HP-UX Process Management

White Paper

Version 1.3

5965-4642

Last modified April 7, 1997

Legal Notices

The information contained within this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein nor for incidental consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government Department is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

First Edition: April 1997 (HP-UX Release 10.30)

1 Process Management

Objectives

Understanding how HP-UX manages processes will help you better interpret how your system carries out its computations. This white paper/chapter discusses:

- What a process is.
- What kernel threads are.
- Process creation, execution, and termination.
- Process-management structures for process scheduling, run queues, scheduling, context switching.

What is a process?

A process is a running program, managed by such system components as the scheduler and the memory management subsystem. Described most simply, a process is a container for a set of instructions that carry out the overall task of a program.

A process consists of text (the code that the process runs), data (used by the code), and stack (temporary data storage for when the process is running). These and other elements are known as the process context.

NOTE

HP-UX is now a threads-based operating system. This affects how we view processes.

Every process has at least one thread. Think of a process as a container for groups of threads. A process holds the address space and shared resources for all the threads in a program in one place. When you manipulate the elements of a program (things you schedule, context switches, and so forth), you always manipulate threads.

Two stacks are associated with a process, kernel stack and user stack. The process uses the user stack when in user space and the kernel stack when in kernel space.

Although processes appear to the user to run simultaneously, in fact a single processor is executing only one process at any given instant.

A process's `proc` structure includes:

- The program's kernel data structures (variables, arrays, records) .
- Process ID, parent process ID, process group ID.
- Process user and group IDs (both real and effective IDs).
- Group access list.
- Information on the process's open files.
- Process's current working directory.
- Audit ID (on trusted systems only).

Process Relationships

Processes maintain hierarchical, parent-child relationships. Every process has one parent, but a parent process can have many child processes. Processes can create processes, which in turn, can create more processes.

A child process inherits its parent's environment (including environment variables, current working directory, open files). Also, all processes except system processes (such as `init`, `pagedaemon`, and `sched` (the "swapper")) belong to process groups, which are explained shortly.

Process and Parent Process IDs

When a process is created, HP-UX assigns the process a unique integer number known as a process ID (PID). The HP-UX kernel identifies each process by its process ID when executing commands and system calls.

A process also has a parent process ID (PPID), which is the PID of the parent process. You can see the PIDs and PPIDs of processes currently running on your system by using the ps command.

User and Group IDs (Real and Effective)

In addition to the process ID, a process has other identification numbers:

- real user ID
- real group ID
- effective user ID
- effective group ID.

Real user ID is an integer value that identifies the owner of the process -- that is, the user ID of the user who invoked the process. Real group ID is an integer value that identifies the group to which the user belongs. The real group ID is shared by all users who belong to the group. It allows members of the same group to share files, and to disallow access to users who do not belong to the group.

The id command displays both integers and names associated with real user ID and real group ID. The `/etc/passwd` file assigns the real user ID and real group ID to the user at login.

The effective user ID and effective group ID allow a process running a program to act as the program's owner while the program executes. The effective IDs are usually identical to the user's real IDs. However, the effective user ID and group ID can be set (setuid and setgid) individually to allow limited superuser capability, by making the effective IDs of the program's processes equal to the real IDs of the program's owner. The classic example is passwd, which allows limited ability to change /etc/passwd.

The effective IDs are used to allow a user to access or modify a data file or to execute a program in a limited manner. When the effective user ID is zero, the user is allowed to execute system calls as superuser.

The effective user and group IDs remain set until:

- The process terminates.
- The effective IDs are reset by an overlaying process, if the setuid or setgid bit is set (see exec(2)).
- The effective, real, and saved IDs are reset by the system calls setuid, setgid, setresuid, or setresgid.

In a trusted system, each user has an audit ID that does not change, even when the user executes programs using a different effective user ID.

Process Context

The context of a process includes its component parts and kernel resources, as detailed in the following table. Note, however, that some of these elements are now defined in terms of threads.

Table 1-1 Context of a process

Type	Purpose
text	Machine/program code that is executed. Maximum size of text is limited by the maxtsiz configurable parameter.
Data	Initialized data that accompanies the text. Contains the global variables used by the program. Maximum size of data is limited by the maxdsiz configurable parameter.
bss	Uninitialized data

Type	Purpose
heap space	Undefined space available to the process as by a call to `malloc(3)` function.
private memory mapped files (`mmap`)	`mmap` files allow applications to map file data into a process's virtual address space.
user and kernel stacks	User stack is used while a thread is executing in user mode to store variables, subroutines; used for function calls in the user program. Maximum size of the process's user stack is limited by `maxssiz` configurable parameter. The kernel stack is used for function calls when the process executes system calls, etc.
user structure (`uarea`)	Per-process information needed when the process is running (and thus can be paged out). Points to process register state, open file descriptors, open devices, system call arguments and return values. Since HP-UX is a threads-based kernel, it has one user structure for each thread structure; each kernel thread has its own `uarea`.
shared libraries	Used by processes to reduce the amount of memory consumed. Shared library functions are mapped into processes.
shared memory	Available range of addresses that are allocated using the shared memory interfaces and that enable processes to share address space. Useful for communicating between processes.

Type	Purpose
process structure	Remains memory resident. Contains per-process information needed to schedule threads. Includes process ID, scheduling priority, run state, signal mask.
register context	Register values and program counter. When a process is running, reg. context is loaded in CPU registers; when not running, register context is stored in the process control block (pcb). Register context is now thread context.
virtual address space	A four-gigabyte (32-bit) range of addresses into which the context of a process is logically mapped.

Space apportioned to a Process

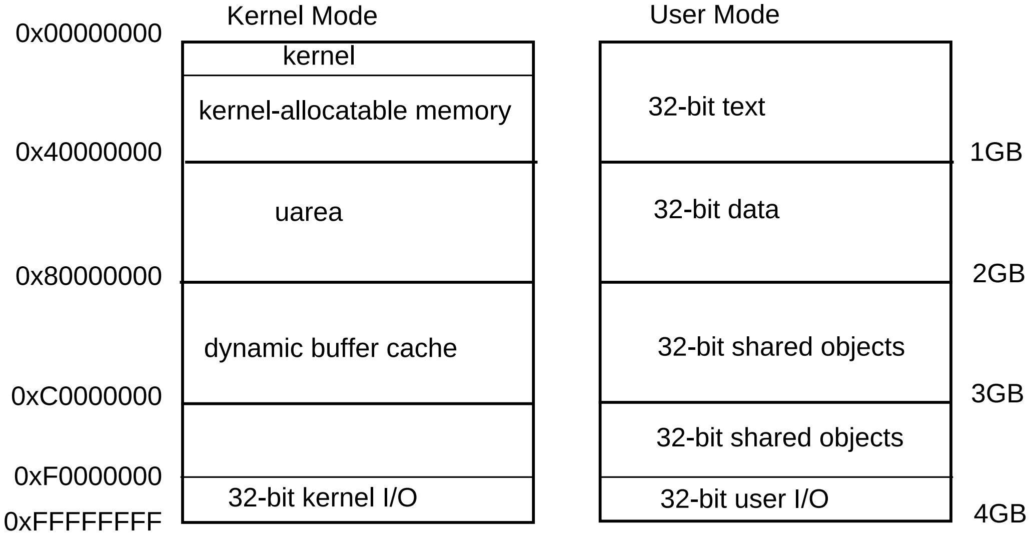
Every process has 4 gigabytes of virtual address space available to it. The four-gigabyte space is subdivided into four one-gigabyte quadrants, each with a different usage. A process keeps track of its quadrants through space registers, which provide provide quadrant locations.

Table 1-2 **Quadrants of virtual address space**

Quadrant	Address Range and Purpose
1	0x00000000 - 0x3FFFFFFF Contains process code (and EXEC_MAGIC data)
2	0x40000000 - 0x7FFFFFFF Contains process data.
3	0x80000000 - 0xBFFFFFFF Contains shared memory, shared memory mapped files, and shared library text.
4	0xC0000000 - 0xFFFFFFFF Same usage as quadrant 3. Bottom of quadrant 3 contains the gateway page. Address range 0xF0000000 to 0xFFFFFFFF is reserved for I/O address space.

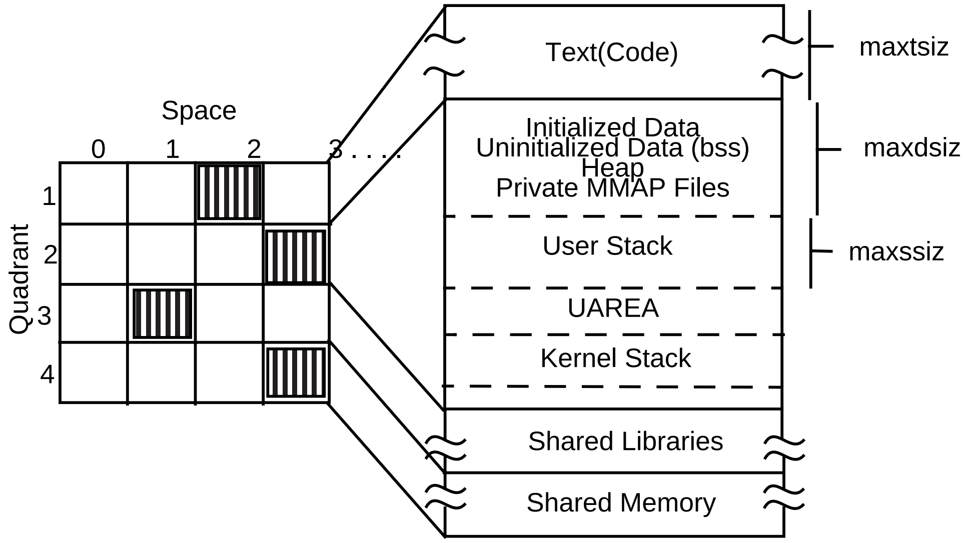
The current 32-bit address space layout can be depicted by comparing how that virtual address space is used in kernel mode and user mode.

Figure 1-1 32-bit address space layout on PA1.x



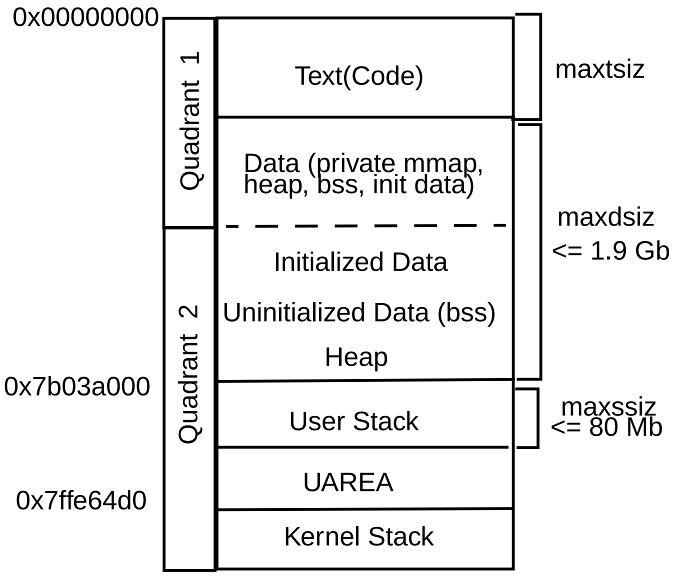
The next figure shows that the various quadrants of a process might be distributed like a patchwork in the virtual address space. (the text might be in space ID 2 of quadrant 1, the shared libraries might be in space ID 1 of quadrant 3, and so on.) Each quadrant, however, handles a specific portion of the thread of execution.

Figure 1-2 Process structure layout, showing context and space use.



Handling maxdsiz with EXEC_MAGIC

Figure 1-3 Process address space in `EXEC_MAGIC` format



By default, a process's four GB address space is divided into four equal quadrants of one GB each. Each quadrant has a specific purpose (text, data, etc) and this accounts for the limitation on maxdsiz of ~960 MB.

An EXEC_MAGIC user executable (a.out) format allows data to start immediately after the code area in the first quadrant, instead of at the beginning of the second quadrant, and grow to the beginning of the user stack. Executables created with this format can handle more than 1 GB of data.

Data space starts in the first quadrant right after the process text and runs through the second quadrant. The data space uses up to approximately 1.9 gigabytes. For EXEC_MAGIC executables, the ~1.9GB limit represents the area from 0x00000000 to 0x7b03a000. The remainder is used for the stacks and uarea. For SHARED_MAGIC, data begins at 0x40000000. Everything else remains at the same location; the maximum stack size (maxssiz) remains 80MB.

NOTE To create an executable in EXEC_MAGIC format, link the executable with the -N option. (See ld(1) man page for details.)

EXEC_MAGIC executables can access more than .9GB of process private data because data is allowed to immediately follow text in quadrant one. For text and data to occupy the same quadrant, they must have the same space, yet process private data must have a unique space ID. Therefore, in the previous implementation of EXEC_MAGIC, text was actually viewed as a part of the data. Because HP-UX currently supports only one virtual translation per physical page (that is, one <space,offset> pair) , EXEC_MAGIC text cannot be shared between multiple processes. To overcome this limitation, the EXEC_MAGIC implementation allows read-only aliasing; multiple addresses can map the same physical page.

Because only one process actually owns a page translation, true copy-on-write is not currently implemented on HP-UX. When a second process attempts to read or write a shared page, the second process receives its own private copy of the page. With read-only aliasing, processes share a text page with different virtual addresses if they are only reading the page. A process will receive its own private copy of the page only when a write is performed.

Because EXEC_MAGIC text segments were considered part of the data segment, the text segment was writable. Because HP-UX guarantees swap space be available whenever a process requires, swap space was reserved for the entire text segment at exec() time. Because most users do not write to their text segment, the swap space reserved for the process is never used. To make more efficient use of swap space, a lazy swap reservation policy is implemented for EXEC_MAGIC text. Swap space is only reserved for pages being written to.

EXEC_MAGIC text is entirely loaded in at exec() time to protect against a.out modifications while the application is being run. EXEC_MAGIC guards against this problem by marking the EXEC_MAGIC executable VTEXT. This allows the text to be demand loaded, instead of being loaded entirely at exec() time.

Null reference semantics are supported on EXEC_MAGIC executables.

NOTE If you have an existing program that you have no source for or do not wish to recompile/link as EXEC_MAGIC, the process will retain a limit of ~960 MB for its data size.

Impact and Performance

- More memory and swap space is available when running multiple copies of the same EXEC_MAGIC executable because unmodified text pages are now shared.
- EXEC_MAGIC executables start up more quickly because text is now demand paged instead of being entirely loaded at exec() time.
- EXEC_MAGIC executables execute more quickly because pages are copied on write instead of being copied on any access.
- EXEC_MAGIC application failure is visible while running, if swap space is unavailable when the procress attempts to write to a page of text. This failure is not visible if self-modifying code is not used.
- Demand paging and copy-on-write features improve performance of EXEC_MAGIC. Performance on SHARED_MAGIC executables remain unaffected.

CAUTION An EXEC_MAGIC application is subject to fail if insufficient swap space is available when the application attempts to write to a page of text. You must use self-modifying code to see this failure.

Process States and transitions

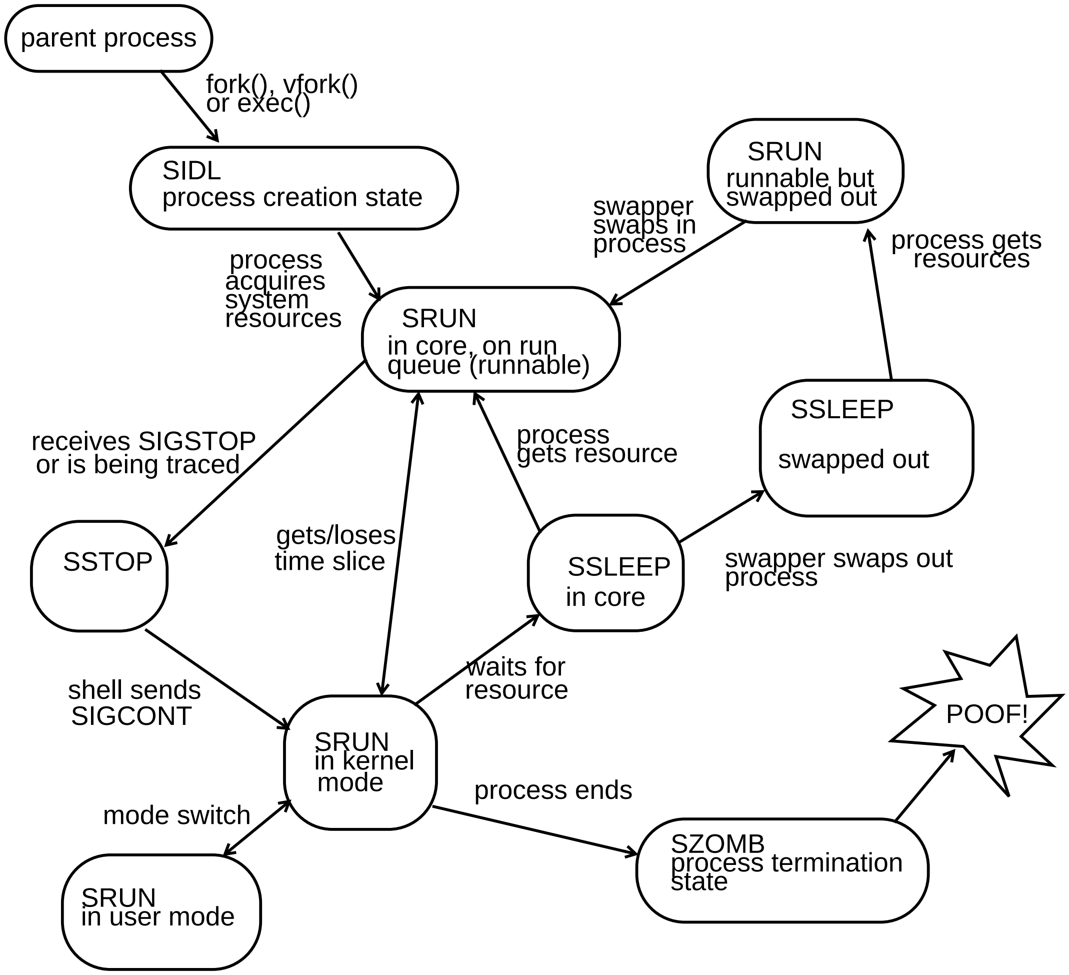
Through the course of its lifetime, a process transits through several states. Queues in main memory keep track of the process by its process ID. A process resides on a queue according to its state; process states are defined in the proc.h header file. Events such as receipt of a signal cause the process to transit from one state to another.

Table 1-3 **Process states**

State	What Takes Place
idle (`SIDL`)	Process is created by a call to `fork`, `vfork`, or `exec`; can be scheduled to run.
run (`SRUN`)	Process is on a run queue, available to execute in either kernel or user mode.
stopped (`SSTOP`)	Executing process is stopped by a signal or parent process
sleep (`SSLEEP`)	Process is not executing; may be waiting for resources
zombie (`SZOMB`)	Having exited, the process no longer exists, but leaves behind for the parent process some record of its execution.

The following figure demonstrates something of the transitions between process states.

Figure 1-4 Process states and transitions



When a program starts up a process, the kernel allocates a structure for it from the process table. The process is now in idle state, waiting for system resources. Once it acquires the resource, the process is linked onto a run queue and made runnable. When the process acquires a time-slice, it runs, switching as necessary between kernel mode and user mode. If a running process receives a SIGSTOP signal (as with control-Z in vi) or is being traced, it enters a stop state. On receiving a SIGCONT signal, the process returns to a run queue (in-core, runnable). If a running process must wait for a resource (such as a semaphore or completion of I/O), the process goes on a sleep queue (sleep state) until getting the resource, at which time the process wakes up and is put on a run queue (in-core, runnable). A sleeping process might also be swapped out, in which case, when it receives its resource (or wakeup signal) the process might be made runnable, but remain swapped out. The process is swapped in and is put on a run queue. Once a process ends, it exits into a zombie state.

What are Kernel Threads?

A process is a representation of an entire running program. By comparison, a kernel thread is a fraction of that program. Like a process, a thread is a sequence of instructions being executed in a program. Kernel threads exist within the context of a process and provide the operating system the means to address and execute smaller segments of the process. It also enables programs to take advantage of capabilities provided by the hardware for concurrent and parallel processing.

The concept of threads is interpreted numerous ways, but to quote a definitive source on the HP-UX implementation (S.J. Norton and M.D. DiPasquale, '*ThreadTime: Multithreaded Programming Guide*, (Upper Saddle River, NJ: Prentice Hall PTR, Hewlett-Packard Professional Books), 1997, p.2):

> *A thread is "an independent flow of control within the process", composed of a [process's register] context, program counter, and a sequence of instructions to execute. An independent flow of control is an execution path through the program code in a process. The register context and program counter contain values that indicate the current state of program execution. The sequence of instructions to execute is the actual program code.*

Further, threads are

- A programming paradigm and associated set of interfaces allowing applications to be broken up into logically distinct tasks that when supported by hardware, can be run in parallel.
- Multiple, independent, executable entities within a process, all sharing the process' address space, yet owning unique resources within the process.

Each thread can be scheduled, synchronized, prioritized, and can send and receive signals. Threads share many of the resources of a process, eliminating much of the overhead involved during creation, termination, and synchronization.

A thread's "management facilities" (register context et al) are used to maintain the thread's "state" information throughout its lifetime. State information monitors the condition of an entity (like a thread or process); it provides a snap-shot of an entity's current condition. For example, when a thread context switch takes place, the newly scheduled thread's register information tells the processor where the thread left off in its execution. More specifically, a thread's program counter would contain the current instruction to be executed upon start up.

As of release 10.30, HP-UX has kernel threads, which change the role of processes. A process is now just a logical container used to group related threads of an application. Each process contains at least one thread. This single (initial) thread is created automatically by the system when the process starts up. An application must explicitly create the additional threads.

A process with only one thread is a "single-threaded process." A process with more than one thread is a "multi-threaded process." Currently, the HP-UX kernel manages single-threaded processes as executable entities that can be scheduled to run on a processor (that is, each process contains only one thread.) Development of HP-UX is moving toward an operating system that supports multi-threaded processes.

Comparison of Threads and Processes

The following lists process resources shared by all threads within a process:

- File descriptors, file creation mask
- User and group IDs, `tty`
- Root working directory, current working directory
- semaphores, memory, program global variables
- signal actions, message queues, timers

The following lists thread resources private to each thread within a process:

- User registers
- Error number (errno)
- Scheduling policy and priority
- Processor affinity
- Signal mask
- Stack
- Thread-specific data

- Kernel `uarea`

Like the context of a process, the context of a thread consists of instructions, attributes, user structure with register context, private storage, thread structure, and thread stack.

Two kernel data structures -- `proc` and `user` -- represent every process in a process-based kernel. (The `proc` structure is non-swappable and user is swappable.) In addition, each process has a kernel stack allocated with the user structure in the `uarea`.

A threads-based kernel also uses a proc and a user structure. Like the proc structure of the process-based kernel, the threads-based proc structure remains memory resident and contains per-process data shared by all the kernel threads within the process.

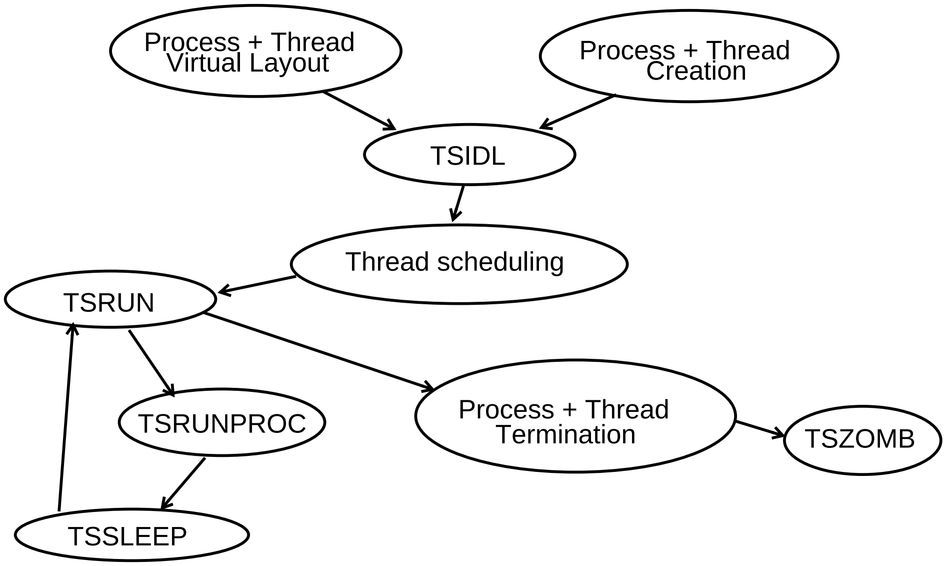
Each thread shares its host process' address space for access to resources owned or used by the process (such as a process' pointers into the file descriptor table). Head and tail pointers to a process' thread list are included in the proc structure.

Each thread manages its own kernel resources with private data structures to maintain state information and a unique counter. A thread is represented by a kthread structure (always memory resident), a user structure (swappable), and a separate kernel stack for each kernel thread.

Every kthread structure contains a pointer to its associated `proc` structure, a pointer to the next thread within the same process. All the active threads in the system are linked together on the active threads list.

Like a process, a thread has a kind of life cycle based on the execution of a program or script. Through the course of time, threads like processes are created, run, sleep, are terminated.

Figure 1-5 Typical thread state sequence


Process + Thread Virtual Layout
Process + Thread Creation
TSIDL
Thread scheduling
TSRUN
TSRUNPROC
Process + Thread Termination
TSZOMB
TSSLEEP

User and Kernel Mode

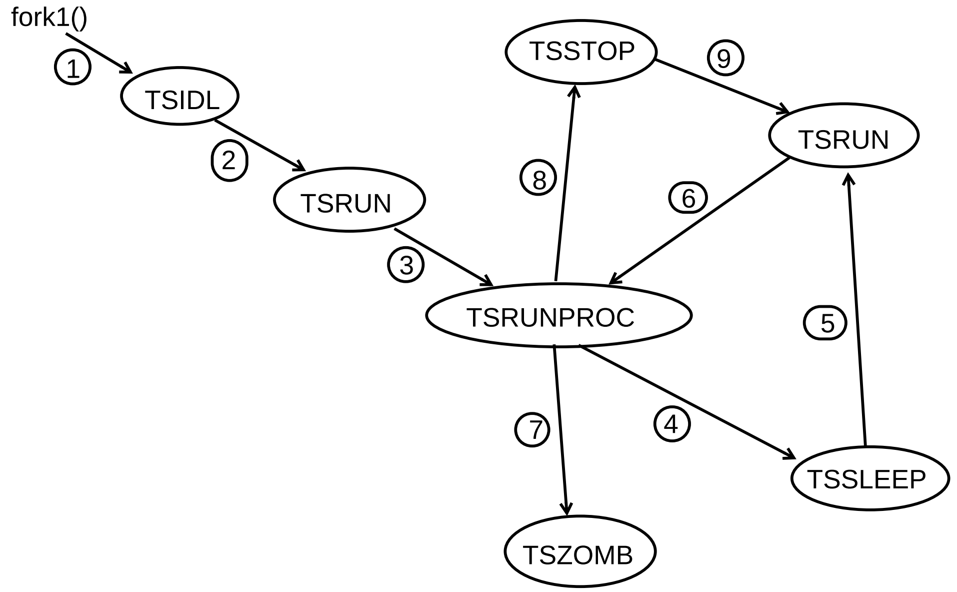
A kernel thread, like a process, operates in user and kernel modes, and through the course of its lifetime switches between the stacks maintained in each mode. Stacks for each mode accumulate information such as variables, addresses, and buffer counts accumulate and it is through these stacks that the thread executes instructions and switches modes.

Certain kinds of instructions trigger mode changes. For example, when a program invokes a system call, the system call stub code passes the system call number through a gateway page that adjusts privilege bits to switch to kernel mode. When a thread switches mode to the kernel, it executes kernel code and uses the kernel stack.

Thread’s Life Cycle

Like the process, the thread can be understood in terms of its “life cycle”, shown in the figure below. Thread state and flag codes are defined in `kthread_private.h`.

Figure 1-6 **Thread life cycle**



1. Process is created via a call to `fork()` or `vfork()`; the `fork1()` routine sets up the process's `pid` (process id) and `tid` (thread id). The process and its thread are linked to the active list. The thread is given a creation state flag of `TSIDL`.

2. `fork1()` calls `newproc()` to create the thread and process, and to set up the pointers to the parent. `newproc()` calls `procdup()` to create a duplicate copy of the parent and allocate the `uarea` for the new child process. The new child thread is flagged runnable and given a flag of `TSRUN`. Once the thread has this flag, it is placed in the run queue.

3. The kernel schedules the thread to run; its state becomes `TSRUNPROC` (running). While in this state, the thread is given the resources it requests. This continues until a clock interrupt occurs, or the thread relinquishes its time to wait for a requested resource, or the thread is preempted by another (higher priority) thread. If this occurs, the thread's context is switched out.

4. A thread is switched out if it must wait for a requested resource. This causes the thread to go into a state of `TSLEEP`. The thread sleeps until its requested resource returns and makes it eligible to run again. During the thread's `TSLEEP` state, the kernel calls

hardclock() every click tick (10ms) to charge the currently running thread with cpu usage. After 4 clock ticks (40ms), hardclock() calls setpri() to adjust the thread's user priority. The thread is given this value on the next context switch. After 10 click tics (100ms), a context switch occurs. The next thread to run will be the threadwith the highest priority in a state of TSRUN. For the remaining threads in TSRUN state, schedcpu() is called after 100 clock tics (1 second). schedcpu() adjusts all thread priorities at this time.

5. Once a thread acquires the requested resource, it calls the wakeup() routine and again changes states from TSLEEP to TSRUN. This makes the thread eligible to run again.
6. On the next context switch the thread is allowed to run, provided it is the next eligible candidate. When allowed to run, the thread state changes again to TSRUNPROC.
7. Once the thread completes its task it calls exit(). It releases all resources and transfers to the TSZOMB state. Once all resources are released, the thread and the process entries are released.
8. If the thread is being traced, it enters the TSSTOP state.
9. Once the thread is resumed, it transfers from TSSTOP to TSRUN.

Multi-Threading

When a task has two or more semi-independent subtasks, multiple threading can increase throughput, give better response time, speed operations, improve program structure, use fewer system resources, and make more efficient use of multiprocessors. With multi-threading, a process has many threads of control. Note, order of execution is still important!

The following terminology will be useful to understand multi-threading:

User threads Handled in user space and controlled using the threads APIs provided in the threads library. Also referred to as user-level or application-level threads.

Kernel threads Handled in kernel space and created by the thread functions in the threads library. Kernel threads are kernel schedulable entities visible to the operating system.

Lightweight processes (LWPs) Threads in the kernel that execute kernel code and system calls.

Bound threads Threads that are permanently bound to LWPs. A bound thread is a user thread bound directly to a kernel thread. Both a user thread and a kernel-scheduled entity are created when a bound thread is created.

Unbound threads Threads that attach and detach from among the LWP pool. An unbound thread is a user thread that can execute on top of any available LWP. Bother bound and unbound threads have their advantages and disadvantages, depending entirely on the application that uses them.

Concurrency At least two threads are in progress at the same time

Parallelism At least two threads are executing simultaneously.

Thread Models

Industrywide, threads are implemented according to four distinct models:

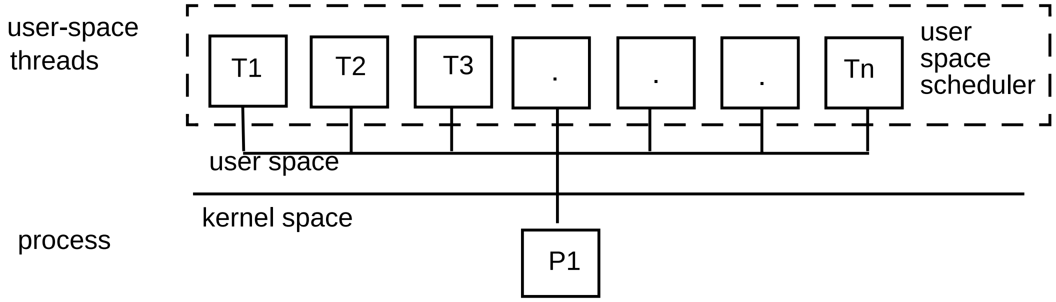
- User-space threads ("many [threads] to one [process]" (M x 1))
- Kernel-space threads ("one to one" (1 x 1))
- user-space and kernel-space threads ("many to many" (M x N))
- POSIX threads (`pthreads`), which can be used with any other model

User-Space Threads (M x 1)

User-space threads (M x 1, based on POSIX draft 4 threads) are created, terminated, synchronized, scheduled, and so forth using interfaces provided by a threads library. Creation, termination, and synchronization operations can be performed extremely fast using user-space threads.

Because user-space threads are not directly visible to the kernel (which is aware only of the overriding process containing the user-space threads), user-space threads (M x 1) require no kernel support. As shown in the following figure, all thread management is done in user space, so there is no overhead associated with crossing the kernel boundary.

Figure 1-7 User-space (M x 1) threads for an application (process)



If one thread blocks, the entire process blocks. When this happens, the benefit of threads parallelism is lost. Wrappers around various system calls can reduce some of the blocking, but at a cost to performance.

User-space threads have been on HP-UX since release 9.0 with DCE and on all 10.0 systems.

Kernel-Space Threads (1 x 1)

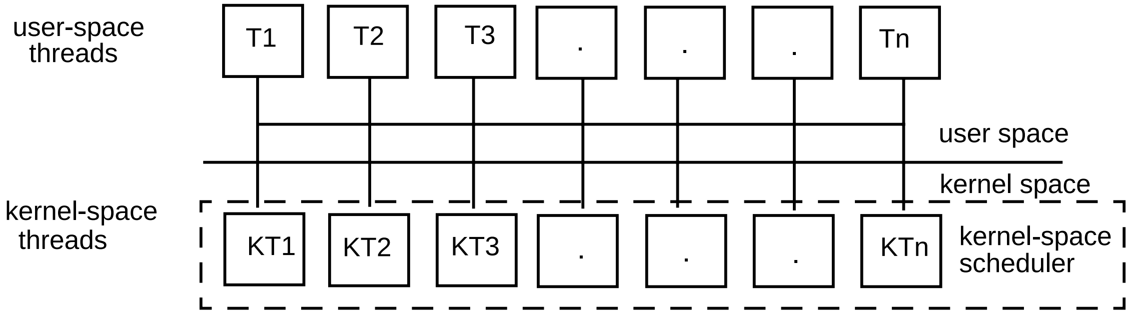
With kernel threads (1 thread to one process, or 1 x 1), each user thread has a corresponding kernel thread. Thus, there is full kernel support for threads.

Each thread is independently schedulable by the kernel, so if one thread blocks, others can still run.

Creation, termination, and synchronization can be slower with kernel threads than user threads, since the kernel must be involved in all thread management. Overhead may be greater, but more concurrency is possible using kernel threads, even with a uniprocessor system. As a result, total application performance with kernel-space threads surpasses that of user-space threads.

Note, however, that developers must be more careful when creating large amounts of threads, as each thread adds more weight to the process and more overhead to the system

Figure 1-8 **Kernel-space (1 x 1) threads for an application (process)**

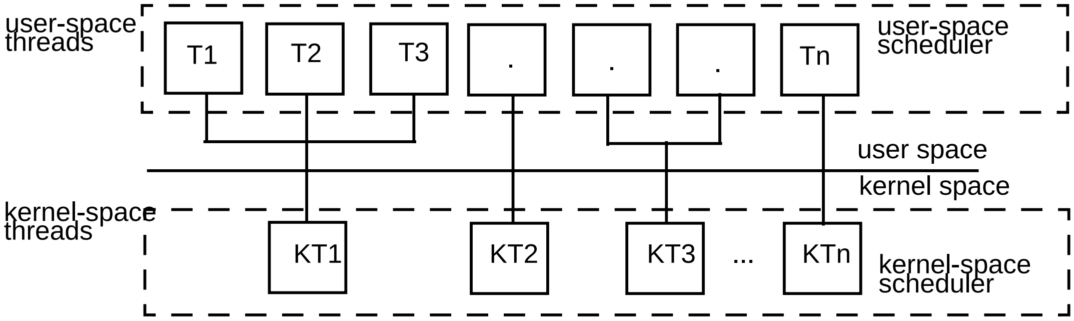


User-Space and Kernel-Space Threads (M x N)

The model of maximal flexibility in the number of threads to processes (M x N) provides the developer with both user-space and kernel-space threads.

The M x N model allows threads to be bound or unbound from kernel threads. Users can use unbound user space threads to take advantage of thread management occurring in user space. Bound threads enable use of the independent kernel scheduling provided for by kernel threads, as well as the true parallelism capabilities (both logical and physical) of an MP machine.

Figure 1-9 User-space and kernel-space (M x N) threads for an application (process)



The combination of user-space and kernel-space threads allow for extremely fast and efficient thread creation, termination, synchronization, and context switching, and thus, better performance and more system throughput than either 1 x 1 or M x 1 model. Developers need not worry about additional threads adding weight to the process or overhead to the system, as in the user-space threads (1 x 1) model. Further, blocking does not require the kernel to context switch to another process; another thread in the process will execute next.

Because the M x N model is the most powerful and flexible for programmers, it is also the most complex. Debugging can be difficult.

POSIX Threads (pthreads)

HP-UX release 10.30 implements the 1 x 1 (kernel-space threads) version of IEEE Portable Operating System Interface standard threads, (POSIX 1003.1c, based on final draft 10). Pthreads includes functions and application programming interfaces (APIs) to support multiple flows of control (threads) within a process. Using pthreads enables developers to create source-code portable applications.

For details on `pthread` functions, attributes, stack information, and so forth, consult the `pthread.h` file.

NOTE

The HP-UX DCE version of threads (M x 1, or user-space threads) complies with POSIX draft 4. However, neither 1 x 1 nor M x N model implementations will be binary or source compatible. In addition, HP also provides extensions to the POSIX.1c threads programming environment to equip the programmer with additional control and powerful features previously unavailable. These extensions, however, are not portable on different platforms. Do not rely on them for writing portable applications!

Process creation

Process 0 is created and initialized at system boot time but all other processes are created by a `fork()` or `vfork()` system call.

- The `fork()` system call causes the creation of a new process. The new (child) process is an exact copy of the calling (parent) process.
- `vfork()` differs from `fork()` only in that the child process can share code and data with the calling process (parent process). This speeds cloning activity significantly at a risk to the integrity of the parent process if `vfork()` is misused.

NOTE The use of `vfork()` for any purpose except as a prelude to an immediate `exec()` or `exit()` is not supported. Any program that relies upon the differences between `fork()` and `vfork()` is not portable across HP-UX systems.

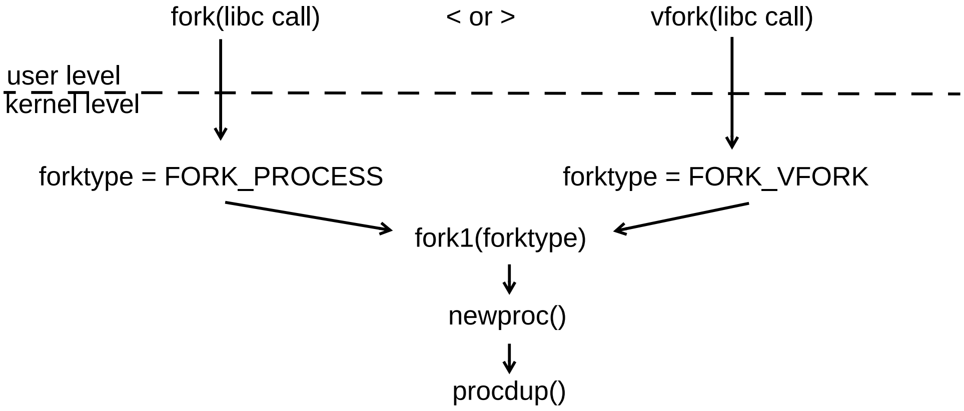
Table 1-4 Comparison of fork() and vfork()

fork()	**vfork()**
Sets context to point to parent. Child process is an exact copy of the parent process. (See `fork(2)` manpage for inherited attributes.)	Can share parent's data and code. `vfork()` returns 0 in the child's context and (later) the pid of the child in the parent's context.
Copy on access	Child borrows the parent's memory and thread of control until a call to `exec()` or `exit()`. Parent must sleep while the child is using its resources, since child shares stack and uarea
Must reserve swap	No reservation of swap

NOTE At user (application) level, processes or threads can create new processes via `fork() or vfork()`.

At kernel level, only threads can fork new processes.

Figure 1-10 Comparing fork() and vfork() at process creation



When `fork`'d, the child process inherits the following attributes from the parent process:

- Real, effective, and saved user IDs.
- Real, effective, and saved group IDs.
- List of supplementary group IDs (see `getgroups(2)`).
- Process group ID.
- File descriptors.
- Close-on-exec flags (see `exec(2)`).
- Signal handling settings (`SIG_DFL, SIG_IGN`, address).
- Signal mask (see `sigvector(2)`).
- Profiling on/off status (see `profil(2)`).
- Command name in the accounting record (see `acct(4)`).
- Nice value (see `nice(2)`).
- All attached shared memory segments (see `shmop(2)`).
- Current working directory
- Root directory (see `chroot(2)`).
- File mode creation mask (see `umask(2)`).

- File size limit (see `ulimit(2)`).
- Real-time priority (see `rtprio(2)`).

Each child file descriptor shares a common open file description with the corresponding parent file descriptor. Thus, changes to the file offset, file access mode, and file status flags of file descriptors in the parent also affect those in the child, and vice-versa.

The child process differs from the parent process in the following ways:

- The child process has a unique process ID.
- The child process has a different parent process ID (which is the process ID of the parent process).
- The set of signals pending for the child process is initialized to the empty set.
- The trace flag (see the `ptrace(2) PT_SETTRC` request is cleared in the child process.
- The `AFORK` flag in the ac_flags component of the accounting record is set in the child process.
- Process locks, text locks, and data locks are not inherited by the child (see `plock(2)`).
- All `semadj` values are cleared (see `semop(2)`).
- The child process's values for `tms_utime, tms_stime, tms_cutime, and tms_cstime` are set to zero..
- The time left until an alarm clock signal is reset to 0 (clearing any pending alarm), and all interval timers are set to 0 (disabled).

The `fork1()` Routine

Both `fork()` and `vfork()` call the `fork1()` routine to create a new process, specifying as `forktype`:

- `FORK_PROCESS` when the `fork()` system call is used
- `FORK_VFORK` when the `vfork()` system call is used

The next table itemizes the subroutines performed by `fork1()`.

Table 1-5 **`fork1(forktype)`**

Subroutine	Purpose
`getnewpid()`	Set up unique process ID. The following PIDs are reserved for the system: • 0 -- `PID_SWAPPER` • 1 -- `PID_PAGEOUT` • 2 -- `PID_INIT` • 3 -- `PID_STAT` • 4 -- `PID_UNHASH` • 5 -- `PID_NETISR` • 6 -- `PID_SOCKREGD` • 7 -- `PID_MAXSYS`
`getnewtid()`	Set up unique thread ID for the main thread of the new process. The following TIDs are reserved for the system: • 0 -- `TID_SWAPPER` • 1 -- `TID_INIT` • 2 -- `TID_PAGEOUT` • 3 -- `TID_STAT` • 4 -- `TID_UNHASH` • 5 -- `TID_NETISR` • 6 -- `TID_SOCKREGD` • 7 -- `TID_MAXSYS`
`proc_count()`	Verify that a user process does not exceed `nproc` (maximum number of `proc` table entries)

Subroutine	Purpose
allocproc()	Allocate space for the proc structure entry and clear it. Allocate memory required for the process by a call to kmalloc. Remove the allocated process table slot from the free list. Mark the entry "process creation in progress", corresponding to a p_flag definition of SIDL (process creation state).
allocthread()	Allocate space for the thread structure and add it to the active thread list.Initialize the entry to a kthread flag state of TSIDL (thread creation state)
link_thread_to_proc()	Link the child thread structure to its proc structure
thread_hash(), proc_hash()	Hash the child thread structure for its TID and the proc structure for its UID and PID.

link_thread_to_active()
link_proc_to_active()
	Link the child thread structure to the active threads list and the child process to the active process list

If fork1() is called with a forktype of FORK_VFORK, memory is allocated and initialized for a vforkinfo() structure, which is referenced through the proc structures of the parent/child vfork() pair. The vforkinfo structure holds state information about the vfork() and a copy of the parent stack. The vforkinfo() structure is used throughout process creation while the child decides to exit() or exec(). struct vforkinfo is found in proc_private.h.

fork1() switches to newproc(), giving it forktype, proc table slot, and thread with which to create the new process.

The newproc() Routine

When fork1 calls newproc(), things proceed differently depending on whether forktype is FORK_VFORK or FORK_PROCESS.

newproc(FORK_VFORK)

newproc() gets a pointer to the parent process and ascertains that no other forks or exits are occurring at the same time.

- newproc() verifies the kt_stat (thread state) is TSIDL; if not it panics.
- newproc() gets a pointer to the parent process and thread.
- When called by vfork, newproc() allocates the vforkinfo buffer, by calling vfork_buffer_init()
    - vfork_buffer_init() determines the kernel stack size, uarea size, and room for growth, allocates memory, fills in vforkbuf information, saves frame_size, pointer to parent's uarea, and pointer to last buffer.
- newproc() computes the size of the fork, then acquires the sched_lock. While holding the lock, newproc() updates process statistics.
- By calling fork_inherit, newproc()performs all direct assignments from the parent to the child proc structure.

NOTE POSIX mandates that the child process and thread inherit the scheduling policy and priority from the parent process that forked them.

- newproc() calls dbfork() subroutine to set up the child's flags to adhere to locking rules (documented in proc_private.h). The child process p_flag is set to SLOAD and the child thread kt_flag is set to TSFIRSTTHREAD.
- newproc() calls reload_ticksleft() to update the child thread (ct) and mark the parent proc pointer (pp) unswappable by calling make_unswappable(). This is to prevent to parent from being swapped while the environment is copied for the child.

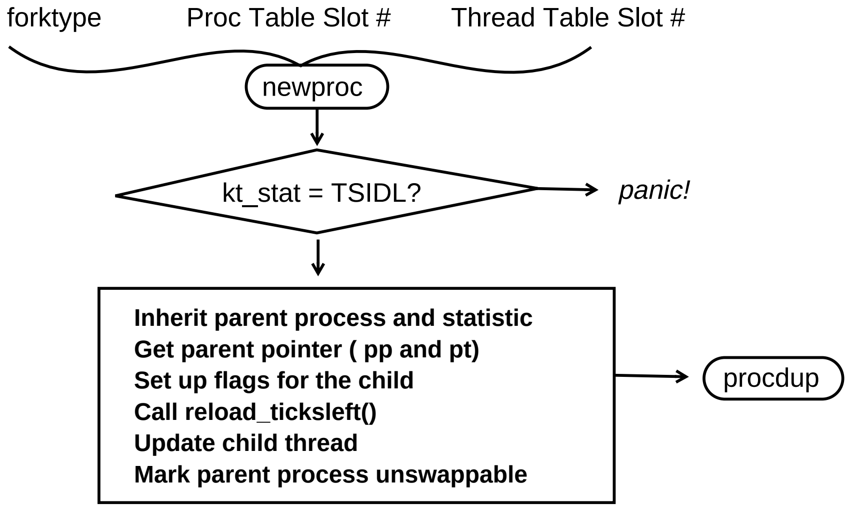
- newproc() releases the spinlock, determines whether the parent is a graphics process (if so, sets up the child save state), simulates the environment of the new process, prevents the parent from being swapped, then switches to procdup(), providing it forktype and addresses of parent and child process and thread.

newproc(FORK_PROCESS)

If newproc() is called by fork1(FORK_PROCESS), newproc() creates the new child process by calling makechild().

- newproc() calls make_unswappable() to prevent the parent from being swapped.
- newproc() switches to procdup(), passing it forktype and addresses of parent and child process and thread.

Figure 1-11 Kernel-level view of the newproc() routine


forktype
Proc Table Slot #
Thread Table Slot #
newproc
kt_stat = TSIDL?
panic!
Inherit parent process and statistic
Get parent pointer (pp and pt)
Set up flags for the child
Call reload_ticksleft()
Update child thread
Mark parent process unswappable
procdup

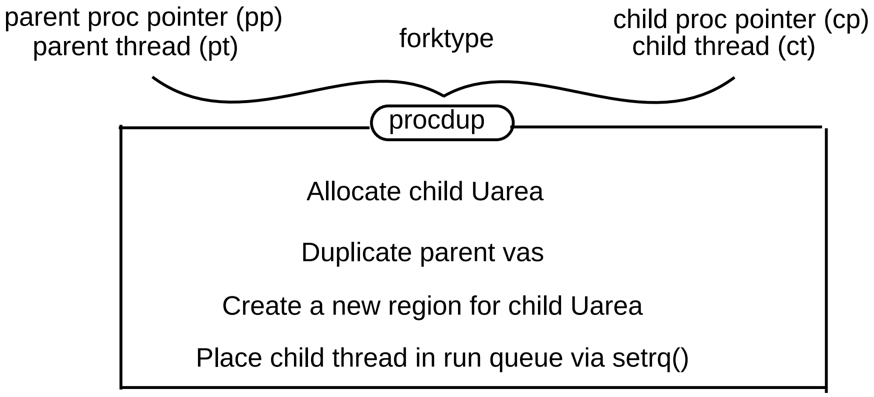
The procdup() Routine

newproc() calls procdup() when most of the child's structure has been created, passing to it the forktype, parent's proc pointer (pp), child's proc pointer (cp), parent's thread (pt), and child's thread (ct).

As it executes, procdup() does the following:

- Builds a uarea and address space for the child.
- Duplicates the parent's virtual address space.
- Creates a new region for the child's uarea.
- Attaches the region: PF_NOPAGE keeps vhand from paging the uarea.
- Marks the pregion to be owned by new child process. The address space is owned by the process.
- Places the child thread on the run queue by calling setrq(). The child thread is marked SET_RUNRUN and the thread is unlocked.

Figure 1-12 Kernel-level view of procdup() routine



vfork State information in struct vforkinfo

To prepare a vfork'd process to run, the vfork_state is maintained in struct vforkinfo. Five states are defined:

- VFORK_INIT
- VFORK_PARENT
- VFORK_CHILDRUN
- VFORK_CHILDEXIT
- VFORK_BAD

During the fork1() routine, vfork_buffer_init sets the vfork_state in vforkinfo to VFORK_INIT. When procdup() places the child thread on the run queue by calling setrq(), it also sets the vfork_state to VFORK_PARENT. The parent sleeps and is not awakened until the child exits or execs. At this point the parent and child share the same uarea and stack.

If the process was initiated with fork() rather than vfork(), the spinlock is unlocked and the process is made swappable.

The child process runs using the parent's stack until it does an exec() or exit().

Process Execution

Once a process is created with `fork()` and `vfork()`, the process calls `exec()` (found in `kern_exec.c`) to begin executing program code. For example, a user might run the command `/usr/bin/ll` from the shell and to execute the command, a call is made to `exec()`.

`exec()`, in all its forms, loads a program from an ordinary, executable file onto the current process, replacing the existing process's text with a new copy of an executable file.

An executable object file consists of a header (see `a.out(4)`), text segment, and data segment. The data segment contains an initialized portion and an uninitialized portion (`bss`). The path or file argument refers to either an executable object file or a script file of data for an interpreter. The entire user context (text, data, `bss`, heap, and user stack) is replaced. Only the arguments passed to `exec()` are passed from the old address space to the new address space. A successful call to `exec()` does not return because the new program overwrites the calling program.

The Routines of exec

The `exec()` system call consists of numerous routines and subroutines that prepare the environment to execute the command in an orderly fashion. The table that follows describes them.

Table 1-6 **Major routines and subroutines of exec()**

Routine	Purpose
`exec`	Called from user space with arguments of filename, `argv` array, and environment array. Calls `execv()`, which calls `execve()`

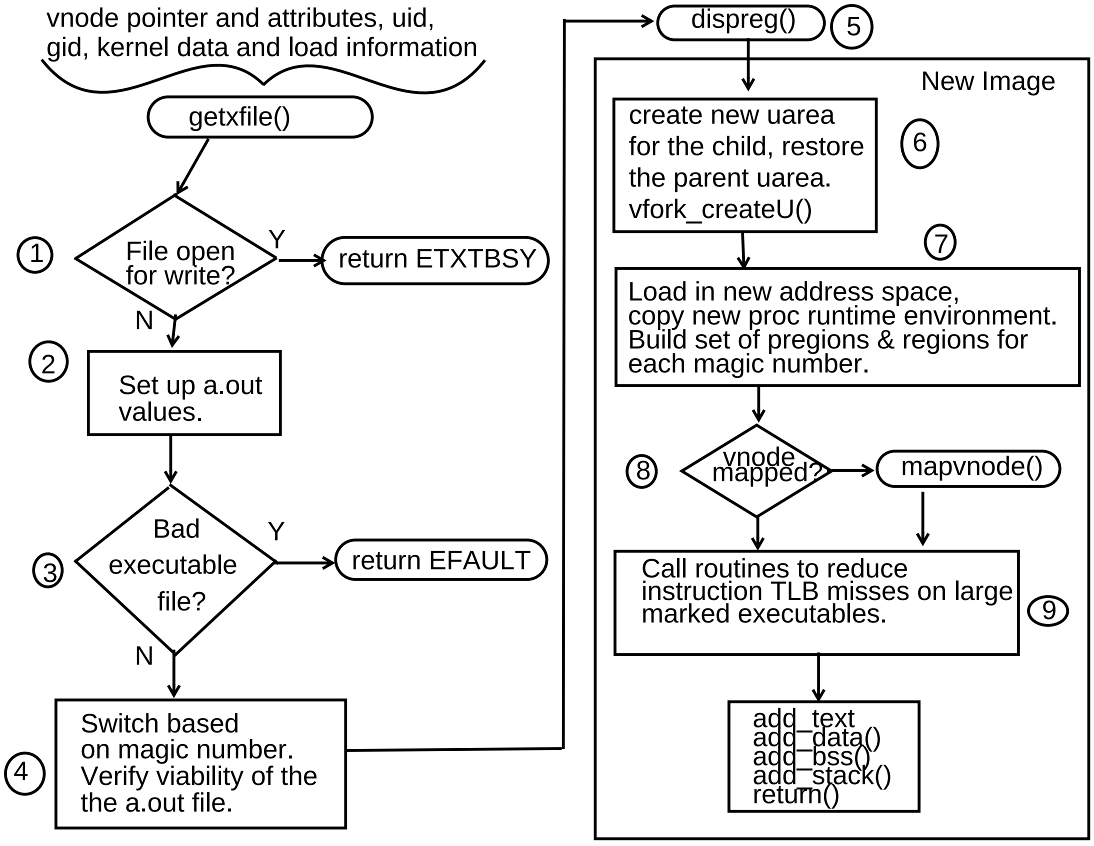
Routine	Purpose
`execve`	Determines characteristics of the executable: • Gets the complete file path name, its vnode and attributes. • Makes calls to the `vnode`-specific routines to extract information about the `uid` and `gid` of the executable. • Ascertains whether the executable is a script, and if so, gets its “interpreter” name, arguments, `vnode` pointer, object file, and relevant kernel information; then sets up the arguments to enable the shell script to run. • Sets up the structure to enable copying in from user space to kernel space. • Copies the filename, argument, and environment pointers. • Gets the new executable by calling the subroutine `getxfile()`.
`getxfile`	Sets up structures: • Sets up memory and reads in the executed file according to the executable’s magic number. • Sets up kernel stack by moving stack pointers from user stack to make room for argument, environment pointers and strings. • Copies the buffer containing the name of the executable and arguments into a per-process record (`pstat_cmd`). • Saves `argc` and `argv` values in the `save_state` structure.
`exec_cleanup`	Copies arguments onto the user stack. If cannot load `a.out`, clean up memory allocations. Release any buffers held and file `vnode`.

A Closer Look at getxfile

The `execve()` function calls `getxfile()` to map out the memory that will be used by executed file. When called, `getxfile()` is passed in:

`vp`	Pointer to a `vnode` representing the file to be executed
`argc`	A count of argments
`uid, gid`	User and group IDs
`ap`	Header file information
`vattr`	`vnode` attributes

Figure 1-13 kernel view of getxfile


vnode pointer and attributes, uid,
gid, kernel data and load information
getxfile()
1
File open for write?
Y
return ETXTBSY
N
2
Set up a.out values.
Bad executable file?
Y
3
return EFAULT
N
4
Switch based on magic number. Verify viability of the the a.out file.
dispreg()
5
New Image
create new uarea for the child, restore the parent uarea. vfork_createU()
6
7
Load in new address space, copy new proc runtime environment. Build set of pregions & regions for each magic number.
8
vnode mapped?
mapvnode()
Call routines to reduce instruction TLB misses on large marked executables.
9
add_text
add_data()
add_bss()
add_stack()
return()

`getxfile()` performs the following:

1. Verifies that no other program is currently writing to the file.
2. Sets up `a.out`-dependent values needed to load the file into memory.
3. Checks page alignment.
4. Switches with the magic number appropriate to the executable's `a.out`. Three types are defined for the case statement:

`EXEC_MAGIC`	407	Creates only a data object
`SHARE_MAGIC`	410	Creates text and data, but does not assume file alignment
`DEMAND_MAGIC`	413	Assumes both memory and file are aligned

   Calls the function `create_execmagic()` for the case of `EXEC_MAGIC` to place the entire executable (text and data) into one region.

   Checks the sizes and alignments of the `a.out`.

5. Determines whether the process will `execself()`. If `execself()`, calls `dispreg()` to dispose of old `pregions`. From this point on, the process is committed to the new image. Releases virtual memory resources of the old process and initializes the virtual memory of the new process.
6. Creates new `uarea` for the child and restore the parent's uarea. At this point, if the process was `vfork`'d, call `vfork_createU()`.
7. Having destroyed the old address space of the process, load the executable into the new address space. Determine whether executable is using static branch prediction and whether text should be locked down using superpages. Build set of `pregions` and regions for each magic number.
8. Depending on the file size and offset and given the implementation of memory-mapped files, determine how much of the file should be mapped. Call mapvnode to provide the flexibility. Call `add_text()` and `add_data()`.
9. For programs with large marked executables, execute code to reduce ITLB misses. Call `add_text()`.

10. After the switch has completed, set up the bss, by calling add_bss(), and the stack, by calling add_stack().

If getxfile is Called by a vfork'd Process

The child process runs using the parent's stack until the process does an exec() or exit(). In the case of exec(), the routine vfork_createU() is called to create a new vas and Uarea for the child (it copies the current ones into these). We then call vfork_switchU() to activate the newly created uarea and to set up the space and pid registers for the child. The state is then set to VFORK_CHILDEXIT. On an exec(), we call vfork_transfer() directly from vfork_createU() to restore the parent's stack. The parent is then awakened.

Figure 1-14 Runnable vfork'd child calls exec()

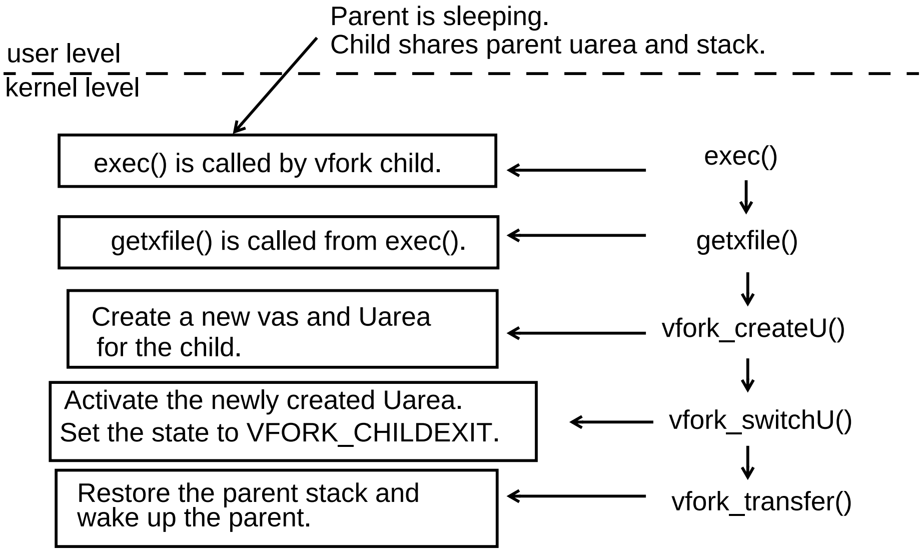

Parent is sleeping.
Child shares parent uarea and stack.
user level
kernel level
exec() is called by vfork child.
exec()
getxfile() is called from exec().
getxfile()
Create a new vas and Uarea for the child.
vfork_createU()
Activate the newly created Uarea.
Set the state to VFORK_CHILDEXIT.
vfork_switchU()
Restore the parent stack and wake up the parent.
vfork_transfer()

Table 1-7 **vfork subroutines called by getxfile**

Subroutine	Purpose
vfork_createU	Called when child does a vfork() followed by an exec(). Sets up a new vas and dups the stack/uarea from the parent (which it has been using until now). Switches the child to use the created stack/uarea.
vfork_switchU	Switches the current process to a new uarea/stack.
vfork_transfer	The code that implements vfork. When a process does a vfork, a vforkinfo struct is allocated, shared, and pointed to by the vfork'd parent & child process. The vfork_state is set to VFORK_INIT until the child is made runnable in procdup(), when the state is set to VFORK_PARENT and the parent is put to sleep. At this point the child runs in the parent's vas using the parent's uarea and stack. The vfork'd process calls vfork_transfer from within the VFORK_PARENT state. The schedlock is held to prevent any process from running during the save(). The sizes of the stack and uarea are calculated and copied into the vforkinfo u_and_stack_buf area to enable the parent to be restored when the child does an exec or exit. Then the state of the process is set to VFORK_CHILDRUN and returned. If the child exits, it changes its state to VFORK_CHILDEXIT, calls swtch() and awakens the parent, which calls resume() to restore its stack. The parent cleans up the vforkinfo structure.

vfork in a Multiprocessor Environment

In a multiprocessor environment, if vfork() is called, the child must not be picked up by another processor before the parent is fully switched out. To prevent this from occurring the TSRUNPROC bit is left on. The code that picks up a process to run (find_process_my_spu()) ignores TSRUNPROC processes. When the parent has switched out completely, it will clear the TSRUNPROC bit for the child.

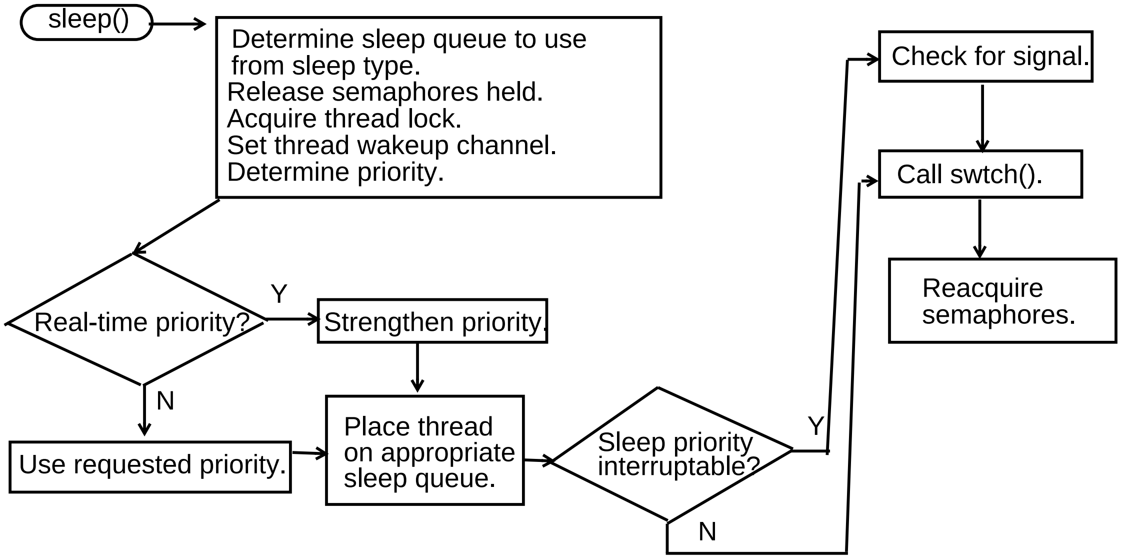
The sleep*() Routines

Unless a thread is running with real-time priority, it will exhaust its time slice and be put to sleep. sleep() causes the calling thread (not the process) to suspend execution for the required time period. A sleeping thread gives up the processor until a wakeup() occurs on the channel on which the thread is placed. *True?* During sleep() the thread enters the scheduling queue at priority (pri).

- When pri <= PZERO, a signal cannot disturb the sleep
- If pri > PZERO the signal request will be processed.
- In the case of RTPRIO scheduling, a signal can be disturbed only if SSIGABL is set. Setting SSIGABL is dependent on the value of pri.

NOTE The sleep.h header file has parameter and sleep hash queue definitions for use by the sleep routines. The ksleep.h header file has structure definitions for the channel queues to which the kernel thread is linked when asleep.

Figure 1-15 **sleep() routine**



- `sleep()` is passed the following parameters:
    - Address of the channel on which to sleep.
    - Priority at which to sleep and sleep flags.
    - Address of thread that called `sleep().`
- The priority of the sleeping thread is determined.
    - If the thread is scheduled real-time, `sleep()` makes its priority the stronger of the requested value and `kt_pri`.
    - Otherwise, `sleep()` uses the requested priority.
- The thread is placed on the appropriate sleep queue and the sleep-queue lock is unlocked.
    - If sleeping at an interruptable priority, the thread is marked `SSIGABL` and handle any signals received.
    - If sleeping at an uninterruptable priority, the thread is marked `!TSSIGABL` and will not handle any signals.
- The thread's voluntary context switches are increased and `swtch()` is called to block the thread.

- Once time passes and the thread awakens, it checks to determine if a signal was received, and if so, handles it.
- Semaphores previously set aside are now called again.

wakeup()

The wakeup() routine is the counterpart to the sleep() routine. If a thread is put to sleep with a call to sleep(), it must be awakened by calling wakeup().

When wakeup() is called, all threads sleeping on the wakeup channel are awakened. The actual work of awakening a thread is accomplished by the real_wakeup() routine, called by wakeup() with the type set to ST_WAKEUP_ALL. When real_wakeup() is passed the channel being aroused, it takes the following actions:

- Determines appropriate sleep queue (slpque) data structure, based on the type of wakeup passed in.
- Acquires the sleep queue lock if needed in the multiprocessing (MP) case; goes to spl6 in the uniprocessing (UP) case.
- Acquires the thread lock for all threads on the appropriate sleep queue.
    - If the kt_wchan matches the argument chan, removes them from the sleep queue and updates the sleep tail array, if needed.
    - Clears kt_wchan and its sleeping time.
    - If threads were TSSLEEP and not for a beta semaphore, real_wakeup() assumes they were not on a run queue and calls force_run() to force the thread into a TSRUN state.
    - Otherwise, if threads were swapped out (TSRUN && !SLOAD), real_wakeup() takes steps to get them swapped in.
    - If the thread is on the ICS, attributes this time to the thread being awakened. Starts a new timing interval attributing the previous one to the thread being awakened.
- Restores the spl level, in the UP case; releases the sleep queue lock as needed in the MP case.

force_run()

The force_run subroutine marks a thread TSRUN, asserts that the thread is in memory (SLOAD), and puts the thread on a run queue with setrq(). If its priority is stronger than the one running, force a context switch. Set the processor's wakeup flag and notify the thread's processor (kt_spu) with the mpsched_set() routine. Otherwise, force_run() improves the the swapper's priority if needed, sets wantin, and wakes up the swapper.

Process Termination

When a process finishes executing, HP-UX terminates it using the `exit` system call.

Circumstances might require a process to synchronize its execution with a child process. This is done with the wait system call, which has several related routines.

During the `exit` system call, a process enters the zombie state and must dispose of child processes. Releasing process and thread structures no longer needed by the exiting process or thread is handled by three routines -- `freeproc()`, `freethread()`, and `kissofdeath()`.

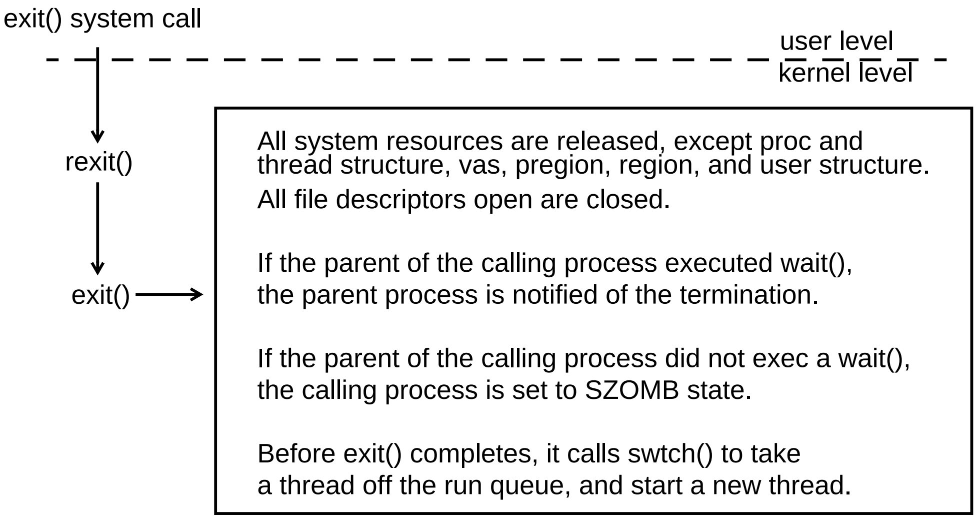
This section will describe each process-termination routine in turn.

The `exit` System Call

`exit()` may be called by a process upon completion, or the kernel may have made the call on behalf of the process due to a problem.

If the parent process of the calling process is executing a `wait()`, `wait3()`, or `waitpid()`, it is notified of the calling process's termination. If the parent of the calling process is not executing a `wait()`, `wait3()`, or `waitpid()`, and does not have `SIGCLD` (death of a child) signal set to `SIG_IGN` (ignore signal), the calling process is transformed into a zombie process. The parent process ID is set to 1 for all of the calling process's existing child processes and zombie processes. This means the process 1 (`init`) inherits each of the child processes.

Figure 1-16 Summary of the exit system call



All system resources are released, except proc and thread structure, vas, pregion, region, and user structure. All file descriptors open are closed.

If the parent of the calling process executed wait(), the parent process is notified of the termination.

If the parent of the calling process did not exec a wait(), the calling process is set to SZOMB state.

Before exit() completes, it calls swtch() to take a thread off the run queue, and start a new thread.

`exit()` passes status to the system and terminates the calling process in the following manner:

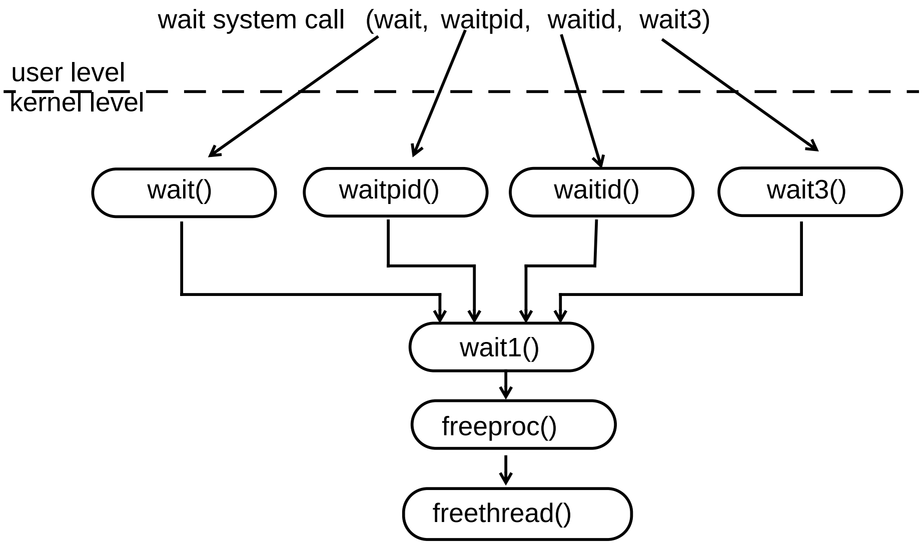
- Clear the process' `STRC` (process being traced) flag by calling `STRC_EXIT(p,kt)`.
- Issue a process-wide directive to reduce an exiting multi-threaded process to a single thread of execution. Clear the multi-threaded flag for non-`vfork` processes.
- Set the process `p_flag` to `SWEXIT` to indicate it is exiting.
- Set process to ignore signal information.
- Make sure the "no swap" counter is balanced.
- Determine whether the exiting process is a controlling process. If it has a controlling `tty`, send `SIGHUP` and free `tty` for another session.
- Release memory-mapped semaphores.
- Disarm all timers and clear any related pending `siginfos`.
- Do exit processing for graphics and `iomap` driver, if needed.
- If the process was created via `vfork()` release the virtual memory and return resources to the parent.

- Cancel all the process's pending asynchronous I/O requests and sleep until all are completed.
- Free up space taken by the table of pointers to file-descriptor chunks.
- Release MQ resources, NFS lock manager resources, audit-control structures, and process-wide and thread copies of credentials.
- Do exit processing for semaphores.
- Destroy any adopted processes.
- Search all processes for zombies whose p_dptr field is not set; send SIGCLD to the parent and wake it up, if parent has called wait().
- Signal the process group that exit is causing it to become an orphan with stopped processes. Deal with orphaned process group and any children.
- Notify init that zombies are waiting.
- Unlink from active list of processes.
- Unlink current thread from active list of threads and set the thread state.
- Process enters SZOMB state, and thread enters TSZOMB state.
- Choose which process to send SIGCLD. If exiting process is a daemon, make it a child of init.
- Awaken init to start the cleanup process.
- If vfork() created the process, reset vfork_state to VFORK_CHILDEXIT, to allow resume() to restore the parent.
- Verify that the thread state is set to TSSLEEP.
- Retain the "thread lock" across the context switch path. Both signal path and init process must wait until the last thread is non-TSRUNPROC.
- Call swtch() to release the process lock.

wait System Call

From the user perspective, the wait system call is actually four different interfaces used to synchronize a child to parent process. All four versions call the wait1() routine, which does the actual work./

Figure 1-17 **Process termination -- `wait()`**



The following table summarizes the basic differences among the `wait()` functions. For further information, consult `kern_exit.c`, `wait(2)` manpage, and `wait.h` header file.

Table 1-8 User interfaces to the wait system call

Interface	Purpose
wait()	Calls wait1() to determine if any zombie process are present; if not, wait1() sleeps. wait() determines which process is of interest based on the pid argument and passes it on to wait1(): -1, all processes >0, processes with process ID == pid 0, processes in same process group as caller <-1, processes with process group ID == -pid Once wait1() returns, wait() passes the exit status of the terminated child back to the calling process.
waitpid()	POSIX version of the wait() system call. It allows the caller to also wait on processes with the same group ID and translate information for wait1().
waitid()	Suspends the calling process until one of its children changes state. It records the current state of a child in infop (passes signal information between threads). The id_type, which contains the set of processes that should receive the signal, can be one of the following: P_PID, process id P_PGID, process group id P_ALL all processes
wait3()	Almost identical to wait(); however, wait3() can be passed a WNOHANG flag through the argument options. If the child has not exited, the WNOHANG flag will not block if no one is waiting.

wait1() subroutine

wait1() searches for a terminated (zombie) child, to gather its status information and remove it. wait1() also looks for stopped (traced) and continued children, and passes back status from them to their parents.

wait1() determines which process is interesting based on the pid argument passed. The following are some of the steps performed by wait1():

- Hold the pm_sema to ensure that a process already checked does not become a zombie while other processes are being checked, thus resulting in a lost wakeup.
- Find and count the step-children who do not belong to the parent process being waited on. If zombies, undebug them and signal their parents. If stopped, signal the step-parent.
- Search all processes. This is mainly due to the fact SZOMB processes are no longer on the active process list.
- While under the protection of an MP_PROCESS_LOCK, make sure that p_zombies_exist = 0.
- If an SZOMB process is found, the entry is removed from any proc hash table (proc_unhash()) and the thread is deallocated (thread_deallocate()).
- Call freeproc() to release the process structures SZOMB.

freeproc(), freethread(), and kissofdeath() Routines

The freeproc(), freethread(), and kissofdeath() routines complete the clean-up effort on behalf of wait1() for a terminating process

Figure 1-18 Process termination

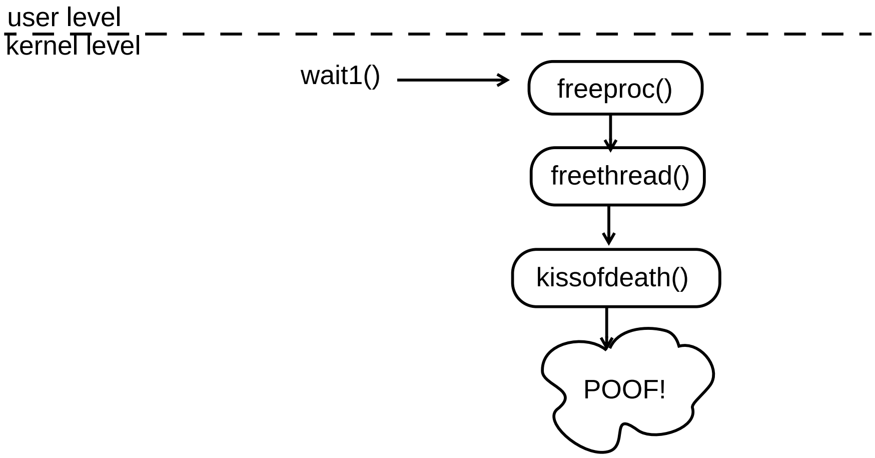


Table 1-9 **Final process termination routines**

Routine	Purpose
`freeproc()`	Called from `wait1()` to release process structures that zombie processes are no longer using. `freeproc()` performs the following steps: • Under the protection of an `MP_PROCESS_LOCK`, clear `p_pl_flags.all`, set `p_stat` to `SUNUSED`, and verify that `p_sigwaiters` is set to `NULL`. • Decrement the number of active `proc` table entries. • Add the child's `rusage` (sum of stats of the reaped child) structure to the parent.. • Release the `rusage` data sturcture, process timers data, msginfo, and any queued.signals pending against the process. • Release the virtual address space. • Return the `proc` entry to the free list
`freethread()`	Called from `wait1()` to release the thread structure. `freethread()` performs the following steps: • Under the protection of an `MP_PROCESS_LOCK`, clear `kt_cntxt_flags` and set `kt_stat` to `TSUNUSED`. • Decrement the number of active thread table entries. • Clear the thread fields. If a thread was not cached, dismantle its uarea by calling `kissofdeath()`. • Release the thread entry by calling `link_thread_to_free-list()`.
`kissofdeath()`	Called from `freethread()` and `freeproc()` to clear the kernel stack and `uarea` of a named process.

Basic Threads Management

NOTE

Detailed coverage of threads management is beyond the scope of this white paper. For detailed information, consult *ThreadTime: The Multithreaded Programming Guide*, by Scott Norton and Mark DiPasquale, published by Prentice Hall under their Hewlett-Packard Professional Books imprint.

The life cycle of a thread is analogous to that of a process. When you `fork()` a process, an initial thread is created. To take advantage of the kernel's ability to run flows of execution either concurrently or (on multiprocessor systems) in parallel, you will want to create additional threads. The following chart pairs the process-oriented function with its threads-oriented API function. In all cases, `pthread` functions are prefaced "`pthread_`"; in several cases, threads routines have no process equivalent.

For complete specification, consult the manpages for `pthread` API functions in section three of the online reference.

Table 1-10 **Comparison of thread API and process functions**

Thread API function	Process function
`pthread_create()`	`fork(), exec()`
`pthread_detach()`	<none>
`pthread_join()`	`wait()`
`pthread_exit()`	`exit()`
`pthread_self()`	`getpid()`
`pthread_equal()`	`pid1 == pid2`
`pthread_kill()`	`kill()`
`pthread_sigmask()`	`sigprocmask()`
`pthread_suspend()`	<none>
`pthread_resume()`	<none>

Thread API function	Process function

`pthread_setschedparam()`	`sched_setscheduler()`
	`sched_setparam()`

`pthread_getschedparam()`	`sched_getscheduler()`
	`sched_getparam()`

`sched_yield()`	`sched_yield()`

Pthread APIs consist of the following:

- 33 thread management functions
- 46 thread synchronization functions
- 15 thread scheduling functions

The following header files have definitions pertinent to threads:

- `pthread.h`
- `sched.h`
- `signal.h`

Pthread API functions operate by the following rules:

- There is no parent/child relationship between threads.
- Any thread can make an independent system call or library call.
- `pthread_*()` functions return 0 on success.
- `pthread_*()` functions return an error number on failure. `errno` is not set.
- You must use `-D_REENTRANT` when compiling a multi-threaded program:
  ```
  .% cc -D_REENTRANT abc.c -lpthread
  ```

Thread Creation Overview

Thread creation resembles process creation. The new thread starts execution at start_routine with arg as its sole parameter. The new thread’s ID is returned to the creator of the thread.

A thread has attributes that can be initialized prior to its creation. These include stack size, scheduling policy, and priority. For the newly created thread to have non-default attributes, you must pass a threads attribute object to `pthread_create()`. You can use the same threads attribute object to create multiple threads.

NOTE After the thread is created, it is recommended that you destroy the attribute object to save memory.

To create a new thread, the user or threads library must allocate the user stack. The kernel does not allocate stacks for threads, other than the initial thread.

Thread Termination Overview

After a thread executes, it has two options:

- Return from its `start_routine`. An implicit call to `pthread_exit()` is made with the function's return value. (Note, the initial thread should not do this. An implicit call to `exit()` is made by the initial thread on return.)
- Call `pthread_exit()` to explicitly self-terminate.

HP-UX Threads Extensions

In addition to the POSIX threads functions, HP-UX provides the extensions shown in the following table. These non-standardized interfaces are subject to change; efforts are being made to standardize them through X/Open.

Table 1-11 **Additional HP threads routines**

HP threads Extension	Purpose
`pthread_suspend()`	Suspends a thread and blocks until the target thread has been suspended. Each time a thread is suspended, its suspension count is incremented.
`pthread_continue`	Resumes execution of a suspended thread.
`pthread_resume_np()`	An HP-only function that provides control over how a thread is resumed by a flags field.
`pthread_num_processors_np()`	Returns number of processors installed on the system.
`pthread_processor_bind_np()`	Binds thread to processor.
`pthread_processor_id_np()`	Identifies a specific processor on the system.

Thread Synchronization

POSIX provides four sets of synchronization primitives from which all other synchronization types can be built:

- mutual exclusion locks (`mutex`)
- condition variables
- semaphores
- read/write locks

All synchronization is “advisory,” meaning that the programmer has the ultimate responsibility for making it work.

Synchronization objects can be process-local (used to synchronize threads within a process) or system visible (used by threads within different processes, such as by using shared memory or memory mapped files).

Synchronization operations perform as follows:

- The `init` function initializes, but does not allocate, except “internal” resources.
- The `destroy` function destroys all but the “internal” resources.
- The `lock` function acquires an object, or if the object is unavailable, waits.
- The try function returns EBUSY if the resource is being used or 0 if the resource is acquired.
- The timed function awaits a synchronization signal for an absolute time or returns ETIMEDOUT if absolute time elapses.
- The `unlock` function releases an acquired resource.

The simplest mechanism for synchronizing threads is to use `pthread_join()`, which waits fora thread to terminate. However, this function is insufficient for multithreaded cases in which threads must synchronize access to shared resources and data structures.

`mutex` Locks

Mutual exclusion (`mutex`) locks allow threads to synchronize access to process resources and shared objects, such as global data. Threads wishing to access an object locked by a mutex will block until the thread holding the object releases it.

`mutex locks` have attributes objects that can be set, based on the following characteristics:

`pshared` Indicates if the `mutex` is shared by multiple processes or is local to the calling process. Valid values are PTHREAD_PROCESS_PRIVATE (default) and PTHREAD_PROCESS_SHARED.

how Tells how a locking thread should block if it cannot acquire the `mutex`. Valid values govern whether or not the `pthread` should block and/or spin in a loop while attempting to acquire the `mutex`. Default behavior, governed by PTHREAD_LIMITED_SPIN_NP, asserts that the thread spin in a loop attempting to acquire the mutex; if not acquired after some determined number of iterations, block until the mutex can be acquired.

kind — Type of `mutex`. By default (`PTHREAD_MUTEX_FAST_NP`), the `mutex` is locked and unlocked in the fastest possible manner and no owner is maintained. Note, however, this can result in deadlock. Other valid values handle the `mutex` as recursive or nonrecursive and set rules for ownership and relocking.

Lock Order

When more than one synchronization variable is required by the same thread at the same time, lock order or hierarchy is vital to prevent deadlock. It is the programmer’s responsibility to define the order of lock acquisition.

Condition Variables

Using a synchronization type called a condition variable, a thread can wait until or indicate that a predicate becomes true. A condition variable requires a mutex to protect the data associated with the predicate.

A condition wait has two forms:

- absolute wait, until the condition occurs
- timed wait, until the condition occurs or the absolute wait time has elapsed.

The condition wait operation releases the `mutex`, blocks waiting for the condition to signaled, at which time it reacquires the `mutex`.

A condition signal operation has two forms:

- Signal a single waiter to wake up
- Broadcast to all waiter to wake up

NOTE

Be cautious about doing a broadcast wake-up, as all threads awakened must reacquire the associated mutex. This can degrade performance.

Condition variables have only one attribute -- pshared. This indicates whether the condition variable is local to the calling process (the default, PTHREAD_PROCESS_PRIVATE) or shared by multiple processes. If shared, the caller must allocate the condition variable in shared memory.

Semaphores

POSIX.1b named and unnamed semaphores have been “tuned” specifically for threads.

The semaphore is initialized to a certain value and decremented. Threads may wait to acquire a semaphore.

- If the current value of the semaphore is grater than 0, it is decremented and the wait call returnes.
- If the value is 0 or less, the thread blocks until the semaphore is available.

NOTE The POSIX.1b semaphores (both named and unnamed) were standardized before the POSIX threads standard. Consequently, they return errors in traditional UNIX fashion (0 == success, -1 with errno == failure). These semantic apply to all types of semaphores supported by HP-UX.

Read/Write Locks

These locks, a variant of the mutex lock, are useful when you have protected data that is read often but only occasionally written. Performance is slower than for a mutex.

Read/write locks allow for any number of concurrent readers and no writers or a single write and no readers. Once a writer has access, readers are blocked from access until the writer releases the lock. If both readers and writers are waiting for a lock, the released lock is given to a writer.

Read/write locks have two initializable attributes:

pshared Indicates if the read/write lock is shared by multiple processes or is local to the calling process. Valid values are PTHREAD_PROCESS_PRIVATE (default) and PTHREAD_PROCESS_SHARED.

how	Tells how a locking thread should block if it cannot acquire the read/write lock. Valid values govern whether or not the pthread should block and/or spin in a loop while attempting to acquire the `mutex`. Default behavior, governed by `PTHREAD_LIMITED_SPIN_NP`, asserts that the thread spin in a loop attempting to acquire the read/write lock; if not acquired after some determined number of iterations, block until the read/write lock can be acquired.

Signal Handling

There are two types of signals:

- Synchronous signals, which are generated as a result of some action taken by a thread at a given instant, such as an illegal instruction or dividing by zero.

  Synchronous signals are sent directly to the thread that caused the signal to be generated.

- Asynchronous signals, which are generated due to an external event, such as `kill()` or timer expirations.

  Asynchronous signals are sent to the process. A single thread within the process that does not have the signal blocked will handle the signal.

Each thread has a signal mask used to block signals from being delivered to the thread. To examine or change the current thread's signal mask, use `pthread_sigmask()`, a function that behaves just like sigprocmask() in the process model. Do not use `sigprocmask()` to change the signal mask of a thread.

Each process contains a signal vector that describes what to do for each signal (for example, ignore, default, or execute a handler). This signal vector is shared by all threads in the process. There may be only one signal handler for any given signal. This handler is used by all threads in the process.

Each signal sent to a process is delivered once and only once to one thread within the process. Signals cannot be “broadcast” to all threads in a process.

The sigwait() function

A POSIX function, sigwait(), allows a thread to wait for a signal to be delivered in a multi-threaded application. This is easier than installing signal handlers to handle the signal when it arrives and dealing with interrupted system calls.

To wait for a signal, use

```
int sigwait(sigset_t *set, int *signal);
```

set is the set of signals being waited for, which must be blocked before calling sigwait. When a signal in set is delivered, this function returns and the signal being delivered is returned in signal.

Thread Cancellation

Threads may cancel or terminate other threads within their process.

Threads targetted for cancellation may hold cancellation requests pending, similar to how signals are blocked and held pending.

A thread's cancellation "state" determines whether cancellation is enabled or disabled (the latter blocks all requests). Valid states are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

A thread's cancellation type determines when a cancellation request is acted upon (that is, when the thread is terminated). A value of PTHREAD_CANCEL_DEFERRED (default) holds cancellation requests pending until the thread enters a function that is a cancellation point. If set to PTHREAD_CANCEL_ASYNCHRONOUS, the thread can be cancelled at any moment.

NOTE When a thread is cancelled, any mutexes, attribute objects, or other resources it is consuming are not released. This can cause application deadlock later! To remedy this, a thread may install cancellation cleanup handlers to release resources in the event it is cancelled.

Thread cancellation cleanup handlers resemble signal handlers. However, a thread may have multiple handlers installed As a thread leaves a non-cancel-safe section, the cancellation cleanup handlers are removed. Any installed cancellation cleanup handlers are executed when

- The thread is cancelled.
- The thread self-terminates (with pthread_exit()).

- The handler is removed (if you specify execute).

A function safe from cancellation is one that does not contain a cancellation point nor call a function that is a cancellation point.

NOTE

Library routines must assume the application to which it is linked uses thread cancellation and protect itself.

Use the following thread cancellation cleanup handlers in your code:

`pthread_cleanup_push()` Use for a thread before it enters a section of code that can be canceled.

`pthread_cleanup_pop()` Remove the handler when the thread is finished executing the code that can be cancelled.

`pthread_cancel()` Cancel a thread or request that it terminate itself. This function does not wait for the thread to terminate.

`pthread_testcancel()` Use to test for a cancellation request and act upon it before performing time-consuming “critical” action. A thread can create a cancellation point. This function returns if no cancellation requests are pending; otherwise it does not return and the thread terminates.

`pthread_setcancelstate()` Use to change a thread’s state of cancellation.

`pthread_setcanceltype()` Use to change a thread’s type of cancellation. These latter two functions can be very handy when entering a section of code that cannot tolerate being interrupted or terminated.

Process Management Structures

The process management system contains the kernel's scheduling subsystem and interprocess communication (IPC) subsystem.

The process management system interacts with the memory management system to make use of virtual memory space. The process control system interacts with the file system when reading files into memory before executing them.

Processes communicate with other processes via shared memory or system calls. Communication between processes (IPC) includes asynchronous signaling of events and synchronous transmission of messages between processes. System calls are requests by a process for some service from the kernel, such as I/O, process coordination, system status, and data exchange.

The effort of coordinating the aspects of a process in and out of execution is handled by a complex of process management structures in the kernel. Every process has an entry in a kernel process table and a `uarea` structure, which contains private data such as control and status information. The context of a process is defined by all the unique elements identifying it -- the contents of its user and kernel stacks, values of its registers, data structures, and variables -- and is tracked in the process management structures.

Process management code is divided into external interface and internal implementation parts. The `proc_iface.h` defines the interface, contains the utility and access functions, external interface types, utility macros. The `proc_private.h` defines the implementation, contains internal functions, types, and macros.

Kernel threads code is similarly organized into kthread_iface.h and kthread_private.h.

The next figure shows process management structures. In the table that follows, the structures are identified and summarized. In the sections that follow, we will examine the most important characteristics of each structure

Figure 1-19 Process structure, virtual layout overview

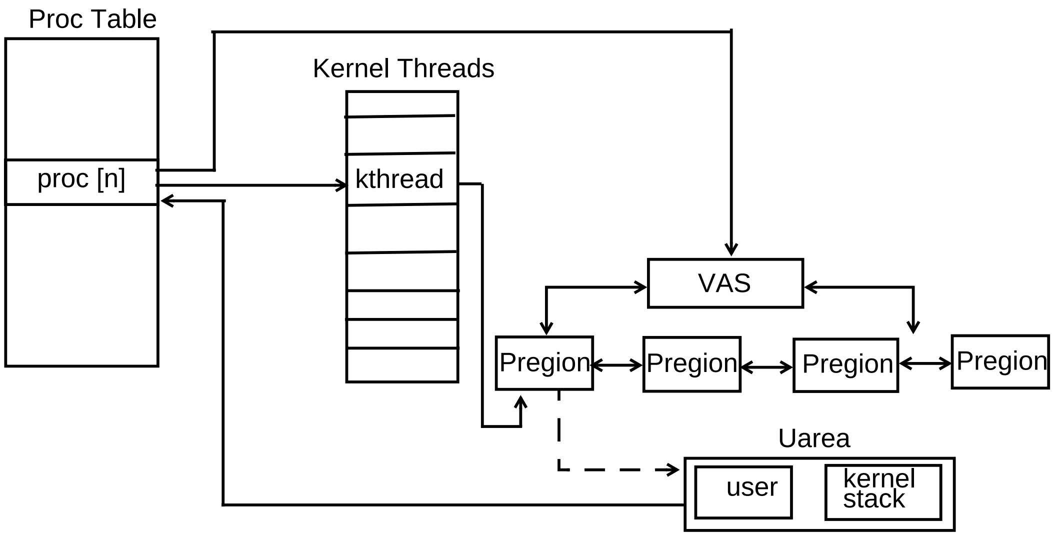


Table 1-12 Principal structures of process management

Structure	Purpose
`proc` table	Allocated at boot time; remains resident in memory (non-swappable). For every process contains an entry of the process's status, signal, and size information, as well as per-process data that is shared by the kernel thread
`kthread` structure	One of two structures representing the kernel thread (the other is the user structure). Contains the scheduling, priority, state, CPU usage information of a kernel thread. Remains resident in memory.

Structure	Purpose
vas	The vas structure contains all the information about a process's virtual space. It is dynamically allocated as needed and is memory resident.
pregion	Contains process and thread information about use of virtual address space for text, data, stack, and shared memory, including page count, protections, and starting addresses of each.
uarea	User structure contains the per-thread data that is swappable.

proc Table

The proc table is comprised of identifying and functional information about every individual process. Each active process has a proc table entry, which includes information on process identification, process threads, process state, process priority and process signal handling. The table resides in memory and may not be swapped, as it must be accessable by the kernel at all times.

Definitions for the proc table are found in the proc_private.h header file.

Figure 1-20 The proc Table

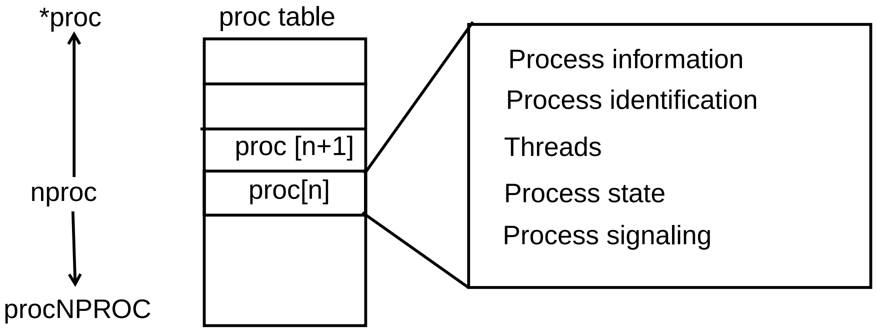
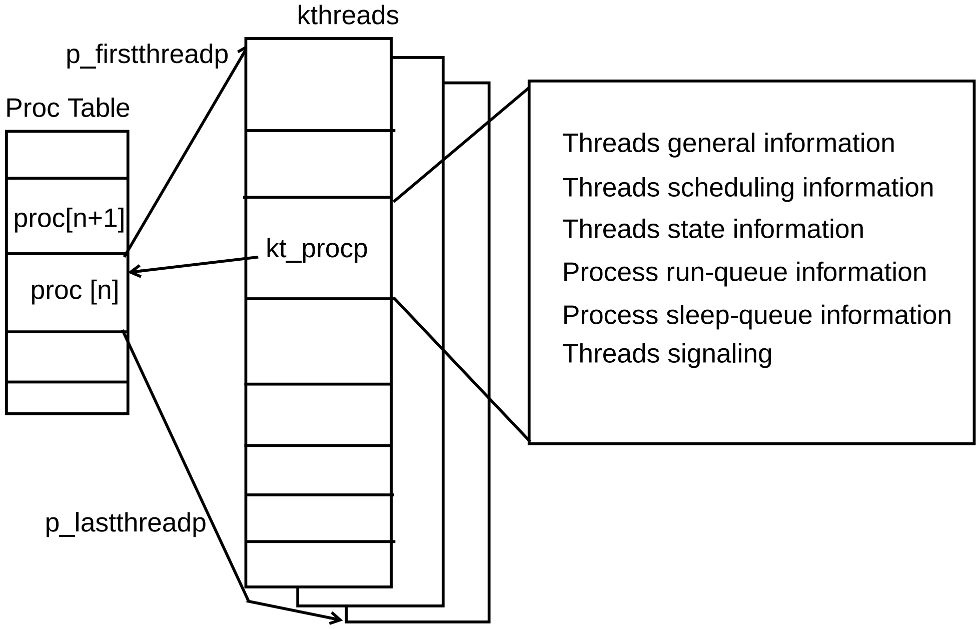


Table 1-13 Principal fields in the proc structure

Type of Field	Name and Purpose
Process identification	Process ID (p_pid) Parent process ID (p_ppid) Read user ID (p_uid) used to direct tty signals process group ID (p_pgrp) Pointer to the pgroup structure (*p_pgrp_p) Maximum number of open files allowed (p_max) Pointer to the region containing the uarea (p_upreg)
threads	Values for first and subsequent threads (p_created_threads) Pointer to first and last thread in the process (p_firstthreadp, p_lastthreadp) Number of live threads in the process, excluding zombies (p_livethreads) List of cached threads (*p_cached_threads)
process state	current process state (p_stat) priority (p_pri) per-process flags (p_flag)
process signaling	signals pending on the process (p_sig) active list of pending signals (*p_ksiactive) signals being ignored (p_sigignore) signals being caught by user (p_sigcatch) Number of signals recognized by process (p_nsig)
Locking information	thread lock for all threads (*thread_lock) per-process lock(*p_lock)

Kernel Thread Structure

Figure 1-21 Kernel threads


kthreads
p_firstthreadp
Proc Table
proc[n+1]
proc [n]
kt_procp
p_lastthreadp
Threads general information
Threads scheduling information
Threads state information
Process run-queue information
Process sleep-queue information
Threads signaling

Each process has an entry in the `proc` table; this information is shared by all kernel threads within the process.

One kernel thread structure (`kthread`) is allocated per active thread. The `kthread` structure is not swappable. It contains all thread-specific data needed while the thread is swapped out, including process ID, pointer to the process address space, file descriptors, current directory, UID, and GID. Other per-thread data (in `user.h`) is swapped with the thread.

Information shared by all threads within a process is stored in the proc structure, rather than the kthread structure. The `kthread` structure contains a pointer to its associated `proc` structure. (In a multi-threads environment the `kthread` would point to other threads that make up the process and controlled by a threads listing maintained in the `proc` table.)

In a threads-based kernel, the run and sleep queues consist of kthreads instead of processes. Each kthread contains forward and backward pointers for traversing these queues. All schedule-related attributes, such as priority and states, are kept at the threads level.

Definitions for the kernel threads structure are found in the kthread_private.h header file include general information, scheduling information, CPU affinity information, state and flag information, and signal information.

Table 1-14 Principal entries in kernel thread structure

Entry in struct kthread	Purpose
*kt_link, *kt_rlink	pointers to forward run/sleep queue link and backward run queue link
*kt_procp	Pointer to proc structure
kt_fandx, kt_pandx	Free active and kthread structure indices
kt_nextp, kt_prevp	Other threads in the same process
kt_flag, kt_flag2	Per-thread flags
kt_cntxt_flags	thread context flags
kt_fractioncpu	fraction of cpu during recent p_deactime
kt_wchan	Event thread is sleeping on
*kt_upreg	pointer to the pregion containing the uarea
kt_deactime	seconds since last deact or react
kt_sleeptime	seconds since last sleep or wakeup
kt_usrpri	User priority (based on kt_cpu and p_nice)
kt_pri	priority (lower numbers are stronger)
kt_cpu	decaying cpu usage for scheduling
kt_stat	Current thead state
kt_cursig	number of current pending signal, if any

Entry in struct kthread	Purpose
kt_spu	SPU number to which thread is assigned
kt_spu_wanted	preference to desired SPU
kt_spu_group	SPU group to which thread is associated
kt_spu_mandatory; kt_sleep_type	Assignment as to whether SPU is mandatory or advisory; directive to wake up all or one SPU
kt_sync_flag	Reader synchronization flags
kt_interruptible	Is the thread interruptible?
kt_wake_suspend	Is a resource waiting for the thread to suspend?
kt_active	Is the thread alive?
kt_halted	Is the thread halted cleanly?
kt_tid	unique thread ID
kt_user_suspcnt, kt_user_stopcnt	user-initiated suspend and job-control stop counts
kt_suspendcnt	Suspend count
*kt_krusagep	Pointer to kernel resource usages

kt_usertime;
kt_systemtime;
kt_interrupttime
	Machine cycles spent in user-mode, system mode and handling interrupts.

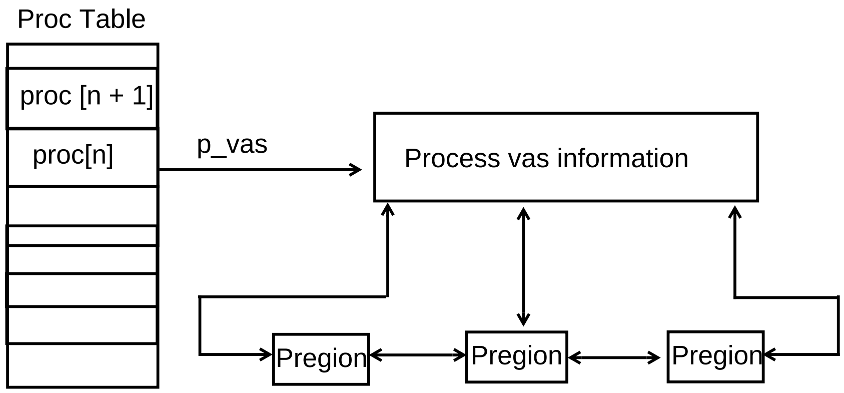
kt_sig	signals pending to the thread
kt_sigmask	Current signal mask
kt_schedpolicy	scheduling policy for the thread
kt_ticksleft	Round-robin clock ticks left
*kt_timers	Pointer to thread's timer structures

Entry in struct kthread	Purpose
*kt_slink	Pointer to linked list of sleeping threads
*kt_sema	Head of per-thread alpha semaphore list
*kt_msem_info	Pointer to msemaphore info structure
*kt_chanq_infop	Pointer to channel queue info structure
kt_dil_signal	Signal to use for DIL interrupts
*kt_cred	Pointer to user credentials[a]
*kt_cdir, *kt_rdir	Curent and root directories of current thread, as shown in struct vnode
*kt_fp	Current file pointer to struct file.
*kt_link, *kt_rlink	pointers to forward run/sleep queue link and backward run queue link

[a] UID, GID, and other credentials are pointed to as a snapshot of the process-wide cred structures when the thread enters the kernel. These are only valid when a thread operates in kernel mode. Permanent changes to the cred structure (e.g., setuid()) should be made to the cred structure pointed to by the proc structure element p_cred.

vas structure

Figure 1-22 Role of the vas structure



Every process has a proc entry containing a pointer (p_vas) to the process's virtual address space. The vas maintains a doubly linked list of pregions that belong to a given process and thread. The vas is always memory resident and provides information based on the process's virtual address space.

NOTE Do not confuse the vas structure with virtual address space (VAS) in memory. The vas structure is a few bytes; VAS is 4 gigabytes.

The following table (derived from vas.h) shows the principal entries in struct vas.

Table 1-15 Entries in vas structure

Entry in struct vas	**Purpose**
va_ll	Doubly linked list of pregions
va_refcnt	Number of pointers to the vas
va_rss, va_prss, va_dprss	Cached approximation of shared and private resident set size, and private RSS in memory and on swap
*va_proc	Pointer to existing process in struct proc
va_flags	Various flags (itemized after this table)

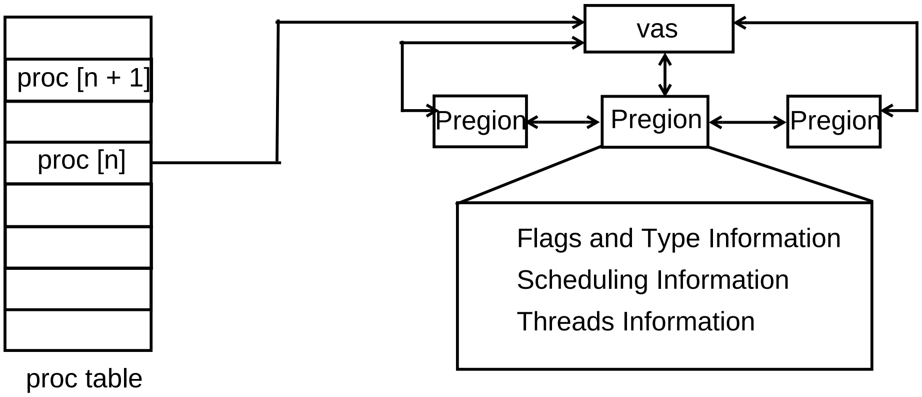
Entry in struct vas	Purpose
va_wcount	number of writable memory-mapped files sharing pseudo-vas
va_vaslock	field in struct rw_lock that controls access to vas
*va_cred	Pointer to process credentials in struct ucred
va_hdl	vas hardware-dependent information
va_ki_vss	Total virtual memory
va_ki_flag	Indication of whether vss has changed
va_ucount	Total virtual memory of user space.

The following definitions correspond to va_flags:

VA_HOLES	vas might have holes within pregions
VA_IOMAP	IOMAP pregion within the vas
VA_WRTEXT	writable text
VA_PSEUDO	pseudo vas, not a process vas
VA_MULTITHEADED	vas conected to a multithreaded process
VA_MCL_FUTURE	new pages that must be mlocked
VA_Q2SHARED	quadrant 2 used for shared data

Pregion Structure

Figure 1-23 pregion in context



The `pregion` represents an active part of the process's Virtual Address Space (VAS). This may consist of the text, data, stack, and shared memory. A pregion is memory resident and dynamically allocated as needed. Each process has a number of `pregions` that describe the regions attached to the process. In this module we will only discuss to the pregion level. The HP-UX Memory Management white paper provides more information about regions.

Table 1-16 pregion types

Type	Definition
`PT_UNUSED`	unused `pregion`
`PT_UAREA`	User area
`PT_TEXT`	Text region
`PT_DATA`	Data region
`PT_STACK`	Stack region
`PT_SHMEM`	Shared memory region

Type	**Definition**
`PT_NULLDREF`	Null pointer dereference
`PT_SIGSTACK`	Signal stack
`PT_IO`	I/O region

These `pregion` types are defined based on the value of `p_type` within the `pregion` structure and can be useful to determine characteristics of a given process. This may be accessed via the `kt_upreg` pointer in the thread table. A process has a minimum of four defined `pregions`, under normal conditions. The total number of `pregion` types defined may be identified with the definition PT_NTYPES.

Figure 1-24 Example of four possible pregions

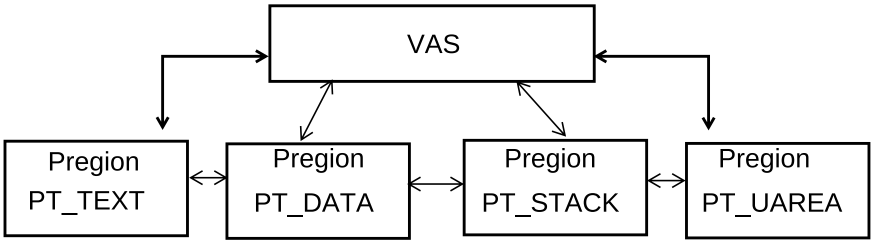


Table 1-17 Entries comprising a pregion

Type	Purpose
Structure information	Pointers to next and previous pregions Pointer and offset into the region virtual space and offset for region number of pages mapped by the pregion Pointer to the VAS
Flags and type	Referenced by p_flags and p_type
Scheduling information	Remaining pages to age (p_ageremain) Indices of next scans for vhand's age and steal hands (p_agescan, p_steadscan) Best nice value for all processes sharing the region used by the pregion (p_bestnice) sleep address for deactivation (p_deactsleep)
Thread information	Value to identify thread, for uarea pregion (p_tid)

Traversing pregion Skip List

Pregion linked lists can get quite large if a process is using many discrete memory-mapped pregions. When this happens, the kernel spends a lot of time walking the pregion list. To avoid the list being walked linearly, we use skip lists,[1] which enable HP-UX to use four forward links instead of one. These are found in the beginning of the vas and pregion structures, in the p_ll element.

[1] Skip lists were developed by William Pugh of the University of Maryland. An article he wrote for CACM can be found at ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.ps.Z.

User Structures (uarea)

The user area is a per-process structure containing data not needed in core when a process is swapped out.

The threads of a process point to the pregion containing the process's user structure, which consists of the `uarea` and kernel stack. The user structure contains the information necessary to the execution of a system call by a thread. The kernel thread's `uarea` is special in that it resides in the same address space as the process data, heap, private MMFs, and user stack. In a multi-threaded environment, each kernel thread is given a separate space for its uarea.

Each thread has a separate kernel stack.

Addressing the uarea is analogous to the prior process-based kernel structure. A kernel thread references its own uarea through struct user. However, you cannot index directly into the user structure as is possible into the `proc` table. The only way into the `uarea` is through the `kt_upreg` pointer in the thread table.

Figure 1-25 User area (uarea) in a thread-structured system

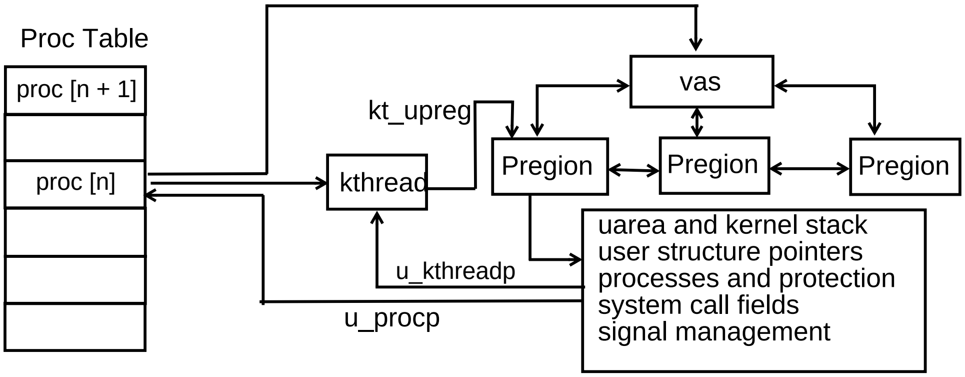


Table 1-18 **Principal entries in the uarea (struct user)**

Type	Purpose
user structure pointers	Pointers to proc and thread structures (u_procp, u_kthreadp) pointers to saved state and most recent savestate (u_sstatep,u_pfaultssp)
system call fields	arguments to current system call (u_arg[]) pointer to the arglist (u_ap) Return error code (u_error) system call return values (r_val(n))
signal management	signals to take on sigstack (u_sigonstack) saved mask from before sigpause (u_oldmask) code to trap (u_code)

The user credentials pointer (for uid, gid, etc) has been moved from the uarea and is now accessed through the p_cred() accessor for the proc structure and the kt_cred()accessor for the kthread structure. See comments under the kt_cred() field in kthread.h for details governing usage.

Process Control Block (pcb)

NOTE HP-UX now handles context switching on a per-thread basis.

A process control block (pcb) is maintained in the user structure of each kernel thread as a repository for thread scheduling information. The pcb contains all the register states of a kernel thread that are saved or restored during a context switch from one threads environment to another.

The context of a current running thread is saved in its associated uarea pcb when a call to swtch() is made. The save() routine saves the current thread state in the pcb on the switch out. The resume() routine maps the user-area of the newly selected thread and restores the process registers from the pcb. When we return from resume(), the selected thread becomes the currently running thread and its uarea is automatically mapped into the virtual memory address of the system's global uarea.

The register's context includes:

- General-purpose Registers
- Space registers
- Control registers
- Instruction Address Queues (Program Counter)
- Processor Status Word (PSW)
- Floating point register

Table 1-19 **Contents of theProcess Control Block (pcb)**

Context element	Purpose
General registers pcb_r1 --> pcb_r31 [GR0 - GR31]	Thirty two general registers that provide the central resource for all computation. These are available for programs at all privilege levels.
Space registers pcb_sr0 --> pcb_sr7 [SR0 - SR7]	Eight space ID registers for virtual addressing.
Control registers pcb_cr0 --> pcb_cr31 [CR0,CR8 - CR31]	Twenty-five control registers that contain system state information.

Context element	Purpose
Program counters (`pcb_pc`)	Two registers that hold the virtual address of the current and next instruction to be executed. • The Instruction Address Offset Queue (IAOQ) is 32 bits long. The upper 30 bits contain the work offset of the instruction and the lower 2 bits maintain the privilege level of the corresponding instruction. • The Instruction Address Space Queue(IASQ) is 32 bits long in a PA-RISC 2.0 (64-bit) system or 16 bits a PA-RISC 1.x (32-bit) system. Contains the Space ID for instructions
Processor Status Word (`pcb_psw`)	Contains the machine level status that relates to a process as it does operations and computations.
Floating point registers `pcb_fr1 --> pcb_fr32`	Maintains the floating point status for the process.

Process Scheduling

To understand how threads of a process run, we have to understand how they are scheduled. Although processes appear to the user to run simultaneously, in fact a single processor is executing only one thread of execution at any given moment.

Several factors contribute to process scheduling:

- Kind of scheduling policy required -- timeshare or real-time. Scheduling policy governs how the process (or thread of execution) interacts with other processes (or threads of execution) at the same priority.
- Choice of scheduler. Four schedulers are available: HP-UX timeshare scheduler (SCHED_HPUX), HP Process Resource Manager (a timeshare scheduler), HP-UX real-time scheduler (HPUX_RTPRIO), and the POSIX-compliant real-time scheduler.
- Priority of the process. Priority denotes the relative importance of the process or thread of execution.
- Run queues from which the process is scheduled.
- Kernel routines that schedule the process.

Scheduling Policies

HP-UX scheduling is governed by policy that connotes the urgency for which the CPU is needed, as either timeshare or real-time. The following table compares the two policies in very general terms.

Table 1-20 **Comparison of Timeshare vs Real-time scheduling**

Timeshare	Real-Time
Typically implemented round-robin.	Implemented as either round-robin or first-in-first-out (FIFO), depending on scheduler.
Kernel lowers priority when process is running; that is, timeshare priorities decay. As you use CPU, your priority becomes weaker. As you become starved for CPU, your priority becomes stronger. Scheduler tends to regress toward the mean.	Priority not adjusted by kernel; that is, real-time priorities are non-decaying. If a real-time priority is set at 50 and another real-time priority is set at 40 (where 40 is stronger than 50), the process or thread of priority 40 will always be more important than the process or thread of priority 50.
Runs in timeslices that can be preempted by process running at higher priority.	Runs until exits or is blocked. Always runs at higher priority than timeshare.

The principle behind the distribution of CPU time is called a timeslice. A timeslice is the amount of time a process can run before the kernel checks to see if there is an equal or stronger priority process ready to run.

- If a timeshare policy is implemented, a process might begin to run and then relinquish the CPU to a process with a stronger priority.
- Real-time processes running round-robin typically run until they are blocked or relinquish CPU after a certain timeslice has occurred.

  Real-time processes running FIFO run until completion, without being preempted.

Scheduling policies act upon sets of thread lists, one thread list for each priority. Any runnable thread may be in any thread list. Multiple scheduling policies are provided. Each nonempty list is ordered, and contains a head (`th_link`) as one end of its order and a tail (`th_rlink`) as the other. The purpose of a scheduling policy is to define the allowable operations on this set of lists (for example, moving threads between and within lists).

Each thread is controlled by an associated scheduling policy and priority. Applications can specify these parameters by explicitly executing the `sched_setscheduler()` or `sched_setparam()` functions.

Hierarchy of Priorities (overview)

All POSIX real-time priority threads have greater scheduling importance than threads with HP-UX real-time or HP-UX timeshare priority. By comparison, all HP-UX real-time priority threads are of greater scheduling importance than HP-UX timeshare priority threads, but are of lesser importance than POSIX real-time threads. Neither POSIX nor HP-UX real-time threads are subject to degradation.

This will be demonstrated in detail shortly.

Schedulers

As of release 10.0, HP-UX implements four schedulers, two time-share and two real-time.

To choose a scheduler, you can use the user command, `rtsched(1)`, which executes processes with your choice of scheduler and enables you to change the real-time priority of currently executing process ID.

```
rtsched -s scheduler -p priority command [arguments]
rtsched [ -s scheduler ] -p priority -P pid
```

Likewise, the system call `rtsched(2)` provides programmatic access to POSIX real-time scheduling operations.

RTSCHED (POSIX) Scheduler

The RTSCHED POSIX-compliant real-time deterministic scheduler provides three scheduling policies, whose characteristics are compared in the following table.

Table 1-21 **RTSCHED policies**

Scheduling Policy	How it works
SCHED_FIFO	Strict first in-first out (FIFO) scheduling policy. This policy contains a range of at least 32 priorities. Threads scheduled under this policy are chosen from a thread list ordered according to the time its threads have been in the list without being executed. The head of the list is the thread that has been in the list the longest time; the tail is the thread that has been in the list the shortest time.
SCHED_RR	Round-robin scheduling policy with a per-system time slice (time quantum). This policy contains a range of at least 32 priorities and is identical to the SCHED_FIFO policy with an additional condition: when the implementation detects that a running process has been executing as a running thread for a time period of length returned by the function sched_rr_get_interval(), or longer, the thread becomes the tail of its thread list, and the head of that thread list is removed and made a running thread.
SCHED_RR2	Round-robin scheduling policy, with a per-priority time slice (time quantum). The priority range for this policy contains at least 32 priorities. This policy is identical to the SCHED_RR policy except that the round-robin time slice interval returned by sched_rr_get_interval() depends upon the priority of the specified thread.

SCHED_RTPRIO Scheduler

Realtime scheduling policy with nondecaying priorities (like SCHED_FIFO and SCHED_RR) with a priority range between the POSIX real-time policies and the HP-UX policies.

For threads executing under this policy, the implementation must use only priorities within the range returned by the functions sched_get_priority_max() and sched_get_priority_min() when SCHED_RTPRIO is provided as the parameter.

NOTE In the SCHED_RTPRIO scheduling policy, smaller numbers represent higher (stronger) priorities, which is the opposite of the POSIX scheduling policies. This is done to provide continuing support for existing applications that depend on this priority ordering.

The strongest priority in the priority range for SCHED_RTPRIO is weaker than the weakest priority in the priority ranges for any of the POSIX policies, SCHED_FIFO, SCHED_RR, and SCHED_RR2.

SCHED_HPUX Scheduler

The SCHED_OTHER policy, also known as SCHED_HPUX and SCHED_TIMESHARE, provides a way for applications to indicate, in a portable way, that they no longer need a real-time scheduling policy.

For threads executing under this policy, the implementation can use only priorities within the range returned by the functions sched_get_priority_max() and sched_get_priority_min() when SCHED_OTHER is provided as the parameter. Note that for the SCHED_OTHER scheduling policy, like SCHED_RTPRIO, smaller numbers represent higher (stronger) priorities, which is the opposite of the POSIX scheduling policies. This is done to provide continuing support for existing applications that depend on this priority ordering. However, it is guaranteed that the priority range for the SCHED_OTHER scheduling policy is properly disjoint from the priority ranges of all of the other scheduling policies described and the strongest priority in the priority range for SCHED_OTHER is weaker than the weakest priority in the priority ranges for any of the other policies, SCHED_FIFO, SCHED_RR, and SCHED_RR2.

Process Resource Manager

The Process Resource Manager (PRM) is an optional HP-UX product coded into the kernel as fss, or Fair Share Scheduler. This time-share scheduler operates on timeshare processes. Real-time processes (RTPRIO and POSIX real-time) are not affected by the fss, but they do affect it, because the fss allocates portions of the CPU to different groups of processes. Unlike the default SCHED_HPUX scheduler, the Process Resource Manager allows the system administrator to budget CPU time to groups of processes with a high degree of specificity.

The remainder of this section will not be dealing with the ramifications of the PRM.

Scheduling Priorities

All processes have a priority, set when the process is invoked and based on factors such as whether the process is running on behalf of user or system and whether the process is created in a time-share or real-time environment.

Associated with each policy is a priority range. The priority ranges foreach policy can (but need not) overlap the priority ranges of other policies.

Two separate ranges of priorities exist: a range of POSIX standard priorities and a range of other HP-UX priorities. The POSIX standard priorities are always higher than all other HP-UX priorities.

Processes are chosen by the scheduler to execute a time-slice based on priority. Priorities range from highest priority to lowest priority and are classified by need. The thread selected to run is at the head of the highest priority nonempty thread list.

Internal vs. External Priority Values

With the implementation of the POSIX `rtsched`, HP-UX priorities are enumerated from two perspectives -- internal and external priority values.

- The internal value represents the kernel's view of the priority.
- The external value represents the user's view of the priority, as is visible using the `ps(1)` command.

In addition, legacy HP-UX priority values are ranked in opposite sequence from POSIX priority values:

- In the POSIX standard, the higher the priority number, the stronger the priority.
- In legacy HP-UX implementation, the lower the priority number, the stronger the priority.

The following macros are defined in `pm_rtsched.h` to enable a program to convert between POSIX and HP-UX priorities and internal to external values:

- **`PRI_ExtPOSIXPri_To_IntHpuxPri`**

To derive the HP-UX kernel (internal) value from the value passed by a user invoking the `rtsched` command (that is, using the POSIX priority value).

- **`PRI_IntHpuxPri_To_ExtPOSIXPri()`**

  To convert HP-UX (kernel) internal priority value to POSIX priority value.

- **`PRI_IntHpuxPri_To_ExtHpuxPri`**

  To convert HP-UX internal to HP-UX external priority values.

`rtsched_numpri` Parameter

A configurable parameter, `rtsched_numpri`, controls:

- The number of scheduling priorities supported by the POSIX rtsched scheduler.
- The range of valid values is 32 to 512 (32 is default)

Increasing `rtsched_numpri` provides more scheduling priorities at the cost of increased context switch time, and to a minor degree, increased memory consumption.

Schedulers and Priority Values

There are now four sets of thread priorities: (Internal to External View)

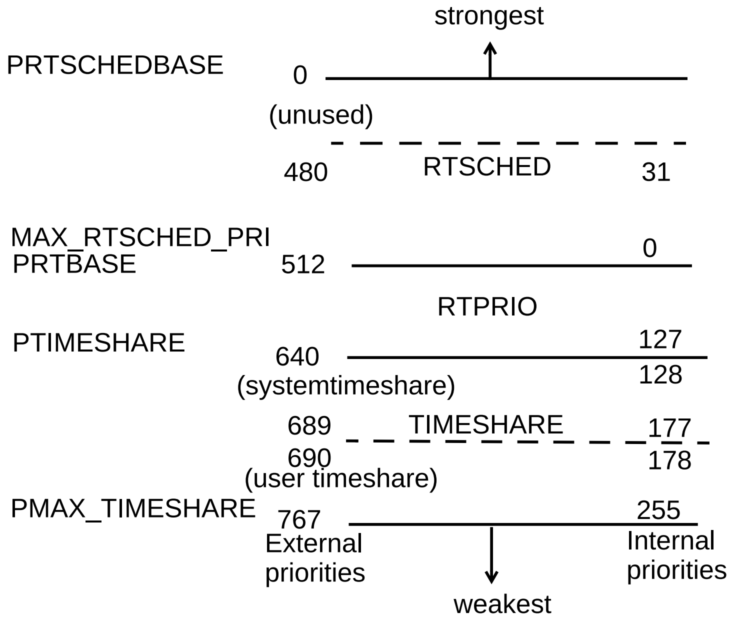
Table 1-22 **Scheduler priority values**

Type of Scheduler	External Values	Internal Values
POSIX Standard	512 to 480	0 to 31
Real-time	512 to 640	0 to 127
System, timeshare	640 to 689	128 to 177
User, timeshare	690 to 767	178 to 255

NOTE For the POSIX standard scheduler, the higher the number, the stronger the priority. For the RTPRIO scheduler, the lower the number, the stronger the priority.

The following figure demonstrates the relationship of the three schedulers ranked by priority and strength.

Figure 1-26 Schedulers and Priority values



The following lists categories of priority, from highest to lowest:

- RTSCHED (POSIX standard) ranks as highest priority range, and is separate from other HP-UX priorities.

  RTSCHED priorities range between 32 and 512 (default 32) and can be set by the tunable parameter rtsched_numpri.

- SCHED_RTPRIO (real-time priority) ranges from 0-127 and is reserved for processes started with rtprio() system calls.
- Two priorities used in a timeshare environment:
  - User priority (178-255), assigned to user processes in a time-share environment.
  - System priority (128-177), used by system processes in a time-share environment.

The kernel can alter the priority of time-share priorities (128-255) but not real-time priorities (0-127).

The following priority values, internal to the kernel, are defined in `param.h`:

`PRTSCHEDBASE` Smallest (strongest) `RTSCHED` priority

`MAX_RTSCHED_PRI` Maximum number of `RTSCHED` priorities

`PRTBASE` Smallest (strongest) `RTPRIO` priority. Defined as `PRTSCHED + MAX_RTSCHED_PRI`.

`PTIMESHARE` Smallest (strongest) timeshare priority. Defined as `PRTBASE + 128`.

`PMAX_TIMESHARE` Largest (weakest) timeshare priority. Defined as `127 + PTIMESHARE`.

Priorities stronger (smaller number) than or equal to `PZERO` cannot be signaled. Priorities weaker (bigger number) than `PZERO` can be signaled.

RTSCHED Priorities

The following discussion illustrates the HP-UX internal view, based on how the user specifies a priority to the `rtsched` command. Each available real-time scheduler policy has a range of priorities (default values shown below).

Scheduler Policy	**highest priority**	**lowest priority**
`SCHED_FIFO`	31	0
`SCHED_RR`	31	0
`SCHED_RR2`	31	0
`SCHED_RTPRIO`	0	127

The user may invoke the rtsched(1) command to assign a scheduler policy and priority. For example,

```
rtsched -s SCHED_RR -p 31 ls
```

Within kernel mode `sched_setparam()` is called to set the scheduling parameters of a process. It (along with `sched_setscheduler()`) is the mechanism by which a process changes its (or another process') scheduling parameters. Presently the only scheduling parameter is priority, `sched_priority`.

The `sched_setparam()` and `sched_setscheduler()` system calls look up the process associated with the user argument `pid`, and call the internal routine `sched_setcommon()` to complete the execution.

`sched_setcommon()` is the common code for `sched_setparam()` and `sched_setscheduler()`. It modifies the threads scheduling priority and policy. The scheduler information for a thread is kept in its thread structure. It is used by the scheduling code, particularly `setrq()`, to decide when the thread runs, with respect to the other threads in the system. `sched_setcommon()` is called with the `sched_lock` held.

`sched_setcommon()` calls the macro `PRI_ExtPOSIXPri_To_IntHpuxPri`, defined in `pm_rtsched.h`. The priority requested is then converted. Since priorities in HP-UX are stronger for smaller values, and the POSIX specification requires the opposite behavior, we merge the two by running the `rtsched` priorities from (`(MAX_RTSCHED_PRI-1) - rtsched_info.rts_numpri`) (strongest) to (`MAX_RTSCHED_PRI-1`) (weakest).

Based on the macro definition using the value passed by the user, the internal value seen by the kernel is calculated as follows:

`((MAX_RTSCHED_PRI - 1) - (ExtP_pri))`

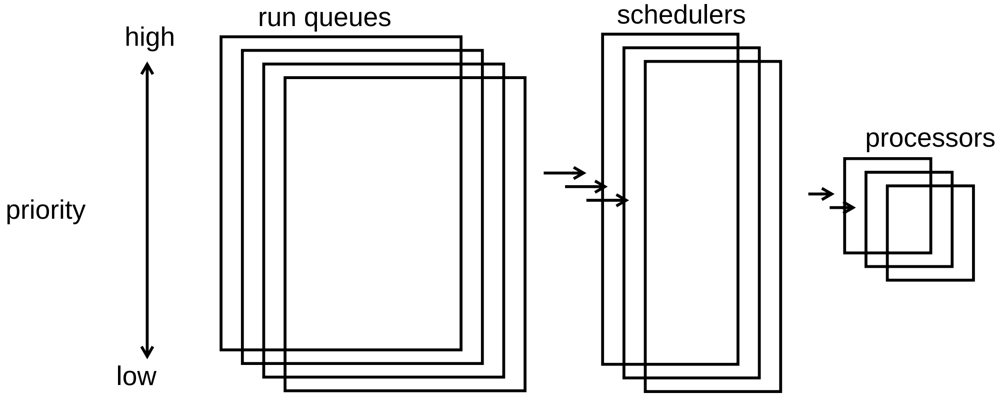
`512 - 1  - 31   = 480`

The kernel priority of the user's process is 480. The value of 480 is the strongest priority available to the user.

Run Queues

A process must be on a queue of runnable processes before the scheduler can choose it to run.

Figure 1-27 Run queues of thread lists waiting to run



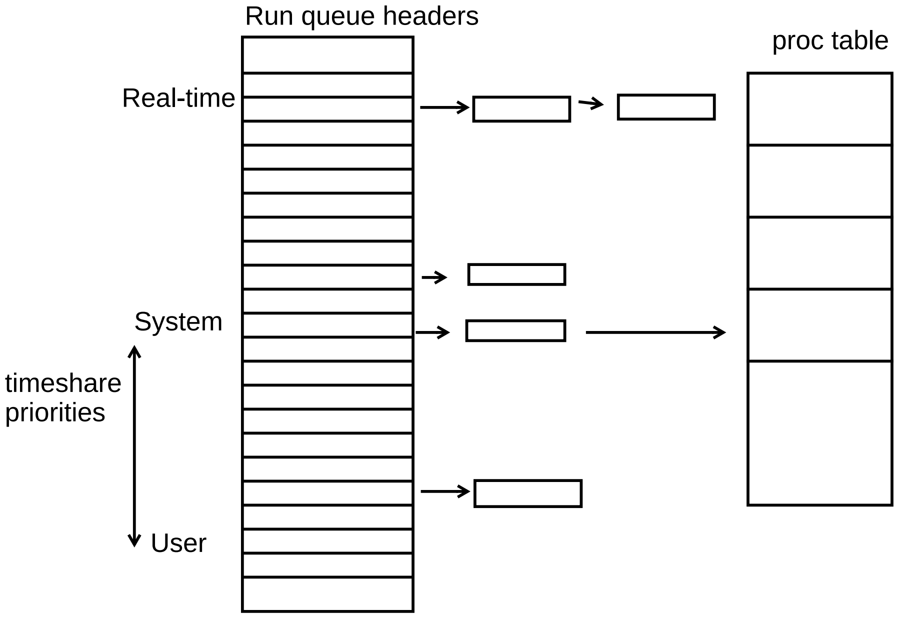
Processes get linked into the run queue based on the process's priority, set in the process table. Run queues are link-listed in decreasing priority. The scheduler chooses the process with the highest priority to run for a given time-slice.

Each process is represented by its header on the list of run queue headers; each entry in the list of run queue headers points to the process table entry for its respective process.

The kernel maintains separate queues for system-mode and user-mode execution. System-mode execution takes precedence for CPU time. User-mode priorities can be preempted -- stopped and swapped out to secondary storage; kernel-mode execution cannot. Processes run until they have to wait for a resource (such as data from a disk, for example), until the kernel preempts them when their run time exceeds a time-slice limit, until an interrupt occurs, or until they exit. The scheduler then chooses a new eligible highest-priority process to run; eventually, the original process will run again when it has the highest priority of any runnable process.

When a timeshare process is not running, the kernel improves the process's priority (lowers its number). When a process is running, its priority worsens. The kernel does not alter priorities on real-time processes. Timeshared processes (both system and user) lose priority as they execute and regain priority when they do not execute

Figure 1-28 Run queue, showing priorities



Run Queue Initialization

Run queues are initialized by the routine `rqinit()`, which is called from `init_main.c` after system monarch processor is established and before final kernel initialization.

`rqinit` examines all potential entries in the system global per-processor information structure (`struct mpinfo`), gets the run queue information and pointers to the linked list of running threads. It then clears the run queue data in `bestq` (an index into the array of run queue points which points to the highest priority non-empty queue), `newavg_on_rq` (the run queue average for the processor), `nready_locked` and `nready_free` (sums provided the total threads in the processor's run queues). `rqinit` then sets the current itimer value for all run queues, links the queue header as the sole element, and sets up the queue.

Next, the RTSCHED-related global run data structures are initialized with the global structure rtsched_info (defined in pm_rtsched.h), which describes the RTSCHED run queues.

Table 1-23 Entries in rtsched_info

Entry	Purpose
rts_nready	Total number of threads on queues
rts_bestq	Hint of which queue to find threads
rts_numpri	Number of RTSCHED priorities
rts_rr_timeslice	Global timeslice for SCHED_RR threads
*rts_timeslicep	Round-robin timeslices for each priority (used by SCHED_RR2 threads)
*rts_qp	Pointer to run queues
*rts_lock	Spinlock for the run queues

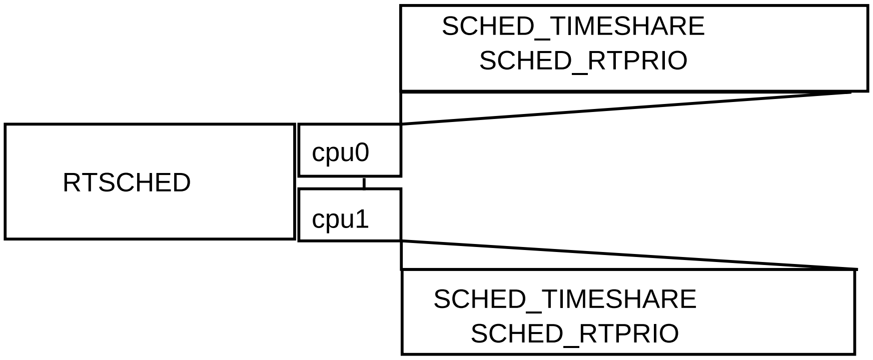
The tunable parameter rtsched_numpri determines how many run queues exist:

- The minimum value allowed is 32, imposed by the POSIX.4 specification and defined as RTSCHED_NUMPRI_FLOOR.
- The maximum supported value of 512 is a constant of the implementation, defined as RTSCHED_NUMPRI_CEILING and set equal to MAX_RTSCHED_PRI. If a higher maximum is required, the latter definition must be changed.

malloc() is called to allocate space for RTSCHED run queues. (rtsched_numpri * sizeof (struct mp_threadhd)) bytes are required. The resulting pointer is stored in rtsched_info.rts_qp.

Timeslice is checked to ensure that it is set to a valid value, which may be either -l (meaning no timeslicing) or positive integers. If it is invalid, it is set to the default, HZ/10. rtsched_info.rts_rr_timeslice is set to timeslice, which round-robins with that many clock ticks. For each of the rtsched_numpri run queues, the struct mp_threadhd header block is linked circularly to itself. Finally, a spinlock is allocated to lock the run queue.

Figure 1-29 Run queue initialization



Note, there is one RTSCHED run queue systemwide, though separate track is kept for each processor. The queue for given thread is based on how the scheduling policy is defined. One global set of run queues is maintained for RTSCHED (SCHED_FIFO, SCHED_RR, SCHED_RR2) threads. Run queues are maintained for each SPU for SCHED_TIMESHARE and SCHED_RTPRIO threads.

RTSCHED Run Queue

The following figure shows threads set to run at various RTSCHED priorities.

The global RTSCHED run queues are searched for the strongest (most deserving) thread to run; the best candidate is returned as a kthread_t. Each priority has one thread list. Any runnable thread may be in any thread list. Multiple scheduling policies are provided. Each nonempty list is ordered, and contains a head (th_link) at one end of its order and a tail (th_rlink) at the other.

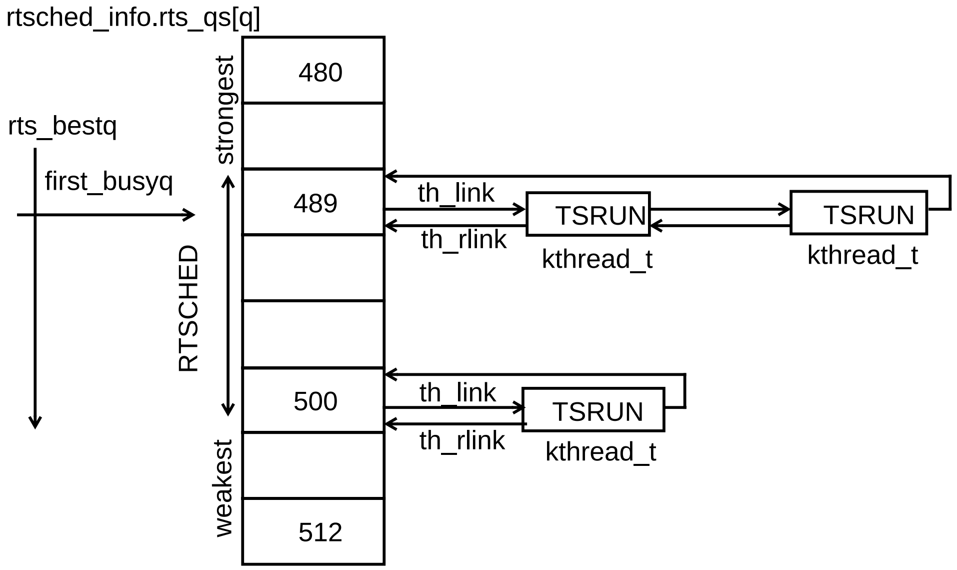
- rtsched_info.rts_qp points to the strongest RTSCHED queue.
- rtsched_info.rts_bestq points to the queue to begin the search.

The search (by the routine find_thread_rtsched()) proceeds from rts_bestq downwards looking for non-empty run queues. When the first non-empty queue is found, its index is noted in the local first_busyq. All threads in that queue are checked to determine if they are truly runnable or blocked on a semaphore.

- If there is a runnable thread, the rts_bestq value is updated to the present queue and a pointer to the thread found is returned to the caller.
- If no truly runnable thread is found, threads blocked on semaphores are considered.

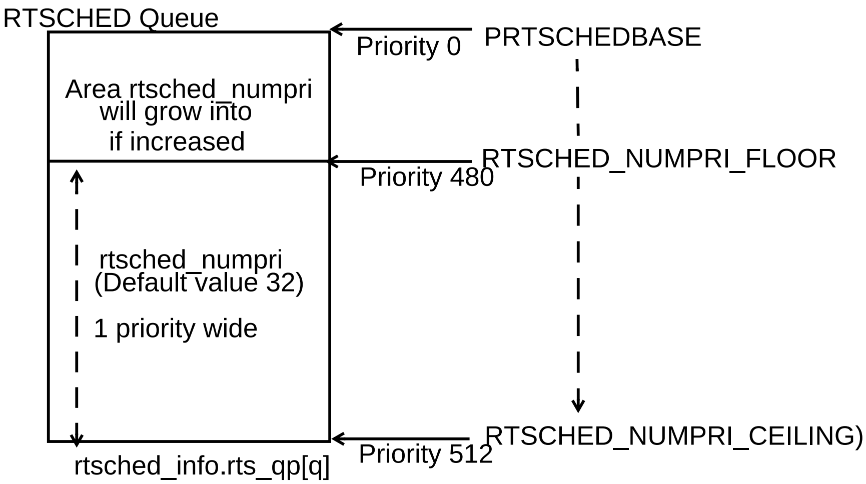
If first_busyq is set, the rts_bestq value is updated to it and the thread at the head of that queue is returned to the caller. If first_busyq did not get set in the loop, the routine panics, because it should be called only if rtsched_info.rts_nready is non-zero.

Figure 1-30 RTSCHED run queue detail



Although the threads scheduler is set to a default value of 32 (RTSCHED_NUMPRI_FLOOR), it can be expanded to a system limit of PRTSCHEDBASE (a value of 0).

Figure 1-31 Threads scheduler

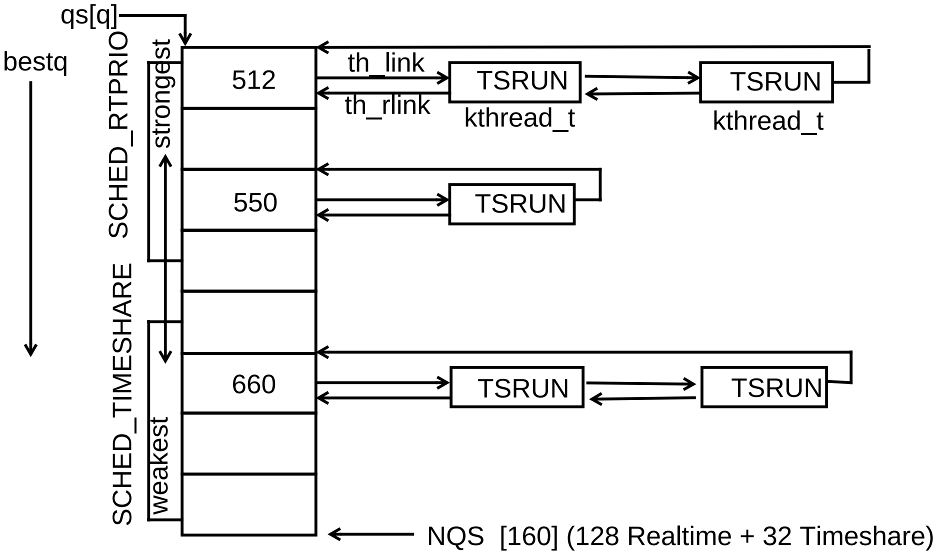


The Combined SCHED_RTPRIO and SCHED_TIMESHARE Run Queue

As shown in the following figure, the SCHED_RTPRIO and SCHED_TIMESHARE priorities use the same queue.

The SCHED_RTPRIO and SCHED_TIMESHARE queue is searched with the same technique as the RTSCHED queue. The most deserving thread is found to run on the current processor. The search starts at bestq, which is an index into the table of run queues. There is one thread list for each priority. Any runable thread may be in any thread list. Multiple scheduling policies are provided. Each nonempty list is ordered, and contains a head (th_link) as one end of its order and a tail (th_rlink) as the other.

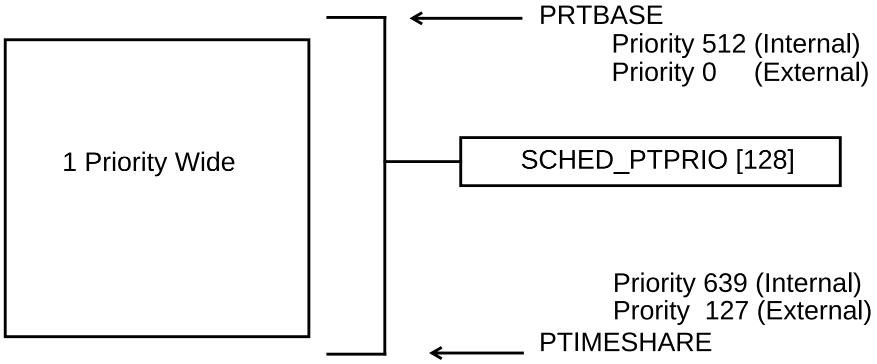
Figure 1-32 SCHED_RTPRIO and SCHED_TIMESHARE queue



The `mp_rq` structure constructs the run queues by linking threads together. The structure `qs` is an array of pointer pairs that act as a doubly linked list of threads. Each entry in qs[] represents a different priority queue. sized by `NQS`, which is `160`. The `qs[].th_link` pointer points to the first thread in the queue and the `qs[].th_rlink` pointer points to the tail.

RTPRIO Run Queue

Figure 1-33 SCHED_RTPRIO (HP-UX REAL TIME) run queue



Priorities 0 (highest realtime priority) through 127 (least realtime priority) are reserved for real time threads. The real time priority thread will run until it sleeps, exits, or is preempted by a higher priority real time thread. Equal priority threads will be run in a round robin fashion.

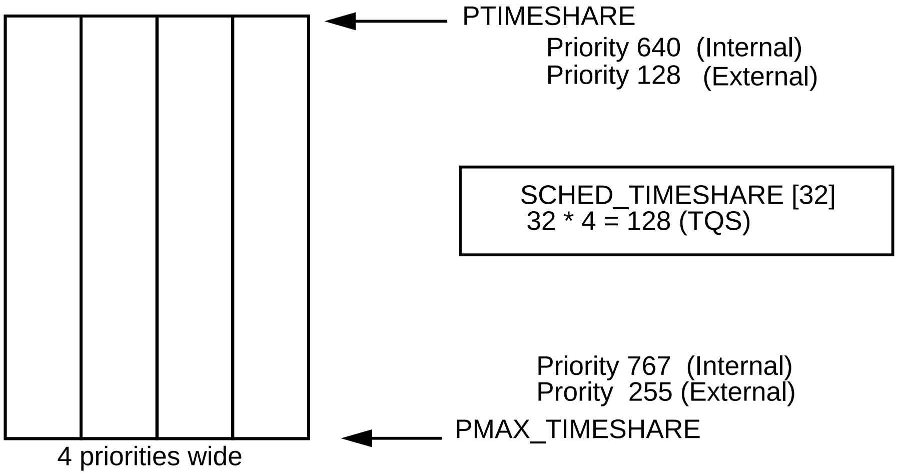
The `rtprio(1)` command may be used to give a thread a real time priority. To use the `rtprio(1)` command a user must belong in the `PRIV_RTPRIO` privilege group or be superuser (`root`). The priorities of real time threads are never modified by the system unless explicitly requested by a user (via a command or system call). Also a real time thread will always run before a time share thread.

The following are a few key points regarding a real-time thread:

- Priorities are not adjusted by the kernel
- Priorities may be adjusted by a system call
- Real-time priority is set in `kt_pri`
- The `p_nice` value has no effect

SCHED_TIMESHARE Run Queue

Figure 1-34 SCHED_TIMESHARE run queue



Timeshare threads are grouped into system priorities (128 through 177) and user priorities (178 through 255). The queues are four priorities wide. The system picks the highest priority timeshare thread, and lets it run for a specific period of time (timeslice). As the thread is running its priority decreases. At the end of the time slice, a new highest priority is chosen.

Waiting threads gain priority and running threads lose priority in order to favor threads that perform I/O and give lesser attention to compute-bound threads.

`SCHED_TIMESHARE` priorities are grouped as follows:

- Real-time priority thread: range 0-127
- Time-share priority thread: range 128-255
- System-level priority thread: range 128-177
- User-level priority thread: range 178-255

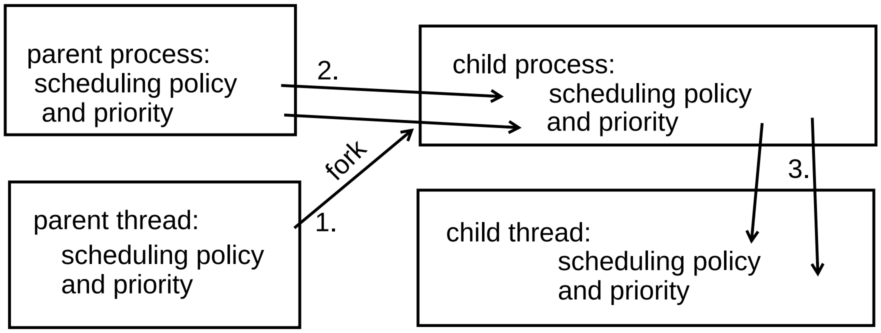
`RTSCHED` priority queues are one priority wide; timeshare priority queues are four priorities wide.

Thread Scheduling

The thread of a parent process forks a child process. The child process inherits the scheduling policy and priority of the parent process. As with the parent thread, it is the child thread whose scheduling policy and priority will be used.

The following figure illustates the flow of creation.

Figure 1-35 Inheritance of Scheduling policy and priority



- Each thread in a process is independently scheduled.
- Each thread contains its own scheduling policy and priority
- Thread scheduling policies and priorities may be assigned before a thread is created (in the threads attributes object) or set dynamically while a thread is running.
- Each thread may be bound directly to a CPU.
- Each thread may be suspended (and later resumed) by any thread within the process.

The following scheduling attributes may be set in the threads attribute object. The newly created thread will contain these scheduling attributes:

`contentionscope`

`PTHREAD_SCOPE_SYSTEM` specifies a bound (1 x 1, kernel-spacel) thread. When a bound thread is created, both a user thread and a kernel-scheduled entity are created.

`PTHREAD_SCOPE_PROCESS` will specify an unbound (M x N, combination user- and kernel-space) thread. (Note, HP-UX release 10.30 does not support unbound threads.)

`inheritsched`

`PTHREAD_INHERIT_SCHED` specifies that the created thread will inherit its scheduling values from the creating thread, instead of from the threads attribute object.

`PTHREAD_EXPLICIT_SCHED` specifies that the created thread will get its scheduling values from the threads attribute object.

`schedpolicy`

The scheduling policy of the newly created thread

`schedparam`

The scheduling parameter (priority) of the newly created thread.

Timeline

A process and its thread change with the passage of time. A thread's priority is adjusted four key times, as shown in the next figure and described in the table that follows..

Figure 1-36 Thread scheduling timeline

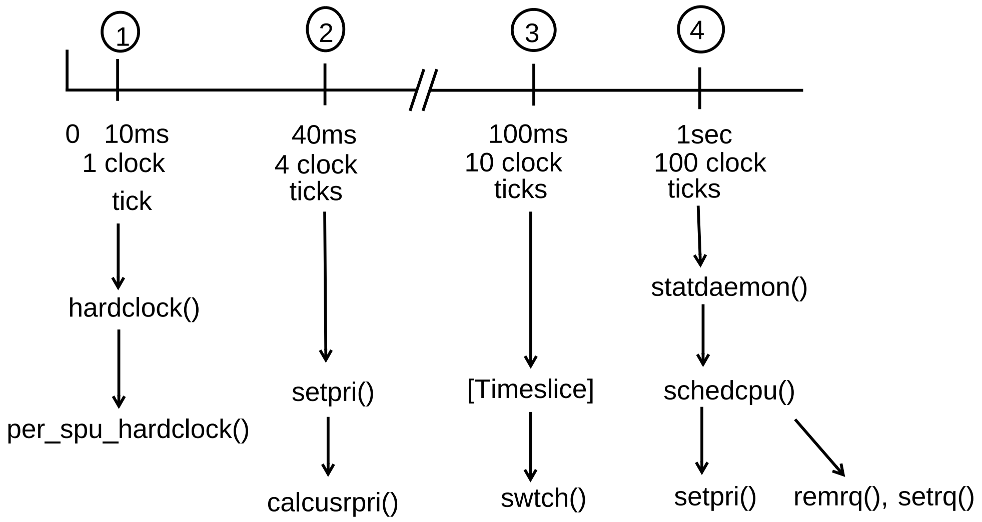


Table 1-24 **Thread priority adjustments**

Interval	What happens
10 milliseconds	The clock-interrupt handling routine `clock_int()` adjusts a time interval on the monarch every clock tick. The monarch processor calls `hardclock()` to handle clock ticks on the monarch for general maintenance (such as disk and LAN states). `hardclock()` calls `per_spu_hardclock()` to charge the running thread with cpu time accumulated (`kt_cpu`).
40 milliseconds	`per_spu_hardclock()` determines the running thread has accumulated 40ms of time and calls `setpri()`. `setpri()` calls `calcusrpri()` to adjust the running thread's user priority (`kt_usrpri`).
100 milliseconds	By default, 10 clock ticks represents the value of timeslice, the configurable kernel parameter that defines the amount of time one thread is allowed to run before the CPU is given to the next thread. Once a timeslice interval has expired a call to `swtch()` is made to enact a context switch.
one second	`statdaemon()` loops on the thread list and once every second calls `schedcpu()` to update all thread priorities. The `kt_usrpri` priority is given to the thread on the next context switch; if in user mode `kt_usrpri` is given immediately.

Thread Scheduling Routines

The following table summarizes the principal thread scheduling routines.

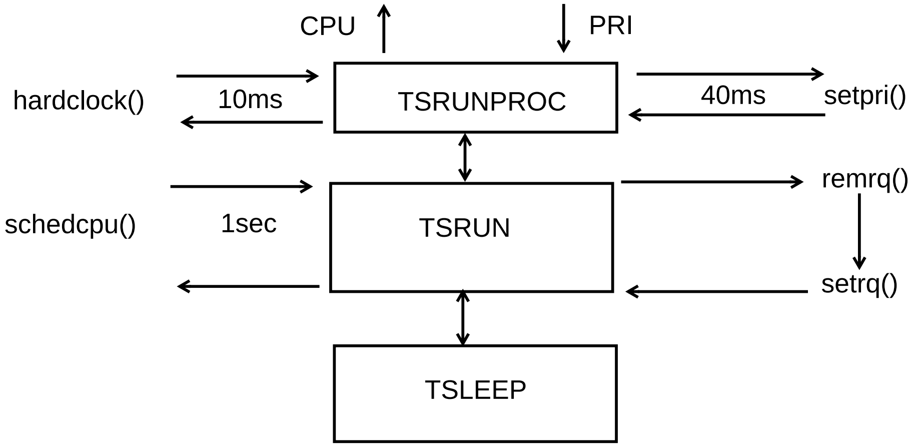
Table 1-25 **Thread scheduling routines**

Routine	Purpose
hardclock()	Runs on the monarch processor to handle clock ticks.
per_spu_hardclock()	handles per-processor hardclock activities.
setpri()	Called with a thread as its argument and returns a user priority for that thread. Calls calcusrpri() to get the new user priority. If the new priority is stronger than that of the currently running thread, setpri() generates an MPSCHED interrupt on the processor executing that thread, stores the new user priority in kt_usrpri and returns it to its caller.
calcusrpri()	The user priority (kt_usrpri) portion of setpri(). calcusrpri() uses the kt_cpu and p_nice(proc) fields of the thread, tt, to determine tt's user priority and return that value without changing any fields in *tt. If tt is a RTPRIO or RTSCHED thread, kt_usrpri is the current value of kt_pri.
swtch()	Finds the most deserving runnable thread, takes it off the run queue, and sets it to run.
statdaemon()	A general-purpose kernel process run once per second to check and update process and virtual memory artifacts, such as signal queueing and free protection IDs. Calls schedcpu() to recompute thread priorities and statistics.

Routine	Purpose
`schedcpu()`	Once a second, `schedcpu()` loops through the thread list to update thread scheduling priorities. If the system has more than one SPU, it balances SPU loads. schedcpu updates thread usage information (`kt_prevrecentcycles` and `kt_fractioncpu`), calculates new `kt_cpu` for the current thread (info used by `setpri()`, updates the statistics of runnable threads on run queues and those swapped out, and awakens the swapper. Calls `setpri()`.
`setrq()`	Routine used to put threads onto the run queues. Set the appropriate protection (`spl7` in UP case, thread lock in MP case). Assert valid HP-UX priority and scheduling policy and perform policy-specific setup
`remrq()`	Routine used to remove a thread from its run queue. With a valid kt_link, set the appropriate protection (`spl7` in the UP case or thread lock in MP case). Find the processor on which the thread is running. Decrement the thread count on run queues. Update the mpinfo structure. Restore the old `spl` level, update `RTSCHED` counts if necessary. Adjust the `kt_pri`, return to `schedcpu.`

Adjusting a Thread Priority

Figure 1-37 Adjusting a thread priority



Every 10 msecs, the routine `hardclock()` is called with spinlock `SPL5` to disable I/O modules and software interrupts. `hardclock()` calls the per-processor routine `per_spu_hardclock()`, which looks for threads whose priority is high enough to run. (Searching the processor run queues depends on the scheduling policy). If a thread is found, the `MPSCHED_INT_BIT` in the processor EIRR (External Interrupt Request Register) is set.

When the system receives an `MPSCHED_INT` interrupt while running a thread in user mode, the trap handler puts the thread on a run queue and switches context, to bring in the high-priority thread.

If the current executing thread is the thread with the highest priority, it is given 100ms (one timeslice) to run. `hardclock()` calls setpri() every 40ms to review the thread's working priority (`kt_pri`). `setpri()` adjusts the user priority (`kt_usrpri`) of a time-share thread process based on `cpu` usage and `nice` values. While a time-share thread is running, `kt_cpu` time increases and its priority (`kt_pri`) worsens. `RTSCHED` or `RTPRIO` thread priorities do not change.

Every 1 second, `schedcpu()` decrements the `kt_cpu` value for each thread on the run queue. `setpri()` is called to calculate a new priority of the current thread being examined in the `schedcpu()` loop. `remrq()` is called to remove that thread from the run queue and then `setrq()` places the thread back into the run queue according to its new priority.

If a process is sleeping or on a swap device (that is, not on the run queue), the user priority (`kt_usrpri`) is adjusted in `setpri()` and `kt_pri` is set in `schedcpu()`.

Context Switching

In a thread-based kernel, the kernel manages context switches between kernel threads, rather than processes. Context switching occurs when the kernel switches from executing one thread to executing another. The kernel saves the context of the currently running thread and resumes the context of the next thread that is scheduled to run. When the kernel preempts a thread, its context is saved. Once the preempted thread is scheduled to run again, its context is restored and it continues as if it had never stopped.

The kernel allows context switch to occur under the following circumstances:

- Thread exits
- Thread's time slice has expired and a trap is generated.
- Thread puts itself to sleep, while awaiting a resource.
- Thread puts itself into a debug or stop state
- Thread returns to user mode from a system call or trap
- A higher-priority thread becomes ready to run

If a kernel thread has a higher priority than the running thread, it can preempt the current running thread. This occurs if the thread is awakened by a resource it has requested. Only user threads can be preempted. HP-UX does not allow preemption in the kernel except when a kernel thread is returning to user mode.

In the case where a single process can schedule multiple kernel threads (1 x 1 and M x N), the kernel will preempt the running thread when it is executing in user space, but not when it is executing in kernel space (for example, during a system call).

The `swtch()` Routine

The `swtch()` routine finds the most deserving runnable thread, takes it off the run queue, and starts running it. The following figure shows the routines used by `swtch()`.

Figure 1-38 **process scheduling -- swtch()**

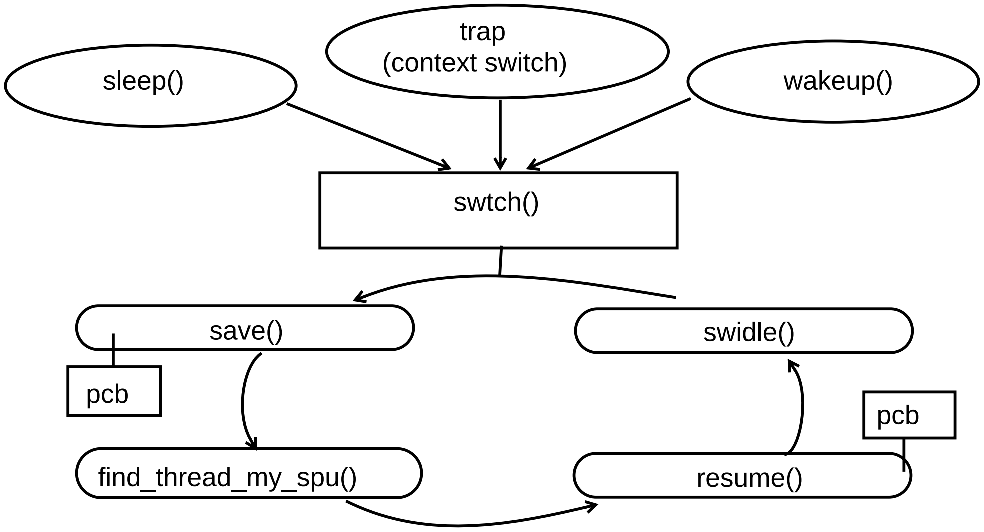


Table 1-26 **swtch() routines**

Routine	Purpose

`swidle()`
`(asm_utl.c)`
	Performs an idle loop while waiting to take action.
	Checks for a valid `kt_link`.
	On a uniprocessor machine without a threadlock thread, goes to `spl7`.
	Finds the thread's spu.
	Decrements the count of threads on run queues.
	Updates `ndeactivated, nready_free, nready_locked` in the `mpinfo()` structure.
	Removes the thread from its run queue. Restores the old `spl` level.
	Updates `RTSCHED` counts.

Routine	Purpose
save() (resume.s)	Routine called to save states. Saves the thread's process control block (pcb) marker
find_thread_my_spu() (pm_policy.c)	For the current CPU, find the most deserving thread to run and remove the old. Search starts at bestq, an index into the table of run queues. When found, set up the new thread to run. Mark the interval timer in the spu's mpinfo. Set the processor state as MPSYS. Remove the thread from its run queue. Verify that it is runnable (kt_stat== TSRUN). Set the EIRR to MPSCHED_INT_ENABLE. Set the thread context bit to TSRUNPROC to indicate the thread is running.
resume() (resume.s)	Restores the register context from pcb and transfers control to enable the thread to resume execution.

Process and Processor Interval Timing

Timing intervals are used to measure user, system, and interrupt times for threads and idle time for processors. These measurements are taken and recorded in machine cycles for maximum precision and accountability. The algorithm for interval timing is described in pm_cycles.h.

Each processor maintains its own timing state by criteria defined in struct mpinfo, found in mp.h.

Table 1-27 **Processor timing states**

Timing state	Purpose
`curstate`	The current state of the processor (`spustate_t`)
`starttime`	Start time (`CR16`) of the current interval
`prevthreadp`	Thread to attribute the current interval.
`idlecycles`	Total cycles the SPU has spent idling since boot (`cycles_t`)

Processor states are shown in the next table.

Table 1-28 **Processor states**

SPU state	Meaning
`SPUSTATE_NONE`	Processor is booting and has not yet entered another state
`SPUSTATE_IDLE`	Processor is idle.
`SPUSTATE_USER`	Processor is in user mode
`SPUSTATE_SYSTEM`	Processor is in `syscall()` or trap.

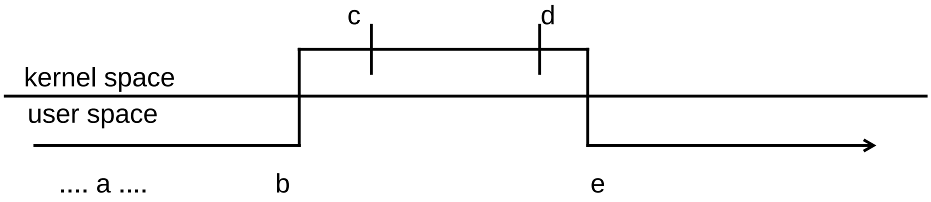
Time spent processing interrupts is attributed to the running process as user or system time, depending on the state of the process when the interrupt occurred. Each time the kernel calls wakeup() while on the interrupt stack, a new interval starts and the time of the previous interval is attributed to the running process. If the processor is idle, the interrupt time is added to the processor's idle time.

State Transitions

A thread leaves `resume()`, either from another thread or the idle loop. Protected by a lock, the routine `resume_cleanup()` notes the time, attributes the interval to the previous thread if there was one or the processor's idle time if not, marks the new interval's start time, and changes the current state to `SPUSTATE_SYSTEM`.

When the processor idles, the routine `swtch()`, protected by a currently held lock, notes the time, attributes the interval to the previous thread, marks the new interval as starting at the noted time, and changes the current state to `SPUSTATE_IDLE`.

Figure 1-39 A user process makes a system call.

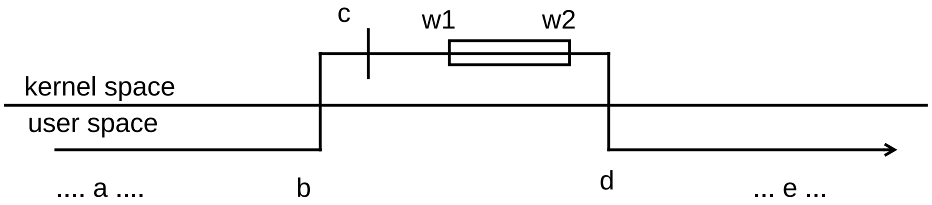


A user process running in user-mode at (a) makes a system call at (b). It returns from the system call at (e) to run again in user-mode. Between (b) and (e) it is in running in system-mode. Toward the beginning of `syscall()` at (c), a new system-mode interval starts. The previous interval is attributed to the thread as user time. Toward the end of `syscall()` at (d), a new user-mode interval starts. The previous interval is attributed to the thread as system-time.

For timing purposes, traps are handled identically, with the following exceptions:

- (c) and (d) are located in `trap()`, not `syscall()`, and
- whether or not (d) starts a user- or system-mode interval depends on the state of the thread at the time of the trap.

Figure 1-40 An interrupt occurs



Interrupts are handled much like traps, but any wakeup that occurs while on the interrupt stack (such as w1 and w2 in the figure above) starts a new interval and its time is attributed to the thread being awakened rather than the previous thread.

Interrupt time attributed to processes is stored in the kt_interrupttime field of the thread structure. Concurrent writes to this field are prevented because wakeup is the only routine (other than allocproc()) that writes to the field, and it only does so under the protection of a spinlock. Reads are performed (by pstat() and others) without locking, by using timecopy() instead.

Conceptually, the work being done is on behalf of the thread being awakened instead of the previously running thread.