A System V Compatible Implementation of 4.2BSD Job Control

David C. Lennert

Hewlett-Packard Company
Information Technology Group
hplabs!hpda!davel

ABSTRACT

This paper gives an overview of how process groups and controlling terminals are handled in System V and 4.2BSD and then discusses the effect 4.2BSD job control has on these things. A modified 4.2BSD interface is discussed which supports 4.2BSD job control functionality but in a way which allows AT&T System V compatibility. This interface has been implemented in Hewlett-Packard's UNIX† system, HP-UX.

1. INTRODUCTION

The job control functionality first introduced into UNIX by Jim Kulp of IIASA and later provided by 4.2BSD UNIX has become a *de facto* industry standard. However, this job control facility, as implemented in 4.2BSD, is incompatible in several respects with System V.

Recently a proposal was submitted to the IEEE P1003 Portable Operating System standard committee by Sun Microsystems [Harris86] which attempts to define 4.2BSD job control functionality in a way compatible with System V. Hewlett-Packard Company has been independently developing a similar proposal. HP's proposal is almost identical to Sun's but goes beyond it to address many "corner case" areas which strongly affect System V compatibility.

This paper gives an overview of the relevant areas of System V functionality which are affected. It then overviews how job control is implemented in 4.2BSD and how this impacts the System V interface. Finally, the HP-UX interface is presented and a similar overview of its implementation is given.

The various overviews cover how job control signals are generated, passed, and acknowledged by the tty driver and user processes. They also explain how process groups are established and changed.

2. FUNDAMENTALS

In the following discussion the reader is assumed to have an understanding of several fundamental concepts found in the UNIX operating system. For convenience these concepts are briefly reviewed here.

2.1. Process Groups and Controlling Terminals

Every process has a unique numeric value associated with it called its *process ID*. Every process also has a non-unique numeric value associated with it called its *process group ID*. A *process group* is a collection of processes having identical numeric process group ID's. Typically, one process in the process group will be the *process group leader*. The process group leader has a process ID which is numerically equal to the process group ID associated with all processes in the process group. Typically, the process group leader is the ancestor of all other processes in the process group.

† UNIX is a trademark of AT&T.

A process can have a *controlling terminal* which is usually the login terminal of the user who created the process. A process can obtain access to its controlling terminal by opening the file */dev/tty*. All processes in the same process group typically share the same controlling terminal. A terminal usually has a process group ID associated with it, called the *tty group ID*. When a user generates a keyboard signal (e.g., by typing the interrupt character), the tty driver sends the appropriate signal to all processes which are members of the process group indicated by the tty group ID. In summary, usually, but not necessarily, all processes in the same process group share the same controlling terminal, and the tty group ID for that terminal is equal to the process group ID of the process group.

For further explanation see [Roch85] and intro(2) in your favorite UNIX Programmer's Manual.

2.2. 4.2BSD Job Control

4.2BSD job control allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at any later point. This only easily works for processes which are stopped and continued during the same login session.

The user almost always employs this facility via the interactive interface jointly supplied by the system tty driver and a job control shell such as csh(1) or ksh(1). The tty driver recognizes a user-defined *suspend character* which causes all current foreground processes to stop and the user's job control shell to resume. The job control shell provides commands which continue stopped processes in either the foreground or background. The tty driver will also stop a background process when it attempts to read from or write to the users terminal. This allows the user to finish or suspend their foreground task without interruption and continue the stopped background process at a more convenient time.

To enable the system to support this, 4.2BSD job control introduces five new signals: SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT. The first four signals cause a process to stop unless the signals are being caught or ignored. SIGCONT always causes a stopped process to continue. (SIGCONT has no effect on processes which are not stopped.) SIGSTOP cannot be caught or ignored.

The tty driver sends some of these signals to all processes in the tty process group under the following conditions: The driver sends SIGTSTP when the user types the suspend or delayed suspend character. The driver sends SIGTTIN (SIGTTOU) when a background process attempts to read from (write to) its controlling terminal. SIGCONT is usually only sent by a job control shell when the user requests that a stopped process be continued. Of course, any signal can be sent by a user via the kill(1) command or by a program via the kill(2) system call.

It should be noted that these signals can be added to a UNIX implementation in a manner which preserves source and object code compatibility. A process is not required to be aware of them. By default the signals do "the right thing."

For further information see [Joy80] and [UCB83].

3. AT&T SYSTEM V

3.1. Introduction

System V process groups closely resemble the concept of a login session. That is, all processes spawned during the same login session tend to belong to the same process group, and keyboard signals are typically sent to all processes spawned from the login session.

3.2. System V Process Group Handling

In System V, the only way to alter the process group associated with a process (p_pgrp) is via setpgrp(2). And this can only set the process group to equal the process ID (pid) of the process. When this happens the resulting process with pid = p_pgrp is called a process group leader. Since a process's pid can never change, once a process issues a setpgrp(2) call it irrevocably becomes a process group leader.

The init(1M) process spawns all other processes on the system either directly or indirectly. Before directly spawning a process (after the fork(2) but before the exec(2)), init calls setpgrp(2). Thus all original children (not orphans) of init are forced to (irrevocably) be process group leaders.

When a new process is created, it is assigned a new pid but it inherits the process group number of its parent. Thus child processes are, by default, not process group leaders (although they can become a process group leader via setpgrp(2)).

When a process group leader which has a controlling terminal (see below) terminates, SIGHUP is sent to all processes in the same process group.

Further, when a process group leader terminates, all processes that belong to this process group are altered to belong to no process group (their p_pgrp is set to zero). More precisely, when any process exits, all processes whose process group (p_pgrp) equals the pid of the terminating process will have their p_pgrp set to zero; this check succeeds only in the case of a terminating process group leader.

3.3. System V Controlling Terminals

A terminal that is currently open by a process may also be a "controlling terminal" for a process group. When certain control characters are typed on a controlling terminal, signals are sent by the terminal driver to all processes that belong to the process group associated with the terminal.

When a process becomes a process group leader (via setpgrp(2)) it automatically loses its controlling terminal. After this, the first terminal (that is not already a controlling terminal) opened by the process is assigned to be the controlling terminal for that process. Also, the process group associated with that terminal (t_pgrp, also known as the tty group ID) is set equal to the process group associated with the process group leader (p_pgrp). All child processes inherit the controlling terminal and process group of their parent.

More precisely, in System V, the process group associated with a terminal (t_pgrp), can be changed in the following ways:

(1) When a terminal is opened by a process group leader (pid == p_pgrp) that does not already have a controlling terminal, it becomes the controlling terminal for that process group (t_pgrp is set equal to p_pgrp) if it is not already a controlling terminal.

(2) When a process group leader (pid == p_pgrp) dies, if it has a controlling terminal that is associated with the same process group (t_pgrp == p_pgrp), then that terminal is disassociated from that process group (t_pgrp is set to zero).

(3) When the last process to have a terminal open closes that terminal, the terminal is disassociated from its process group (t_pgrp is set to zero).

3.4. System V Typical Scenario

This is a typical scenario for the birth and death of a process group and its controlling terminal.

The init(1M) process wants to enable a terminal for login. It calls fork(2) to create a new process and then calls setpgrp(2) to make the process a process group leader which also removes the process's controlling terminal. It then runs the getty(1M) program as the process via exec(2). Getty opens the terminal causing it to become getty's controlling terminal and be associated with getty's process group (t_pgrp is set to p_pgrp). Getty replaces itself with login(1) which replaces itself with a login shell, e.g., sh(1). Usually no program calls setpgrp(2) and thus all descendent processes of the login shell are in the same process group and have the same controlling terminal; keyboard signals are sent to all processes launched during this session.

When a logout occurs, the login shell (which is the process group leader) dies and the controlling terminal is freed up (t_pgrp is set to zero) so that it can be claimed as a controlling terminal by a subsequent getty respawned by init. SIGHUP is sent to all processes in the same process group. The process group (p_pgrp) of all descendent processes is then set to zero.

Note that there may continue to be background processes (previously started by the now defunct login shell) which continue to execute but keyboard signals will no longer be sent to these processes (since both t_pgrp and p_pgrp equal zero).

4. 4.2BSD

4.1. Introduction

4.2BSD process groups closely resemble the concept of a task within a login session, where a task represents a set of processes which are affected as a group by job control operations. Every time a job control shell (e.g., csh) spawns either a foreground or background command, all processes in the pipeline (and their descendents) are placed in their own unique process group with the first command in the pipeline being the process group leader.

A task is in the foreground when the process group associated with the controlling terminal for the task (t_pgrp) is equal to the process group associated with the processes in the task (p_pgrp). Otherwise the task is in the background. A job control shell moves a job between the foreground and background by adjusting the terminal process group (t_pgrp) of the controlling terminal.

Note that 4.2BSD forms new process groups with process group leaders much more often than System V usually does (every command versus every login).

4.2. 4.2BSD Process Group Handling

In 4.2BSD, the process group associated with a process (p_pgrp) can be altered in two ways. The first is via setpgrp(2). 4.2BSD's setpgrp(2) is analogous to System V's setpgrp(2) except that the former can affect processes other than the current process and can cause the affected process to adopt a process group other than that process's process ID (pid). Thus, unlike System V, a process can cease to be a process group leader.

In addition to setpgrp(2), a process that is not a member of any process group (p_pgrp == 0) will "inherit" or join the process group associated with its controlling terminal at the time the process is assigned a controlling terminal during open(2). If the terminal being opened is not presently the controlling terminal for any process group, then the process opening the terminal will first be made a process group leader (p_pgrp will be set to p_pid) and then the terminal will become the controlling terminal for this new process group. All this is done by the tty open code.

When a new process is created it inherits the process group of its parent.

Unlike System V init(1M), 4.2BSD init(8) does not call setpgrp(2) when spawning other processes. All processes spawned by init inherit init's process group which happens to be zero ("not a member of any process group"). This is actually crucial for assigning controlling terminals; see below.

4.3. 4.2BSD Controlling Terminals

Unlike System V, a 4.2BSD process does not lose its controlling terminal when altering its process group (via setpgrp(2)). Also unlike System V, a 4.2BSD process that is a process group leader (pid == p_pgrp) but which has no controlling terminal does not receive a controlling terminal when opening a new terminal.

A process can obtain a controlling terminal under 4.2BSD in only the following ways:

(1) A process can inherit a controlling terminal from its parent.

(2) A process that is not a member of any process group (p_pgrp == 0) can open any terminal and that terminal will become its controlling terminal (whether or not it is already the controlling terminal for another process). However, this can happen in one of two ways:

    If the terminal is not already a controlling terminal (t_pgrp == 0) then the opening process becomes a process group leader (its p_pgrp is set equal to its pid) and the terminal becomes its controlling terminal (t_pgrp is set to the new p_pgrp value).

    If the terminal is already a controlling terminal for another process (t_pgrp is not zero) then the opening process joins the process group already associated with the controlling terminal. That is, p_pgrp is set equal to the current t_pgrp. Note that the opening process does not become a process group leader, i.e., p_pgrp is not equal to its pid.

Note that this procedure only happens during the first terminal open for a process that was either originally spawned by init or whose ancestor processes (all the way back to init) never altered their process group (p_pgrp) either by opening a terminal or calling setpgrp(2).

A terminal ceases to be a controlling terminal (t_pgrp is set to zero) under 4.2BSD in the following way:

(1) When the last process to have a terminal open closes that terminal then the terminal is disassociated from its process group (t_pgrp is set to zero).

There are two other facilities unique to 4.2BSD which affect access to control terminals: the TIOCSPGRP ioctl(2) and vhangup(2).

The TIOCSPGRP ioctl(2) function changes a terminal's process group (t_pgrp) to any desired value. It is typically used by csh(1) to control which set of processes (process group) is in the foreground.

The vhangup(2) function is invoked by init after forking but before exec'ing getty. This function removes read and write permission for all processes (including the caller) that have the controlling terminal open (whether or not it is their controlling terminal). It then sends SIGHUP to the process group associated with the terminal (t_pgrp). The latter action is similar to the System V functionality that sends SIGHUP to a process group on death of the process group leader; 4.2BSD does not do this on the death of a process group leader.

4.4. 4.2BSD Typical Scenario

This is a typical scenario for the birth and death of a login, its controlling terminal, and process groups associated with a job.

The init(8) process wants to enable a terminal for login. First it creates a new process via fork(2). Then it opens the terminal which (because the p_pgrp inherited from init is zero) causes it to become the controlling terminal for this process and either alters the process group (p_pgrp) of the process to match the terminal process group (t_pgrp) if non-zero, or alters both p_pgrp and t_pgrp to equal the process ID (pid) if t_pgrp is zero. At this point the new process has a controlling terminal whose process group (t_pgrp) is equal to the process's process group (p_pgrp). However, the process may not be a process group leader (i.e., p_pgrp may not equal pid).

Now the new process calls vhangup(2) to remove access permissions for the controlling terminal from all processes (as well as sending SIGHUP to any processes in the process group previously associated with the terminal). The new process then reopens the terminal to get a file descriptor with read and write permissions since the vhangup(2) removed these permission from the file descriptor returned by the previous open. The previous file descriptor is not closed until now to prevent losing the controlling terminal; (remember that p_pgrp for the new process is no longer zero.)

The new process now replaces itself with getty(8) which replaces itself with login(1) which replaces itself with a login shell, e.g., csh(1). Csh now begins to manipulate the process group associated with the terminal (t_pgrp) via the TIOCSPGRP and TIOCGPGRP ioctl(2) calls and the process group associated with its child processes (p_pgrp) via setpgrp(2) in order to allow job control. This happens (briefly) in the following way:

Csh launches a pipeline by making all programs in the pipeline be immediate descendents of csh. (This is different from sh which makes all programs in the pipeline except the last be descendents of the last program in the pipeline.) All programs in the pipeline belong to the same process group (not the same as csh's process group) and the first program in the pipeline is the process group leader (its pid is equal to the process group for the pipeline). If the pipeline is being launched in the foreground (or moved to the foreground) then the process group associated with the terminal (t_pgrp) is set to the process group of the pipeline.

When a logout occurs, the login shell dies. Any pending SIGTTIN, SIGTTOU, and SIGTSTP signals are cleared for all descendent processes. All immediate child processes are inherited as orphans by init; if any are currently stopped then they are killed (SIGKILL). If the exiting process is the last process that has the controlling terminal open then the terminal's process group (t_pgrp) is set to zero, otherwise it is left alone. Nothing special is done for process group leaders; in fact, login shells are frequently not process group leaders. (SIGHUP is not sent and the controlling terminal is not necessarily cleared.)

Note that there may continue to be processes (previously started by the now defunct login shell) which continue to execute. And that keyboard signals can still be sent to these processes under some circumstances (specifically when the processes were in the foreground (p_pgrp == t_pgrp) when the login shell died; this usually only happens when the login shell is killed from another terminal via kill(1).) Note also that this continues to be true even after a new session logs in on the same terminal since the new login shell joins the process group which is already associated with the terminal from the prior login.

4.5. Job Control Signal Handling

The following discussions concerning signals and kernel process synchronization are similar to ones found in [Thom78], [Ritch79], and [Bach79].

4.5.1. Basic Overview

Usually a process is either running or sleeping waiting for an event to occur (e.g., I/O completion). When a signal is sent to a process (either by another process or an I/O driver) what actually occurs is that a flag is set for the receiving (or target) process indicating that the signal has been sent and the target process performs the actual signal operation to itself the next time it runs. Thus sending a signal amounts to requesting the target process to itself perform a particular action. If the target process is already running it is interrupted to process the signal. If it is runnable but not currently running then the system merely waits for it to become the currently running process at which point the signal is acknowledged. If the target process is sleeping then either it is moved into a runnable state (if it is sleeping at an "interruptable" priority) or it is left sleeping (at an "uninterruptable" priority) and the signal is not acknowledged until the slept on event occurs.

The kernel procedure which sends a signal is psignal() and is executed by the sending process or driver. Psignal() updates a list of pending signals for the receiving process. If the receiving process is the currently running process and it is executing in kernel mode then the pending signal is acknowledged when the current system call completes. (This is the case where the sending process and the receiving process are the same.) If the receiving process is the currently running process and it is executing in user mode then a special event is generated which causes the process to enter the kernel and acknowledge the pending signal. (This is the case where the sending "process" is really an interrupt handler which, for example, is servicing an interrupt character typed on a user's terminal.) If the receiving process is sleeping but not holding off signals then it is set running via wakeup(); the pending signal is acknowledged as soon as the receiving process executes. If the receiving process is suspended in a sleep state that holds off signals ("sleeping uninterruptably") then it is left sleeping; the pending signal will be acknowledged after the waited for event occurs.

The procedure which tests for a pending signal is issig() and is executed by the receiving process. Issig() is executed whenever the receiving process changes from kernel mode to user mode execution; for example, at the completion of a system call. It is also executed whenever the receiving process is awakened from being suspended in a sleep state that does not hold off signals ("sleeping interruptably").

The procedure which performs the requested signal operation (e.g., invoking a signal handler or killing the process) is psig() and is executed by the receiving process if issig() returns true.

This basic structure is essentially the same in System V, 4.2BSD, and HP-UX. However, under 4.2BSD-style job control, these general principles can work slightly differently:

When processing stop signals, the psignal() function, called by the sending process, actually stops the target process sometimes. In these cases, the target process never realizes that it received the signal or that it stopped. However, in other cases, psignal() performs the usual process of setting the flag (p_sig) requesting that the target process stop itself the next time it runs.

The issig() function, called by the target process, can actually stop the target process.

The psig() function is only called in the case where a user handler has been provided for the job control signal.

A more complete description of job control signal handling is contained in the pseudocode below.

4.5.2. psignal()

To send SIGCONT to a target process:

```
sending SIGCONT clears any pending stop signals;
if the target process is STOPPED but is also SLEEPING (p_wchan != 0)
        merely continue the process's SLEEP;

else if the target process is STOPPED and is NOT SLEEPING (p_wchan == 0)
        set the process RUNNING;
```

To send a stop signal (SIGTSTP, SIGTTIN, SIGTTOU, SIGSTOP) to a target process:

```
sending a stop signal clears any pending SIGCONT;

if the target process is RUNNABLE or RUNNING
        note the pending signal in p_sig;

else if the target process is SLEEPING NON-interruptably
        note the pending signal in p_sig;

else if the target process is SLEEPING interruptably
      and IS catching the signal
        note the pending signal in p_sig and
        wakeup the process from its sleep;

else if the target process is SLEEPING interruptably
      and is NOT catching the signal
        stop the process by setting its state to SSTOP
        but leave it sleeping on its p_wchan;
        send SIGCLD to parent (if it expects BSD-style)
```

General note: sending a stop signal (other than SIGSTOP) to a child of init causes the target process to be killed.

4.5.3. issig()

Issig() is called in all cases except where the process was sleeping interruptably and was not catching the signal.

To acknowledge a pending SIGCONT or stop signal:

```
if in the middle of a VFORK
        hold off all stop signals (pretend they don't exist yet)

else if catching the signal
        return a request to invoke user signal handler via psig()

else if SIGCONT
        do nothing  /* pretend it doesn't exist */
```

```
else /* stop signals */
        stop the process by setting its state to SSTOP
        send SIGCLD to parent (if it expects BSD-style)
        call swtch() to dispatch another process
```

General note: sending a stop signal (other than SIGSTOP) to a child of init causes the target process to be killed.

4.5.4. psig()

Psig() is called whenever issig() returns an indication that a user handler is defined for a job control signal. Psig() merely invokes the user signal handler.

4.5.5. wakeup()

The fact that a process is sleeping (waiting for an event to occur) is indicated by two process state values: p_wchan is non-zero, indicating the event being waited for, and the process state is SSLEEP.

Wakeup() usually causes all processes waiting (sleeping) on a specified event to be awakened. When a process is awakened two things happen: The process is removed from the sleep queue (p_wchan is cleared) and it is added to the run queue.

If, however, wakeup() discovers a process whose p_wchan matches the specified event but whose process state is SSTOP (stopped) then the process is removed from the sleep queue (indicating that the waited for even has happened) but it is not placed on the run queue. A subsequent SIGCONT will cause it to be placed on the run queue.

Thus it is possible to have a process which is both sleeping (p_wchan non-zero) and stopped (process state is SSTOP rather than SSLEEP).

4.5.6. Signal Setup via init

When init(8) launches any process it causes the process to ignore all the job control stop signals (SIGTSTP, SIGTTIN, & SIGTTOU). This allows login shells which are not job control shells to automatically ignore the signals. Further, all descendent processes of such a login shell will also ignore these signals unless they explicitly enable them.

4.6. Foreground/Background Processes

4.6.1. Basic Overview

4.2BSD job control supports the notion of a process being in the *foreground* or *background.* The distinction is a background process is usually forced to stop when it attempts to perform I/O (including most control operations) on its controlling terminal, while a foreground process is not hindered.

Specifically, when a background process attempts to read from its controlling terminal it is sent the SIGTTIN signal which, by default, causes it to stop. When it attempts to write to its controlling terminal and LTOSTOP has been enabled for the terminal, then the process is sent the SIGTTOU signal which, by default, causes it to stop. If, however, a background process has chosen to catch the signal, the specified user handler is invoked. If the process is ignoring or masking the stop signal(s), then the terminal I/O request returns an I/O error, EIO.

A background process is one whose process group (p_pgrp) is not equal to the process group of its controlling terminal (t_pgrp) (and t_pgrp is not zero). All other processes (including ones doing I/O to terminals that are not their controlling terminals) are considered to be in the foreground.

4.6.2. Tty Driver Provisions

To distinguish between foreground and background programs the tty driver must perform checks on attempted I/O operations to a process's controlling terminal. This is done in several places.

At the beginning of a read/write system call the tty driver checks to see if the calling process is in the background. If it is, then all processes in the process group of the calling process are sent the appropriate signal (SIGTTIN or SIGTTOU) unless the signal is masked or ignored by the calling process. In this case the driver returns the EIO error. After the tty driver sends the signal, the calling process is put to sleep waiting for the *lightning bolt event*†. This allows the calling process to receive the signal (and usually stop). When the process returns from the sleep (usually by being continued) the tty driver repeats the foreground/background check before proceeding with the operation.

When the process is in the foreground, the I/O operation proceeds. In the case of a terminal read this usually results in the process being put to sleep to wait for input characters to arrive. At this point the user could type their suspend character (e.g., ˆZ). This causes the interrupt portion of the tty driver to send SIGTSTP to the controlling terminal's process group (i.e., all processes which are in the foreground). In our scenario this would include the process sleeping on terminal input, and this would typically cause it to stop.

When a sleeping process is stopped it is also left sleeping as well. If, in this case, tty input characters subsequently arrived then the process would be awakened. However, because it is also stopped, it would not be set running; it would merely be "unslept".

At some later time the process would be continued (e.g., via a csh "fg" or "bg" command which sends SIGCONT). If the process had not been previously unslept it would merely continue its sleeping; it would receive no indication that it had stopped and continued. If the process had been previously unslept it would now be set running.

When the process is set running it resumes execution in the tty driver. Because a (potentially substantial) amount of time has elapsed and because the process may have been stopped and restarted, the tty driver is no longer sure whether this process is still in the foreground. So before checking if input characters are available, the tty driver rechecks whether the process is in the foreground or background. This is necessary because, in our scenario, the stopped process could have been continued in the background (via csh "bg"). To check this the tty driver merely repeats the foreground/background check it made at the beginning of the system call.

5. SYSTEM V INCOMPATIBILITIES AND THEIR RESOLUTIONS

Job control as implemented in 4.2BSD is incompatible with System V semantics in some significant respects. This section discusses each of these incompatibilities and the resolution implemented in HP-UX to maintain System V compatibility.

The system interface needed to support 4.2BSD-style job control, tailored for System V compatibility as discussed in this section, is presented in the form of manual page excerpts in [Len86].

5.1. Setpgrp(2) Changes

Because the needed semantics of 4.2BSD setpgrp(2) conflict with the semantics of System V setpgrp(2), the 4.2BSD setpgrp(2) function was renamed to be setpgrp2(2). (The choice of new name is arbitrary; setpgrp2 was chosen in the same spirit as 4.2BSD's wait3(2).)

5.2. SIGHUP Changes

System V semantics state that when a process group leader dies, all processes in the same process group are sent the SIGHUP signal which, by default, kills all the processes.

Job control shells execute a command by making all processes in the pipeline belong to the same (brand new) process group and by making the first program in the pipeline be the process group leader. Typically, the first program in a pipeline terminates before the other programs. Under System V semantics, this would cause the premature death of the remaining pipeline. Because of this, 4.2BSD does not generate SIGHUP on process group leader death.

† The lightning bolt event is a standard UNIX event which occurs frequently, for example, every second.

In order to support System V semantics and still allow job control to function properly, HP-UX makes a distinction between a "System V process group leader" and a "job control process group leader". A System V process group leader is given System V semantics (SIGHUP is generated) and a job control process group leader is given 4.2BSD semantics (SIGHUP is not generated). A process which becomes a process group leader via setpgrp(2) is considered to be a System V process group leader. A process which becomes a process group leader via setpgrp2(2) is considered to be a job control process group leader. Since the HP-UX (and System V) init(1M) program calls setpgrp(2) on behalf of all processes it spawns, all login shells start out as System V process group leaders. A process must explicitly call setpgrp2(2) to deviate from the System V semantics.

5.3. SIGCLD Changes

Under System V, SIGCLD is sent to a process whenever one of its immediate child processes dies. Under 4.2BSD, SIGCLD (or its variant, SIGCHLD) is also generated when a process changes state from running to stopped. Since a System V application would not expect to receive SIGCLD under these new circumstances and since a job control shell would not be able to function properly without such notification, a compatible compromise was developed.

The (parent) process wishing to trap SIGCLD may set a flag when calling the HP-UX sigvector(2)† routine to establish a signal handler. This flag will cause SIGCLD to be sent for stopped children, in addition to terminated children. A System V application using signal(2) will see the System V compatible SIGCLD semantics.

5.4. Controlling Terminal Changes

Under System V, whenever a process group leader dies, the controlling terminal associated with that process group (if any) is deallocated (disassociated from that process group). 4.2BSD does not deallocate controlling terminals on process group leader death for the following reason: Job control shells make the lead process in every pipeline a process group leader. If the controlling terminal for each pipeline were deallocated whenever the lead process terminated, then the remaining processes would effectively become background processes (assuming they were currently in the foreground) and would stop when any of them attempted subsequent I/O to the terminal.

To allow both semantics, controlling terminals are only deallocated when a "System V process group leader" dies and not when a "job control process group leader" dies. (See the discussion of SIGHUP changes above.)

However, this change leads to the following problem: In order for a terminal to be allocated as a controlling terminal for a new login, it must be deallocated when the previous login terminates. System V relies on process group leader death to deallocate controlling terminals (since all login shells are forced to be process group leaders by init(1M)). This is no longer reliable since login shells could become "job control process group leaders". Further, not all logins are spawned directly by init(1M); the 4.2BSD rlogin facility is a prime example. 4.2BSD solves this problem by allowing a new login to join the process group of the controlling terminal which is still allocated from the previous login. However this violates System V compatibility.

The solution chosen was to mark a process that causes a controlling terminal to be allocated and to deallocate the controlling terminal whenever that process terminates. This reliably catches logins which are spawned either directly or indirectly from init(1M), whether they are "System V process group leaders" or not. Controlling terminals continue to be deallocated on death of System V process group leaders using the System V semantics.

---

† Sigvector(2) is an HP-UX extension proposed to the IEEE P1003 [Head85] which supports both the reliable signal operations of 4.2BSD sigvec(2) and the conventional signal operations of System V signal(2). In HP-UX, signal(2) is implemented as a library using sigvector(2). Note that the changes proposed here to sigvector(2) can be identically made to 4.2BSD sigvec(2).

5.5. Security

Several security holes exist in the 4.2BSD process group altering mechanisms. To plug these holes the following changes were made.

4.2BSD setpgrp(2) allows a process to alter the process group associated with another process to any value. 4.2BSD restricts this operation so that the affected process must pass the same security restrictions enforced when sending signals, or must be a descendent of the calling process. However, this still allows a process to join a process group already associated with another user. To tighten this security, setpgrp2(2) was further restricted such that if the specified new process group value is equal to the process ID (pid) or process group ID of any existing processes, then all such processes must pass the above security restrictions.

Similarly, the 4.2BSD TIOCSPGRP ioctl(2) allows a terminal's process group to be altered to any value. This allows a user's terminal to easily become an additional "controlling terminal" for another user's process group; keyboard signals can be sent to the other user's processes, thus bypassing the security enforced by kill(2). Because of this, the TIOCSPGRP ioctl(2) was altered to enforce similar security restrictions as setpgrp2(2).

In System V and 4.2BSD, a process can obtain access to its controlling terminal by opening the file */dev/tty*. Under System V, processes left executing after a user's logout are allowed further access to */dev/tty* until the terminal it represents is reallocated as a controlling terminal for a new login. More specifically, */dev/tty* access is allowed whenever the process group ID of the leftover process matches the process group ID of the terminal. These IDs continue to match immediately after logout (since both have been zeroed) until the terminal is re-enabled for login by getty(1M). (Note that when the new login terminates, */dev/tty* access is restored again to these prior processes because the controlling terminal's process group ID is re-zeroed.) Further, if a process has its controlling terminal opened directly (not via the */dev/tty* synonym) then access is not restricted at all after logout. These System V semantics can constitute security problems. However, they are not explicitly required by the System V Interface Definition [ATT86].

4.2BSD does nothing to hamper */dev/tty* access for processes remaining after logout. The process group ID for the controlling terminal is not altered, and, in fact, it is preserved even into the next login (since subsequent logins join the already existing process group associated with the terminal, if any). These semantics also represent security problems. However, 4.2BSD does prohibit access to the controlling terminal if it is opened directly; this is accomplished when init(8) issues the vhangup(2) system call.

Although preserving the System V semantics for controlling terminal access after logout is not deemed necessary or even recommended, it is easy to do in the following way. Whenever a process that allocated a controlling terminal dies, all processes which share this controlling terminal have their process group ID zeroed. This is analogous to, and occurs in addition to, the System V behavior of zeroing the process group ID for all related processes when their process group leader dies. */dev/tty* checks similar to System V can then be employed.

5.6. TTY Driver Considerations

For System V compatibility, the suspend and delayed suspend characters are defaulted to a disabled value (0377). This means that job control is "inactive" by default when a user logs on. The user must explicitly activate job control by defining either or both of these characters via stty(1) or some similar interface.

There should be no problem allowing 4.2BSD-style job control, as modified here, to co-exist with System V's shell layers job control system. (See shl(1) and sxt(7) in the System V Release 2 reference manuals.)

6. HP-UX

6.1. Introduction

HP-UX process groups are used in two major ways.

System V process groups closely resemble the concept of a login session. That is, all processes spawned during the same login session tend to belong to the same process group, and keyboard signals are typically sent to all processes spawned from the login session.

Job control process groups closely resemble the concept of a task within a login session, where a task represents a set of processes which are affected as a group by job control operations. Every time a job control shell (e.g., csh) spawns either a foreground or background command, all processes in the pipeline (and their descendents) are placed in their own unique process group with the first command in the pipeline being the process group leader.

A task is in the foreground when the process group associated with the controlling terminal for the task (t_pgrp) is equal to the process group associated with the processes in the task (p_pgrp). Otherwise the task is in the background. A job control shell moves a job between the foreground and background by adjusting the terminal process group (t_pgrp) of the controlling terminal.

Note that a job control shell forms new process groups with process group leaders much more often than a non-job control (System V) shell usually does (every command versus every login).

6.2. HP-UX Process Group Handling

In HP-UX, the process group associated with a process (p_pgrp) can be altered via setpgrp(2) or setpgrp2(2).

As in System V, setpgrp(2) can only set the process group to equal the process ID (pid) of the process. When this happens, the resulting process with pid = p_pgrp is called a System V process group leader.

Setpgrp2(2) is analogous to setpgrp(2) except that it can affect processes other than the current process and can cause the affected process to adopt a process group other than that process's process ID (pid). Setpgrp2(2) also forms job control process groups rather than System V process groups. Using setpgrp2(2), the calling process, or certain other processes, can either become a job control process group leader or can cease to be a process group leader.

Because job control process groups are handled slightly differently by HP-UX than System V process groups, HP-UX marks processes that are job control process group leaders (i.e., that have called setpgrp2(2) without subsequently calling setpgrp(2)).

The init(1M) process spawns all other processes on the system either directly or indirectly. Before directly spawning a process (after the fork(2) but before the exec(2)), init calls setpgrp(2). Thus all original children (not orphans) of init are forced to start out as System V process group leaders.

When a new process is created, it is assigned a new pid but it inherits the process group number of its parent. Thus child processes are, by default, not process group leaders (although they can become a process group leader via either setpgrp(2) or setpgrp2(2)).

When a System V process group leader that has a controlling terminal (see below) terminates, SIGHUP is sent to all processes in the same process group. Further, when a System V process group leader terminates, all processes which belong to this process group are altered to belong to no process group (their p_pgrp is set to zero).

Also, whenever any process that allocated a controlling terminal terminates, all processes that share this controlling terminal are altered to belong to no process group (their p_pgrp is set to zero).

When any process exits, any pending SIGTTIN, SIGTTOU, and SIGTSTP signals are cleared from all descendent processes (not just immediate children).

6.3. HP-UX Controlling Terminals

A terminal that is currently open by a process may also be a "controlling terminal" for a process group (collection of processes). When certain control characters are typed on a controlling terminal, signals are sent by the terminal driver to all processes which belong to the process group associated with the terminal. These include the job control suspend and delayed suspend characters.

Controlling terminals also play a role in determining whether a process is in the foreground or background. See FOREGROUND/BACKGROUND PROCESSES above.

When a process becomes a System V process group leader (via setpgrp(2)) it automatically loses its controlling terminal. (This does not happen for a job control process group leader, i.e. when calling setpgrp2(2).) After this, the first terminal (that is not already a controlling terminal) opened by a process that is a (System V or job control) process group leader is assigned to be the controlling terminal for that process; also the process group associated with that terminal (t_pgrp) is set equal to the process group associated with the process group leader process (p_pgrp). All child processes inherit the controlling terminal and process group of their parent.

More precisely, in HP-UX, the process group associated with a terminal (t_pgrp), can be changed in the following ways:

(1) When a terminal is opened by a System V or job control process group leader (pid == p_pgrp) that does not already have a controlling terminal, it becomes the controlling terminal for that process group (t_pgrp is set equal to p_pgrp) if it is not already a controlling terminal.

(2) When a System V process group leader (pid == p_pgrp) dies, if it has a controlling terminal that is associated with the same process group (t_pgrp == p_pgrp), that terminal is disassociated from that process group (t_pgrp is set to zero).

(3) When any process dies which originally caused a controlling terminal to be created (see (1) above), if it still has a controlling terminal, that terminal is disassociated from its process group (t_pgrp is set to zero).

(4) When the last process to have a terminal open closes that terminal, the terminal is disassociated from its process group (t_pgrp is set to zero).

(5) The TIOCSPGRP ioctl(2) call can explicitly change a terminal's process group (t_pgrp) to any value within certain security restrictions. This is used by a job control shell to change which set of processes (process group) is in the foreground.

6.4. HP-UX Typical Scenario

This is a typical scenario for the birth and death of a login, its controlling terminal, and the process groups associated with a job.

The init(1M) process wants to enable a terminal for login. It creates a new process via fork(2) and calls setpgrp(2) to make it a System V process group leader which also removes its controlling terminal. Init then runs the getty(1M) program as the process via exec(2). Getty opens the terminal causing the terminal to become getty's controlling terminal and be associated with getty's process group (t_pgrp is set to p_pgrp). As a side effect, this process is now marked as having created a controlling terminal; when it dies the controlling terminal will be freed for re-use. Getty replaces itself with login(1) which replaces itself with a login shell.

At this point one of two scenarios typically takes place. The login shell is either a job control shell (e.g., csh(1)) or it is not (e.g., sh(1)).

If the login shell is not a job control shell then things proceed much as they do on System V. Usually no program calls setpgrp(2) or setpgrp2(2) and thus all descendent processes of the login shell are in the same process group and have the same controlling terminal; keyboard signals are sent to all processes launched during this session.

If the login shell is a job control shell, then job control operations are performed. Csh begins to manipulate the process group associated with the terminal (t_pgrp) via the TIOCSPGRP and TIOCGPGRP ioctl(2) calls and the process group associated with its child processes (p_pgrp) via setpgrp2(2) in order to allow job control. This happens (briefly) in the following way:

Csh launches a pipeline by making all programs in the pipeline be immediate descendents of csh. (This is different from sh which makes all programs in the pipeline except the last be descendents of the last program in the pipeline.) All programs in the pipeline belong to the same process group (not the same as csh's process group) and the first program in the pipeline is the process group leader (its pid is equal to the process group for the pipeline). This process is specially marked as a job control process group leader since it was established via setpgrp2(2); this basically prevents SIGHUP from being sent to the pipeline when the lead process dies. If the pipeline is being launched in the foreground (or moved to the foreground) then the process group associated with the terminal (t_pgrp) is set to the process group of the pipeline via the TIOCSPGRP ioctl(2).

When a logout occurs, the login shell dies. Any pending SIGTTIN, SIGTTOU, and SIGTSTP signals are cleared for all descendent processes. All immediate child processes are inherited as orphans by init; if any are currently stopped then they are killed (SIGKILL). Since the login shell (actually the getty before it was overlaid) created a controlling terminal, the controlling terminal is now freed (t_pgrp is set to zero) so that it can be claimed as a controlling terminal by a subsequent getty respawned by init; also, all processes which share this controlling terminal have their process group (p_pgrp) set to zero.

When a logout occurs and the login shell is a System V process group leader, SIGHUP is sent to all processes in the same process group, and the process group (p_pgrp) of all descendent processes is set to zero.

Note that there may continue to be background processes (previously started by the now defunct login shell) which continue to execute but keyboard signals will no longer be sent to these processes (since both t_pgrp and p_pgrp equal zero).

7. ACKNOWLEDGEMENTS

The following people from Hewlett-Packard contributed to the interface design and implementation of job control for HP-UX: Jim Barton, Dave Decot, Larry Dwyer, Jeff Glasson, Rita Hanson, Stephen Hares, Steve Head, Bob Lenk, John Marvin, Dave Mears, Peter Notess, Arn Schaeffer, Eviatar Shafrir.

Guy Harris from Sun Microsystems made many substantive comments and suggestions which contributed to the interface design and to this paper.

8. REFERENCES

[ATT86] *System V Interface Definition*, Issue 2, AT&T, 1986.

[Bach84] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Operating Systems", *AT&T Bell Lab. Tech. J.*, **63,** No. 8 (October 1984), pp. 1733-1749.

[Head85] Stephen Head and Donn Terry, "Reliable Signals Proposal", IEEE P1003 Proposal #P.042, Hewlett-Packard Co., September 11, 1985.

[Joy80] William Joy, "An Introduction to the C Shell", Computer Science Division, University of California at Berkeley, November 1980.

[Len86] David Lennert, Guy Harris, et. al., "System V Compatible BSD-style Job Control Facilities", IEEE P1003 Proposal #P.047, Hewlett-Packard Co. & Sun Microsystems, April 9, 1986.

[Ritch79] Dennis M. Ritchie, "The UNIX I/O System", *UNIX Programmer's Manual*, Seventh Edition, Volume 2b, Bell Telephone Laboratories, Murray Hill, NJ, January 1979.

[Roch85] Marc J. Rochkind, *Advanced UNIX Programming*, Englewood Cliffs, N.J.: Prentice-Hall, 1985.

[Harris86] Guy Harris, "Notes on Signal, Terminal Interface, and User/Group ID Handling Proposals", IEEE P1003 Proposal #P.045, Sun Microsystems, January 11, 1986.

[Thom78] K. Thompson, "UNIX Implementation", *Bell System Tech. J.*, **57,** No. 6 (July - August 1978), pp. 1931-1946.

[UCB83] *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, August 1983.

APPENDIX

JOB CONTROL MANUAL PAGE EXCERPTS

The following pages contain the UNIX manual pages which are effected by adding a System V compatible implementation of 4.2BSD job control. Note that each manual page generally contains only that portion of text which differs from the System V manual page of the same name. Sometimes, unchanged text is provided for locality reference. Changed text lines are flagged with change bars.