIDm and PIDx hold the same memory space, file descriptor and other resources. The differences and similarities of the resources can be referred to “man fork” .



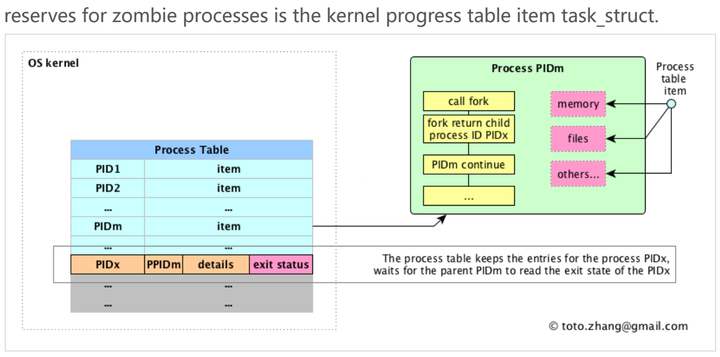
The following code creates a child process, The parent and child processes simultaneously print information to standard output.

```c
int main(int argc, char** argv)
{
    pid_t pid = Fork();

    // Child, PID is 0, STEP 5
    if (pid == 0)
    {
        // exec functions are called to start a executable programme.
        printf("child process: PID = %d\n", getpid());
        proc_child();
        exit(0);
    }

    // Parent, PID > 0, STEP 4
    // Parent continues
    printf("parent process: PID = %d\n", getpid());
    proc_parent();
    exit(0);
}
```

Process exit

There are many ways to end a process. For example, the process has some ways to end itself, such as returning from the main function, calling exit(), _exit(), _Exit() function, the last thread of the process ends or the last thread of the process calls pthread_exit() function, which will cause the process to exit. There are many ways for a process to be forced to end, such as receiving some signals such as SIGKILL, SIGABRT, SIGQUIT, SIGINT and so on, which lead to the passive exit of the process.

No matter how the process ends up running, it will eventually migrate from the kernel runtime state to the zombie state and become a zombie process. Before a process migrates to a zombie state, the kernel frees up memory and file resources occupied by the process. Therefore, the occupation of system resources by zombie processes can be ignored. The only thing the kernel reserves for zombie processes is the kernel progress table item task_struct.

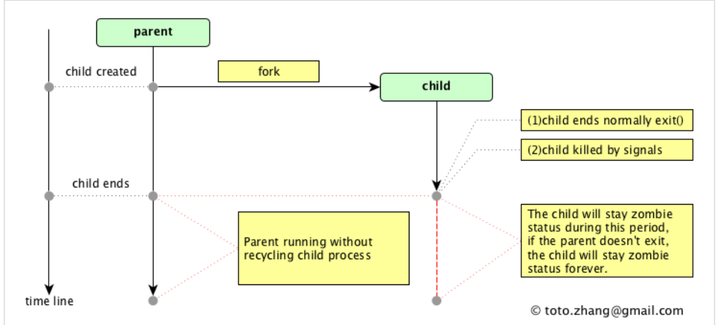


Zombie process

As can be seen from the schematic diagram of process exit, from the perspective of the kernel, the exited processes that only occupy the kernel process table item and do not occupy any system resources are zombie processes. From the user’s point of view, a process that has been terminated in some way, but whose exit status has not been rycycled by the parent process, is a zombie process.

How zombie process occur

The root cause of zombie process is that the parent process does not recycle the exited child process, resulting in the child process to become a zombie process. Use the following code to create a zombie process and view the status of the process through the ps command.



Parent process, continuously create multiple child processes, with each one executing the child program. The parent process continues to sleep and does not exit, while the child process is not reclaimed.

```c
//parent.c slice
int main(int argc, char** argv)
{
    int i = 0;
    pid_t pid[MAX_CHLD_PROC_NUM];

    for (i = 0; i < MAX_CHLD_PROC_NUM; i++)
    {
        if ((pid[i] = Fork()) == 0)
        {
            execve("./child", NULL, NULL);
        }
    }

    for(;;)
    {
        printf("parent process: PID = %d sleep.\n", getpid());
        sleep(10);
        printf("parent process: PID = %d wakeup.\n", getpid());
    }

    exit(0);
}
```

Child process，print the process ID and exit.

```c
//child.c slice
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("child process: PID = %d exit.\n", getpid());
    exit(0);
}
```

Compile and execute. After the parent process runs, it creates 10 child processes, and sleep forever. After the 10 child processes prints PID, the 10 child processes ends. We can see all these 10 childs become zombie state.

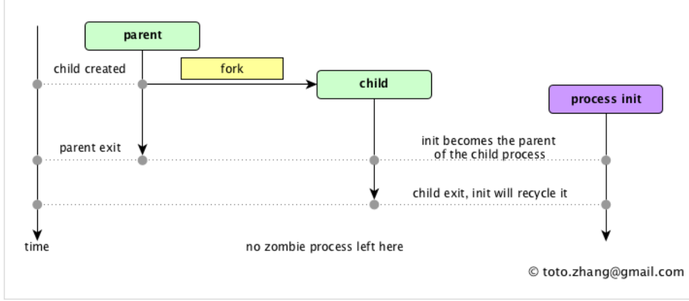
```
toto@guru:~$ gcc -Wimplicit-function-declaration --std=c99 parent.c -o parent
toto@guru:~$ gcc -Wimplicit-function-declaration --std=c99 child.c -o child

toto@guru:~$ parent
parent process: PID = 9286 sleep.
 child process: PID = 9287 exit.
 child process: PID = 9289 exit.
 child process: PID = 9288 exit.
 child process: PID = 9290 exit.
 child process: PID = 9291 exit.
 child process: PID = 9292 exit.
 child process: PID = 9296 exit.
 child process: PID = 9293 exit.
 child process: PID = 9295 exit.
 child process: PID = 9294 exit.
parent process: PID = 9286 wakeup.
parent process: PID = 9286 sleep.
...

toto@guru:~$ ps -aux | grep Z
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
toto      9287  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9288  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9289  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9290  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9291  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9292  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9293  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9294  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9295  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
toto      9296  0.0  0.0      0     0 pts/0    Z+   23:15   0:00 [child] <defunct
```

Recycle of zombie process

Normally, when using a multi-process model, we' d better ensure that the parent process recycles the child processes when the child processes exit, read the exit state of child process or explicitly ignored. Be sure to avoid situations that the parent process directly ignores the child process exit status as previous example. There are two scenarios for child process recycling.

The parent process finishes running before the child process

In this case, because there is no parent process, the child process becomes the orphan process. In POSIX standard system, the process is organized as a process tree, so the orphan process will eventually become a node of the process tree, that is, a parent process must be found. At this point, the init process of the system becomes the parent of the orphan process. If the child exits at this point, the system process init recycles it. Note that the system init process does not have to be number 1 init.



Modify the child code slightly to execute the parent process again.

```c
#include <stdio.h>

int main(int argc, char** argv)
{
    while(1)
    {
        printf("child process: PID = %d sleep.\n", getpid());
        sleep(10);
        printf("child process: PID = %d wakeup.\n", getpid());
    }

    printf("child process: PID = %d exit.\n", getpid());
    exit(0);
}
```

Process tree state

```
toto@guru:~$ pstree -p 2011
init(2011)─┬─at-spi-bus-laun(2130)─┬─dbus-daemon(2136)
           ...                     ...
           ├─gnome-terminal(2661)─┬─bash(2670)─┬─hexo(2876)─┬─{hexo}(2878)
           ...                    │            │            ...
                                  ...          └─parent(4259)─┬─child(4260)
                                                              ├─child(4261)
                                                              ├─child(4262)
                                                              ├─child(4263)
                                                              ├─child(4264)
                                                              ├─child(4265)
                                                              ├─child(4266)
                                                              ├─child(4267)
                                                              ├─child(4268)
                                                              └─child(4269)
```

Kill the parent process and view the process tree again.

```
toto@guru:~$ kill -9 4259

toto@guru:~$ pstree -p 2011
init(2011)─┬─at-spi-bus-laun(2130)─┬─dbus-daemon(2136)
           │                       ├─{at-spi-bus-laun}(2133)
           ...                     ...
           ├─child(4260)
           ├─child(4261)
           ├─child(4262)
           ├─child(4263)
           ├─child(4264)
           ├─child(4265)
           ├─child(4266)
           ├─child(4267)
           ├─child(4268)
           └─child(4269)
```

At this point, kill the child process, no more zombie process. The following is the situation of child process in the system after kill 4260 ~ 4268, I only left 4269 process stay in sleep state.

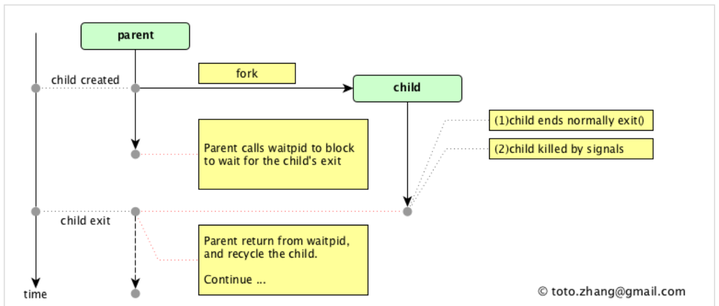
```
toto@guru:~$ ps -aux | grep child
toto      4269  0.0  0.0   4200   792 pts/9    S    07:31   0:00 [child]
```

Although init process can recycle orphaned zombie processes, when implementing multiple processes, defensive design is required to try to recycle child processes from parent processes.

The child process finishes running before the parent process

In this case, the parent process must recycle the child process itself. The system call waitpid is used for recycling. There are two methods, the first is synchronous blocking recycling, and the second is asynchronous non-blocking recycling.

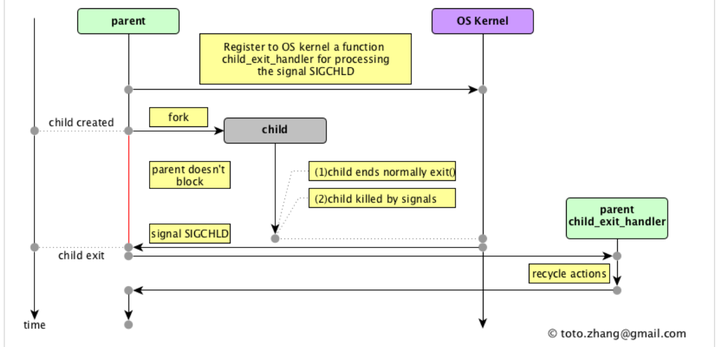
synchronous recycling, after the parent process creates the child process, waitpid is called and blocked to wait for all the child processes. After all the child processes finish excution, the waitpid unblocks and the parent process continues its processing until it exits.



The parent process code is modified.

```c
//parent.c
int main(int argc, char** argv)
{
    int   status = 0;
    pid_t ret;
    pid_t pid[MAX_CHLD_PROC_NUM];

    //10 child processes created
    for (int i = 0; i < MAX_CHLD_PROC_NUM; i++)
    {
        if ((pid[i] = Fork()) == 0)
        {
            execve("./child", NULL, NULL);
        }
    }

    //parent blocked to wait for all of the childs
    while ((ret = waitpid(-1, &status, 0)) > 0)
    {
        if (WIFEXITED(status))
        {
            printf("child process %d exit with exit status %d\n", ret, WEXITSTATU
        }
        else if (WIFSIGNALED(status))
        {
            printf("child process %d killed by signal %d\n", ret, WTERMSIG(status
        }
        else if (WIFSTOPPED(status))
        {
            printf("child process %d stoped by signal %d\n", ret, WSTOPSIG(status
        }
        else
        {
            printf("child process %d exit unknown\n", ret);
        }
    }

    printf("parent process exit\n");

    exit(0);
}
```

Asynchronous recycling, the parent process first registers with the operating system kernel the handler when the child process exits, namely the SIGCHLD signal handler, and then continues to execute its own processing flow. Until the child process exits, the operating system kernel interrupts the parent process with signals and enters the signal processing function. After signal interrupt processing is completed, the processing flow of the parent process continues.



The parent process code is modified.

```c
int main(int argc, char** argv)
{
    pid_t pid[MAX_CHLD_PROC_NUM];

    //Register OS Kernel the SIGCHLD handler
    Signal(SIGCHLD, child_exit_handler);

    //10 childs
    for (int i = 0; i < MAX_CHLD_PROC_NUM; i++)
    {
        if ((pid[i] = Fork()) == 0)
        {
            execve("./child", NULL, NULL);
        }
    }

    //Parent continues
    while(1)
    {
        printf("parent process running\n");
        sleep(2);
    }

    exit(0);
}
```

Signal handler.

```c
void child_exit_handler(int sig)
{
    pid_t ret;
    int status = 0;

    //Parent blocked to wait
    while ((ret = waitpid(-1, &status, 0)) > 0)
    {
        if (WIFEXITED(status))
        {
            printf("child process %d exit with exit status %d\n", ret,
                    WEXITSTATUS(status));
        }
        else if (WIFSIGNALED(status))
        {
            printf("child process %d killed by signal %d\n", ret,
                    WTERMSIG(status));
        }
        else if (WIFSTOPPED(status))
        {
            printf("child process %d stoped by signal %d\n", ret,
                    WSTOPSIG(status));
        }
        else
        {
            printf("child process %d exit unknown\n", ret);
        }
    }
}
```

Harm of zombie process

In a server system, if a parent process continues to spawn a zombie process, it will eventually cause the kernel’ s progress table to be filled up and the system will not be able to regenerate a new child process. The resulting phenomenon can be puzzling and difficult to locate. So the best way to avoid zombie processes is to make sure that when designing any multi-process system, the parent process takes the responsbilities to recycle the child processes. For the parent process that cannot be modified, in the process of operation and maintenance, some external automatic monitoring means should be used to constantly pay attention to the number of zombie processes, and restart the parent process that creates the zombie process when necessary, forcing the zombie process to be automatically recycled by the system init process.

‹ TCP Self Connection Problem
TCP state transmissions ›