HP-UX 11i
Technical
Whitepaper

Number of Processes and Process ID Values on HP-UX

Revision 1.0

ABSTRACT

Effective in HP-UX 11i version 3, the maximum number of allowed processes is greater than in prior versions. The value range for process identifiers (PIDs) is also increased to accommodate more processes. Some existing applications might be coded with built-in assumptions on old process and PID limits. Such applications can fail on configurations that exceed the old limits.

This paper provides information about the expanded values and limits, about how applications might depend on the prior limits, and offers some remedies.

Table of Contents

Introduction

Effective in HP-UX 11i version 3, the maximum number of allowed processes is greater than in prior versions. To support this increase, the range of values used for process IDs (PIDs) is also increased. Similarly, the range of values used for process group IDs (PGIDs) and session IDs (SIDs) is increased (since these are given the PID value of the group or session leader).

Some applications may have been coded with builtin assumptions that the number of processes and the value of PIDs/PGIDs/SIDs cannot exceed 30,000 – the maximum imposed by prior versions of HP-UX.[1] Such applications can fail on configurations that allow or create processes with IDs greater than 30,000 or more than 30,000 active processes. This paper provides information about the expanded values and limits, about how applications might depend on the prior limits, and offers some remedies.

Terminology

11i v3

This refers to version 3 of HP-UX 11i. The `uname(1)` command reports the release as B.11.31. The previous versions of HP-UX 11i referenced in this document are version 2 (B.11.23) and version 1 (B.11.11).

active processes

Total number of processes that exist simultaneously and that can be found in a list of processes produced by the `ps(1)` command. This includes defunct (zombie) processes that have not yet been waited upon by their parent process (with a function such as `wait(2)`).

CHILD_MAX, _SC_CHILD_MAX

The `CHILD_MAX` system variable, obtainable via the function `sysconf(_SC_CHILD_MAX)` or the command `getconf CHILD_MAX`, is the same value as that of the `maxuprc` system tunable parameter.

MAXPID, PID_MAX

`MAXPID` and `PID_MAX` specify the maximum value for a process identifier (PID). In past releases their value was 30,000 (before 11i v1) and 8,388,607 (11i v1, v2). In 11i v3 the value is 1,073,741,823.

maxuprc

This is the system tunable parameter that specifies the upper limit on the number of active processes associated with a single user ID. It represents the maximum per-user processes. In HP-UX 11i v3, this value can now exceed 30,000. Note that the number of active child processes for a single parent process might be limited to a smaller value.

NPROC

This is the system tunable parameter that specifies the upper limit on the number of active processes. The maximum value on OS versions prior to HP-UX 11i v3 was 30,000. For version 3, the maximum supported value is 60,000. The default is 4200.

[1] Such assumptions violate the interface specification of HP-UX 11i. Nonetheless, some old coding habits may have contributed to such application assumptions.

PGID
See process group ID.

PID
See process ID.

`pid_t`
Is the C-language programmatic data type used for declaring variables which hold PID values. It has always been and will continue to be a 32-bit integer. Its definition was introduced in HP-UX release 8.0.

process group ID
This is an integer value which is used as the handle or identifier for a process group. Its value is usually the same as the PID of the group leader process. Also known as PGID. See `setpgrp(2)`.

process ID
An integer value which is used as the handle or identifier for an active process. Values are reused as processes are destroyed and new processes created (and assigned an ID previously used by some other process). Also known as PID. Note that PIDs are also used as process group IDs (PGIDs) and session IDs (SIDs).

`process_id_min`, `process_id_max`
Kernel tunable parameters introduced in HP-UX 11i v3. They specify the lowest and highest values to use for process IDs when creating new processes. Their default values are 0 and 30,000, respectively, to provide an environment compatible with that of prior versions of HP-UX.

session ID
An integer value that is used as the handle or identifier for a session. Its value is usually the same as the process ID of the session leader process. See `setsid(2)`.

SID
See session ID

Administrator Notes

This section covers information for the system administrator.

Kernel Tunable Parameters

Several kernel tunable parameters control the number of active processes and the values used for PIDs:

- `nproc` – Specifies a limit on the total number of active processes in the system. The maximum value for this parameter on prior HP-UX versions was 30,000. For 11i v3, it is 60,000.[2] Setting values higher than 30,000 can cause some applications to fail.

[2] The software might allow the `nproc` value to exceed 60,000. However, larger values have not been tested by HP and are not supported.

- `process_id_max` – Specifies the maximum value the system should use for PIDs assigned to new processes. Changing it does not affect PIDs already assigned. This tunable parameter first appears in 11i v3. The default value is 30,000 to provide an environment compatible with that of HP-UX versions prior to 11i v3. Increase this tunable when more than 30,000 active processes are needed or if you want PIDs to be reused less frequently. However, increasing it to a value higher than 30,000 can cause some applications to fail.
- `process_id_min` – Specifies the minimum value the system should use for PIDs assigned to new processes. Changing it does not affect PIDs already assigned. This tunable paramter first appears in 11i v3. The default value is 0. The purpose of this tunable is to support program qualification. It allows the creation of an environment where large values are used for all PIDs. This environment can then be used to validate, in part, that programs can properly handle large PID values. When this tunable is increased, the system should be rebooted to ensure all processes have large PID values. For program validation, the recommended values for `process_id_min` and `process_id_max` are 1,000,000,000 and 1,070,000,000, respectively.

Cautions

As noted in the previous section, setting tunable limits on PID values, or the number of active processes to values higher than 30000, can cause some applications to fail. There are several critical maximum values:

- 30,000 – applications with hard-coded dependencies on the old limits of 30,000 can fail if PID values or the number of processes exceed 30,000
- 32,767 – applications with stored PID values in 16-bit signed integer variables can fail if PID values exceed 32,767. Applications that maintain counts of processes in 16-bit signed integer variables can fail if the number of processes exceeds 32,767.
- 65,535 – applications with stored PID values in 16-bit unsigned integer variables can fail if PID values exceed 65,535. Applications that maintain counts of processes in 16-bit unsigned integer variables can fail if the number of processes exceeds 65,535.
- 99,999 – applications with a builtin assumption that PID values do not exceed 5 decimal digits can fail if PID values exceed 99,999. PID values with more than 5 decimal digits can also cause display format issues (such as uneven columns, fields that run together, and so on).

These are the most common dependencies on the number of processes or the PID values. Programs can have other dependencies. The next section describes how correct programs should handle the values, some of the incorrect assumptions programs might make, and how to detect them in program source.

Programmer Notes

This section describes aspects of PID values and the number of processes useful to program developers. Programmers should also be familiar with the Administrator Notes in the previous section.

Correct Handling of PID and Number of Processes

This section describes how correct programs handle PID values and the number of processes. Note that what applies to PID values also applies to PGIDs and SIDs.

Use pid_t for PID, PGID, SID Values

The `pid_t` data type should be used in the declaration of all variables that hold PID, PGID, or SID values.[3] It has sufficient precision to hold any PID value.

Use 32-bit int for Process Counters

Programs that maintain counts of the number of active processes in the system, or any subset thereof, should use the 32-bit `int` or equivalent data type to hold these counts. This provides more than adequate precision so that overflows are not a problem.

Comparision Checks on PIDs

PID values should be compared only for equality (or inequality). Never check for PID values greater than or less than other PID values. Never check for specific PID values because there is no association between specific values and specific processes.

One exception is the PID of the system initialization process (referred to as `init`). The PID value reported by interfaces such as `getppid(2)` is always 1 if the process is the init process. It is thus appropriate to check for a return value of 1 from this interface.

Validity checks on a PID value are usually unnecessary. When desired for program self-checking, the only validity check appropriate to make is that the PID is a positive value. (Note that the `kill()` function can be used to determine whether a process having that PID exists, and is accessible by the caller, by passing 0 for the signal.)

Use Data Structures Other Than Arrays for Tracking

PID values should not be used as array indexes. PID values can exceed 1 billion (1,000,000,000), but the number of valid PIDs at any point in time is relatively small (never greater than the value of `nproc`). When tracking PID values, it is better to use other data structures such as hash tables. Further, because the PID range is dynamically tunable, new PID values might exceed the size of a previously created array.

Print/Display Formats for Ten Decimal Digits

Any print or display formats should be able to handle ten (10) decimal digits, since PID values can range to greater than 1 billion.

Incorrect Handling

This section describes how some programs have built-in assumptions on prior PID or active process limits.

Using 16-Bit short or unsigned short for PIDs

A `short` data type variable has only 16 bits, and can represent values only up to 32,767 (signed) or 65,535 (unsigned). Some programs might be using `short` variables to hold PID values. Such programs fail for PID values that exceed the precision of such variables.

---

[3] The `pid_t` data type has been the specified type to use since HP-UX release 8.0. Programs that use a 32-bit `int` type support the expanded range of values properly. However, HP recommends that the `pid_t` data type be used as specified in the manpages. This will ensure source compatibility with any future expansion in the size of the data type.

Caution: It is possible that a type other than short is used, but where the base type is short. The additional system-defined types (from sys/types.h and sys/_inttypes.h) include: int16_t, uint16_t, int_least16_t, uint_least16_t, ubit16, sbit16,cnt_t,nlink_t,use_t. There might be others in headers included by the specific program's source code.

Using 16-Bit Data Types for Process (or Thread) Counts

A program might have some short counter whose value is a function of the number of processes. An obvious example is a counter in a loop which calls pstat_getproc(2). Such programs fail if the number of processes exceeds the precision of such a counter.

Checking Against MAXPID, PID_MAX, or 30,000

The MAXPID and PID_MAX constants specify the maximum value for a process identifier (PID). For 11i v3 their value is 1,073,741,823. In releases prior to 11i v1, their value was 30,000. For 11i v1 and v2, their value was 8,388,607 (however, actual PID values generated by these versions never exceeded 30,000).

The MAXPID symbolic constant is undocumented but, nonetheless, it has been used in some programs as the maximum PID value against which other PID values are validated. The PID_MAX symbolic constant is documented but with the provision that the "Actual limit might be greater than specified value on certain HP-UX systems" (see limits(5)) . Thus, these symbols should not be used directly in application program code.

Some programs might use these constants to validate PID (or PGID or SID) values. Such programs, if compiled on releases prior to 11i v3, fail such validations on systems where PID values are larger than the MAXPID and PID_MAX value on the release under which they were compiled.

Some program code might equate these values with the maximum number of processes in the system. These programs can fail if compiled on releases before 11i v1 and if the number of processes becomes greater than 30,000.

Some programs use the constant 30,000 rather than MAXPID or PID_MAX. These programs are also vulnerable to failure.

Using Arrays to Track Processes

Because previous releases limited the maximum PID value and number of active processes to 30,000, some programmers found it convenient to use an array to track processes. The array can simply be indexed by the PID. Assuming 4-byte elements (32-bit pointers), such an array consumes less than 128 K of virtual memory.

Such an array now needs to be 4 GB, since the maximum PID is 2^{30}-1. Allocating such a large array is likely to waste address space and to cause increased swap space reservation (unless allocated in a lazy swap region). It also can increase the need for RAM and paging (depending on the distribution of PIDs, the page size, and available RAM), thereby resulting in lower performance.

Five Decimal Digits

Because PID values traditionally do not exceed 30,000, some programs might be dependent on the notion that PID values do not exceed five (5) decimal digits. Issues can arise if PIDs are larger than 99,999 for the following reasons:

- Programs that parse output from another program based on character position in the line can fail to parse it correctly when that output contains PID values.

- Poor aesthetics in output because of misalignment of columns when PIDs use more than the expected five character positions.
- Poor aesthetics in output because of wrapping around on the standard 80 columns, where it did not previously do so, due to PIDs taking more than five character positions.
- Running together of adjacent fields making output unreadable. For example, In `sprintf("%6d%6d%6d", x, pid, y)` the values of the variables `x` and `pid` will run together if the PID exceeds five digits.

Temporary File Name Parsing

Programs should never attempt to parse the temporary file names that are generated by functions such as `mktemp(3)`. The traditional decimal encoding of the caller's PID value is not used when PID values exceed five decimal digits.

Special PIDs

Process ID values should not be compared with any literal or symbolic constant. With one exception, there are no documented special PID values.

The exception, as described in "Comparison Checks on PIDs" is for the system initialization process.

Additional Issues

pstat_getfile()

The `pstat_getfile`() function encodes only a 16-bit process index into one of its arguments. This interface is obsolete. Programs must now use its replacement, `pstat_getfile2`().

shmctl()

Prior to HP-UX 11i v1 (B.11.11) the `shmid_ds.shm_nattch` and `shmid_ds.shm_cnattch` fields as reported by `shmctl(IPC_STAT,…)` were 16-bit unsigned integers. This overflows if the number of processes attached to a particular shared memory object exceeds 65,535.

It is unlikely for any application to have more than that number of processes sharing a memory object.[4] Still, it is good practice to recompile any application that uses `shmctl()` on 11i v1 or later.

_CLASSIC_TYPES

The use of the `_CLASSIC_TYPES` compile-time switch is obsolete (since HP-UX 8.0). However, some programs might still be using it. Such programs might encounter problems related to having PID values reported in 16-bit fields.

Performance of Programs Using pstat_getproc()

Programs that use the `pstat_getproc`() function do so to obtain information about processes in the system. If that information is stored, the data structures used should be evaluated for performance and scaling. Also, consider increasing the buffer passed to this function, so that information for more processes can be retrieved in each call (thereby decreasing the number of calls that are needed).

[4] Because the 11i v3 documented maximum processes in the system (limit specified by the `nproc` kernel tunable parameter) is 60,000, indeed, very unlikely.

Some experimental evidence suggests a buffer of 200 entries is optimal when a large number of active processes are present.

nproc

This is the name of the kernel tunable parameter that specifies the maximum number of simultaneous active processes that can be created on the system. Programs that reference this symbol (for example, via `nlist()`, `tuneinfo()` or the `kmtune` command) might have some dependency on the old limits. Because `nproc` is also dynamically tunable, programs should be prepared for this value changing during operation.

Algorithm Scaling

In rare cases, a large number of processes in the system can affect the performance and scaling of an application program. A program algorithm designed to work with no more than 30,000 (or fewer) processes might not perform well when that number is exceeded.

For instance, consider a program that tracks processes in the system. Assume it internally maintains a hash table for quick look-up of processes. If that hash table is sized optimally for 500 processes, it might not work well for 60,000. Indeed, it might already not work well for 30,000 processes. To address this, it might be sufficient to simply increase the size of the hash table.

Detecting Program Sources That Make Incorrect Assumptions

Programmers can detect some source code issues using `lint` or similar tools. These tools will report loss of precision or accuracy when PID values returned by functions such as `fork()` are stored in short data types. Some versions of the C compiler also note such issues when the `+w1` option is used to increase warnings.

Programmers can also scan code with tools such as `grep` to find sequences that need further investigation for dependencies on PIDs. Following is a list of regular expressions that can be useful in scanning program sources:

```
“short”, “u*int16_t”,  “u*int_least16_t”, “[us]bit16”, “cnt_t”,
“nlink_t”, “use_t” , “pstat_getproc“, “pstat_getfile“, “MAXPID”,
“PID_MAX”, “shm_c*nattch”,  “30000”, “_CLASSIC_TYPES”, “nproc”,
“fork”
```

From the output of the scan , you can determine whether any of the references are declarations for variables that are be used to hold PID values or process counts.

Searching for `short` (or the other types) might produce many false “hits.” If so, consider changing the expressions to look for the substring `id` or `cnt` or `count` on the same line (for example. `short[[:space:]].*id`). The check for `id` assumes that most programmers will include that two character sequence in the name of any variable used to hold PID, PGID, or SID values. The check for `cnt` or `count` makes the assumption that most programmers will include those sequences in the name of any variable used to count instances of objects such as processes.

Testing

After any program source code update, it is always necessary to validate it for proper operation. That validation should be carried out in the expected operational environments. The following sections describe how to set up environments with large PIDs or with a large number of processes in order to adequately validate updates to support such environments.

Testing Large PIDs

To validate that programs properly handle large PID values, HP-UX 11i v3 adds the ability to make all PID values large. To do so requires that the system tunable parameters `process_id_min` and `process_id_max` be set to large values (for example, 1000000000 and 1070000000, respectively). With these settings, programs can be validated in an environment where all processes have large PID values.[5] [6] It is recommended that a reboot be done after setting these parameters, so that all existing processes are re-created with larger PIDs. For more information, see the associated manpages.

Testing Large Numbers of Processes

To validate that programs properly handle large numbers of processes requires that a small program be written to create lots of processes. The program being validated can then be executed once those processes are created. The following example is an outline of such a program:

```
// sample program to create 50626 processes
// the program can be stopped by sending
// SIGTERM signal to main process

sig_handler() {
      relay SIGTERM to all my child procs
      exit
}

create_children()
      level++
      if (level < 4) {
            setsid()  // start new process group and session
            for 15 iterations {
                  fork()
                  if child {
                        create_children()
                  }
                  if parent {
                        record child pid for later
                  }
            }
            for 15 iterations {
                  // children will exit on SIGTERM
                  wait() for children
            }
            exit
      }
```

[5] Except for certain kernel system processes, which always have very low PID values.

[6] For some programs, it may be valuable to also test in environments where some PID values are very small and some are very large. For these, create some test processes before and after increasing `process_id_min`, and don't reboot before completing the tests.

```
        else {
              // leaf level in process tree
              // just wait for signal
              sigpause()
              exit
        }
  }

  main() {
        setup SIGTERM signal handler
        setup SIGINT handler
        level = 0
        create_children()
  }
```

Be sure that your system has sufficient memory to support a large number of processes. Approximately 256 K physical memory per process is recommended for this experiment.[7] Thus, 4 GB is required for the test program, in addition to that needed for other system activity and the application to be validated.

For more information

For additional information see nproc(5), process_id_max(5), process_id_min(5), pstat(2), pstat_getproc(2), pstat_getfile2(2), limits(5).

[7] The memory requirement on a per-process basis varies with the specific demands of the applications executed by those processes. The per-process memory required for this experiment should not be considered a general rule.

Part number: 5991-6426 January 2007