The ‘‘Session Tty’’ Manager

S.M. Bellovin
ulysses!smb

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

In many UNIX® systems, it is possible for a program to retain access to the login terminal after the user has logged out. This poses obvious security risks and can also confuse the modem control signals. We solve this for System V by adding a layer of indirection known as the *session tty* driver. At login time, a session device is linked to the physical terminal. User programs have access to the session device only, and may not open the physical line. Upon logout or carrier drop, the link is severed. New login sessions are given new session devices, and are thus insulated from persistent processes. Use of session devices is controlled by a new system process known as the *session manager*; by means of suitable plumbing primitives, a ‘‘reconnect after line drop’’ facility can easily be implemented.

1. INTRODUCTION

‘‘Any software problem can be solved
by adding another layer of indirection.’’

When a user logs on to a UNIX® system, a shell is fired up with file descriptors 0, 1, and 2 connected to the physical tty device used. All commands executed by that user are descendants of the shell, and normally use the same tty device for input and output. These associations — so fundamental to the design of the system — cause trouble under certain circumstances. These problems may be solved by adding a level of indirection called the ‘‘session manager’’.

Before we discuss the solution, it is, of course, helpful to know what problems it claims to cure. There are several, mostly having to do with ending conditions:

1. When the physical device receives an external close indication — i.e., when carrier has dropped, the user’s terminal has been turned off or disconnected, or a network close request is received — this fact must be propagated to all processes using that terminal. It is not sufficient to send `SIGHUP`. Some processes may have the signal ignored; others may not have the device as their controlling tty, and hence will not receive the signal at all. In either case, the result is the same: the next user of the line can receive some strange and wondrous garbage.
2. When the login shell — the process group leader — exits, the ‘‘session’’ (whatever that is) should end. Access rights to the terminal by any child processes should be revoked, regardless of whether the physical connection has been broken. If we are dealing with a network connection, such a disassociation may be needed to complete the protocol `close` processing. The system currently attempts to deal with this by sending `SIGHUP` when a process group leader ends; as noted, this is often insufficient.
3. When the host signals external equipment (i.e., modems) that a terminal session has ended, typically by dropping DTR, it must then immediately spawn a new `getty` process, and re-enable DTR. Failure to do either causes trouble: some other process may try to read from the line instead of `getty`[1], or the line may appear to be dead, thus blocking a telephone ‘‘hunt group’’.

Historically, these requirements have been a very fruitful source of bugs. Most versions of the UNIX system either suffer from such bugs, contain large amounts of code to try to avoid them (i.e., *forceclose*() in the 9th Edition and 4.*n*BSD kernels), or both. None of these solutions is particularly satisfactory.

The session tty mechanism avoids all of these problems by decoupling the physical tty device from the tty device visible to the user. A logical connection is established at login time; it is severed when either the physical device notices that the connection has dropped, or when the login shell exits.

In addition to solving the original problem, there are several other benefits to session ttys:

1. Permitting a user to reconnect to a session after a physical hangup is easy.
2. Remote login sessions can be assigned ''tty'' names for `/etc/utmp` entries; thus, they will show up via *who*.
3. The `/etc/utmp` entry for the login session will remain around while any child processes persist. This helps system administrators track down hidden resource consumers.
4. Resources allocated to a session — say, a tape drive assigned to a user for the duration of several commands — should be revoked when the session ends. In current systems, it is very hard to determine when this is true.
5. Certain stream modules whose usage must be beyond the user's control (i.e, encryption or audit modules) can be protected from tampering.
6. Non-login sessions — circuit-based network connections, *cron* jobs, etc. — can have `utmp` entries as well.
7. Depending on certain design questions that are not yet clear, session ttys may provide a clean solution to the problem of which window is logged in when using *layers*.
8. Again, depending on certain as-yet unresolved questions, use of tty lines for two-way traffic may become significantly cleaner and easier.

The implementation discussed here is for System V Release 3, as it heavily relies on *streams*[1] and on several other unique features, such as multiplexor devices and clone devices.[2] Some features not in the standard system, such as a streams-based terminal line discipline, are also used. It is possible to implement the session manager on other versions of the UNIX system, but it is somewhat more difficult to do so.

2. HOW IT'S DONE

There are three key components to the new design: the *session manager*, a multiplexed device driver known as *sesstty*, and one or more *line managers*. Depending on the exact hardware and software configuration, the line manager may be a separate process, part of an existing ''listener'' process, or part of the session manager itself.

2.1 The Current Structure

Before we discuss the new implementation, let us review the current login management structure. When the system makes a transition to multi-user state, `/etc/init` creates a child process for each tty line, as defined in **/etc/inittab**; each child process in turn executes `/etc/getty`, which opens the actual device. When the open succeeds, `getty` sets up the hardware environment (principally the line speed), collects the user's login id, and passes control to `/bin/login`. As noted earlier, the physical device

1. The security implications of this are amusing; it allows a process to imitate `getty` and `login`, thus capturing users' passwords.
2. See the glossary at the end.

opened by `getty` is passed directly to `login`. `Login` validates the login id (i.e., verifies a password), creates an entry in `/etc/utmp`[3], sets up the user's environment, and passes control to the user's shell. The only *fork*() operation in the entire process is the one performed by `init`; the user's shell is thus a direct descendant of `init`, with a process id known to `init`. We refer to that process as a *session process*. When the shell terminates — that is, when the user logs off — `init` is notified of the process id; it cleans up the `utmp` entry and spawns another `getty` process for that line.

The following table summarizes the process:

TABLE 1. Division of Responsibility

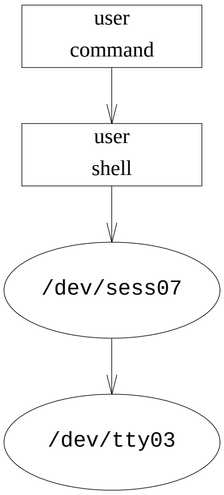
Program	*Function*	*External triggers*
`/etc/init`	Fork and invoke `getty` Clean up `/etc/utmp` entry	Previous session process died
`/etc/getty`	Hardware environment setup Collect login id	Line open succeeds
`/bin/login`	Validate login id Software environment setup Create `/etc/utmp` entry	
`/bin/sh`	(user session) exit	 User logs off or line drops

Network tty connections do not differ in any fundamental way. `/etc/init` is no longer in the picture; the login shell will be child of some network listener process. It in turn is responsible for cleaning up any `utmp` entries. `Getty` may or may not be used.

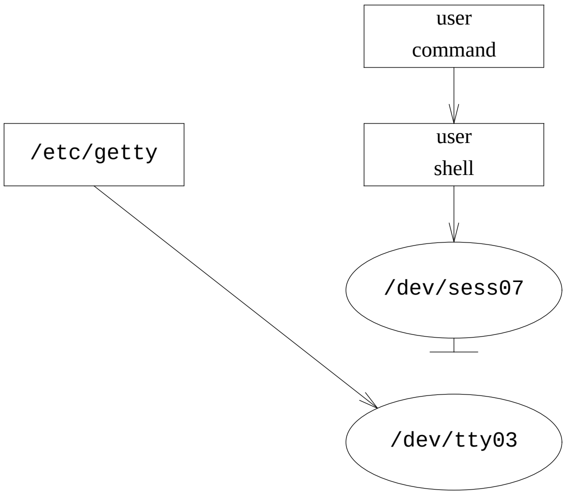
2.2 A Brave New World

The primary conceptual change in the new system is that the physical device is no longer passed directly to `login`. Rather, a software pseudo-device — `sesstty`, as you probably guessed — is opened by `getty`. By dint of the *clone* device driver, a separate `sesstty` device is used for each session. Next, the physical line is spliced to the `sesstty` device by the `I_LINK` *ioctl*() call. For the duration of this session, all references to the line, including its `/etc/utmp` entry, are via the `sesstty` device instead. The following diagram summarizes the situation:

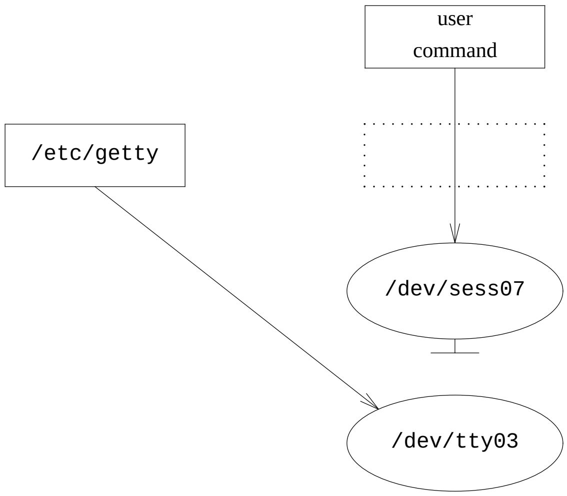
3. Strictly speaking, the `/etc/utmp` entry is created by `init`. This is primarily a bookkeeping entry, however; the significant entry — i.e., the one showing that someone has logged in — is created by `login`.



When the physical line drops, the connection between the session device and the physical device is broken. As shown below, a new invocation of `getty` can attempt to reopen the line, without affecting the old session device still being used.



The connection is also severed if the shell exits while any of its children are still running:



Although the new scheme appears to be a minor enhancement to the current system, there are actually several complicating factors. For one thing, we have added some new external events. A new `getty` process must be spawned when the link is severed; this may or may not be coincident with the demise of the session process. Second, session cleanup, notably clearing the `/etc/utmp` entry, occurs when the `sesstty` device is closed, which in turn will not normally happen until all child processes terminate. Both are currently handled by `/etc/init`; however, it seems unwise to further complicate `init` further by adding new functionality to it.

We solve these problems by divesting two of `init`'s roles to some new programs, the *session manager* and the *line manager*. `Init` will fire up a single session manager, and one or more line managers, in accordance with the current system state. The line manager is similar to the current `/etc/getty`. Its primary task is to open new tty lines; additionally, it handles hardware environment setup and collects and validates the login id. After that, however, it does not invoke `login`; rather, it opens a stream to the session manager and issues `I_SENDFD` *ioctl*() calls to pass the physical line to the session manager. It never invokes an actual shell. A typical implementation would fork a child for each line or network connection; the child would exit when the line had been handed to the session manager.

In general, there may be an arbitrary number of line managers. One might handle all ordinary tty lines, a second might handle remote login requests as part of a ‘‘listener’’ process, etc. If the line manager is some sort of listener process, no special action need be taken to re-enable the line after it has dropped. The situation is somewhat more complex if we are dealing with ordinary tty lines, where each port must be opened separately. In such cases, an explicit message must be sent to the line manager by the session manager.

The session manager is the parent of all user shells, thereby relieving `/etc/init` of the responsibility. At initialization time, it opens the *control channel* of `sesstty`. When the session manager receives a file descriptor from a line manager, it links it to a session device, creates the initial `utmp` entry, and forks to a replacement for `/bin/login`, known as `/bin/loginenv`. This program — which is not privileged — is responsible for setting up the user environment before passing control to the shell. When the shell terminates, the link between the physical device and the session device is severed (which implies a *close*() of the physical device will take place), and the line manager is advised by the session manager to re-enable the line.

The session manager also receives control messages from `sesstty` telling of physical line drops. Again, the connection to the session device is severed and the line manager appraised of the situation.

In neither case, though, is the `utmp` entry cleared; that happens only when the session device is closed. Such status changes are passed to the session manager via the control channel.

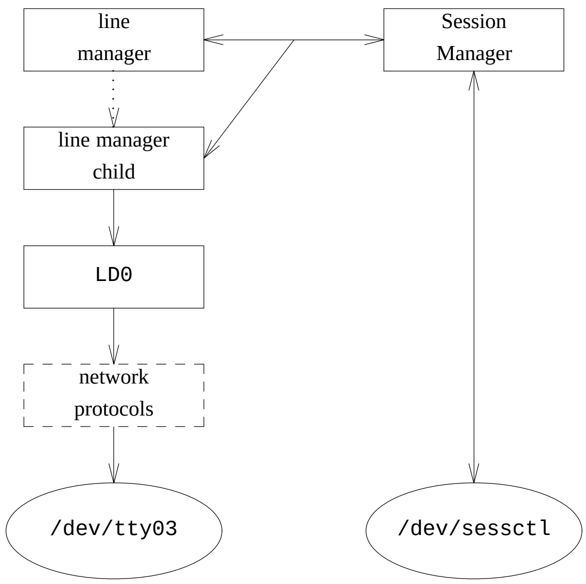
`Sesstty`, although one driver, is divided into two logical sections. The first section handles the control channel; very little data is actually sent on this channel. It is used to pass status change messages to the session manager, and as the device for `I_LINK` and `I_UNLINK` calls. The second section is the mostly-transparent user device handler. Most data and control messages are passed through, whether upstream or downstream. Hangup messages from the real device are diverted to the control channel, though, as are close messages from the stream head.

2.3 The Gory Details

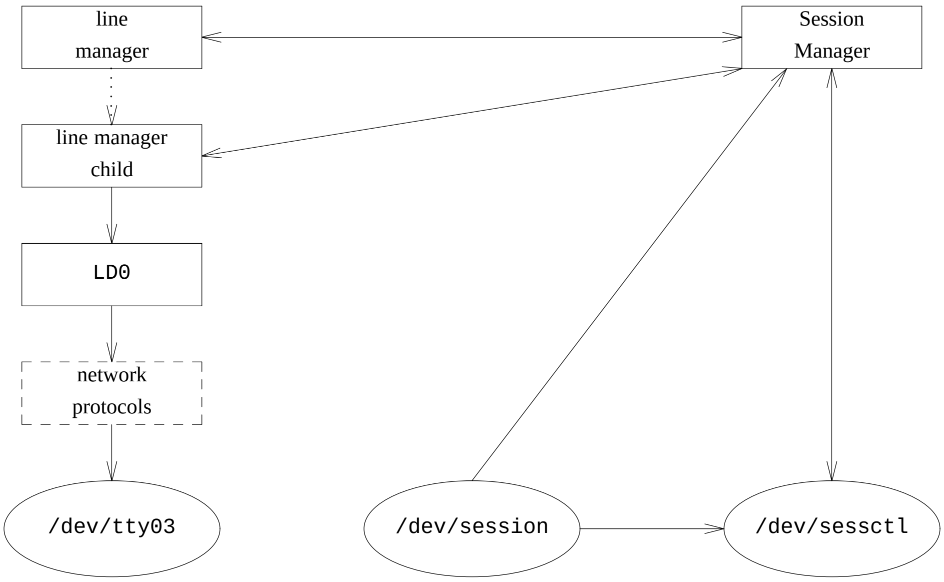
Let us now re-examine the entire structure, this time paying attention to all of the small details. We will do this by examining the boot-time initialization procedures, and then following the progress of a login session from beginning to end, with occasional diversions to discuss what a component is doing while awaiting an event.

At boot time, the session manager opens the session driver control channel and creates a *named stream pipe* for communication with the line managers. Each line manager creates a new stream pipe, opens the named pipe to the session manager, and passes one end of the stream pipe to the session manager. Thus, each line manager has a unique channel to the session manager.

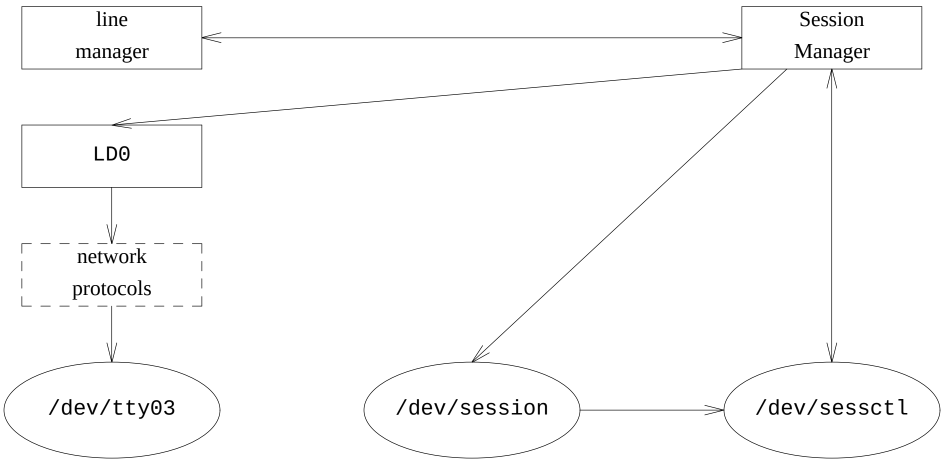
A session starts when the line manager — we will assume for the moment some form of ‘‘super-`getty`’’ — detects that an *open*() attempt on some line has succeeded. The line manager will then *fork*() a per-line child process, to handle the negotiations with the user and the session manager. The child process will collect the login id, much as `getty` does now. It will then *push* `LD0` (the standard tty line discipline), issue the appropriate `TCSETA` call to configure the line, and invoke an authentication mechanism.



Following that, the line manager child creates a new stream pipe and passes it to the session manager, thereby creating a unique channel for each session. This channel is used to pass login information from the line manager child to the session manager, and to send it the file descriptor for the physical line via `I_SENDFD`. The session manager opens the clone device `/dev/session` to create the control terminal for the new session.



At this point, the per-line process may exit.



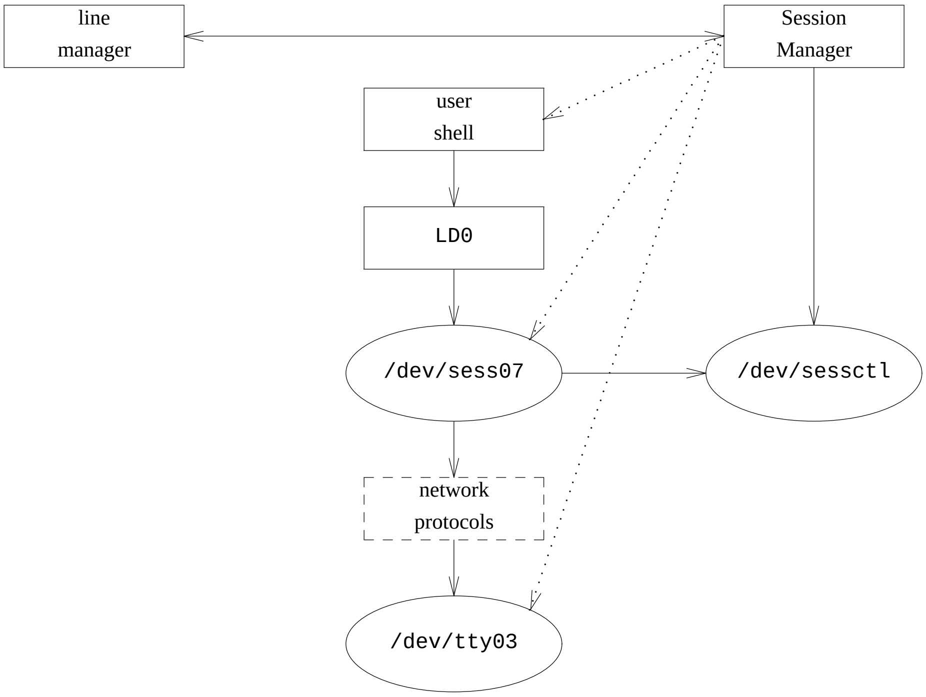
The scenario is similar if we are using a listener-based line manager. The primary difference is that any network-related line disciplines must be pushed onto the stream before LD0. As we shall see, the exact set of disciplines used must be carefully selected, as they will move out of reach of the user before the login process is complete.

Most of the time, the session manager is in a loop waiting for messages on the control channel. Eventually, it will receive a validated login id and a physical device on this channel. At this point, a `utmp` entry is created and a child process is created. This child process must shed any current control tty (i.e., the control tty of the line manager) and open the session device to acquire a new control tty. It then performs a `TCGETA` to note the line environment, pops `LD0`, issues an `I_LINK` to splice the physical line to the session device, and pushes `LD0` onto the session device. This juggling of `LD0` is necessary because line disciplines below the splice point — say, a network protocol module — may not be popped once the splice has taken place. But many programs will want to remove `LD0` on their own; it is not reasonable to isolate it behind this brick wall. (The wall is actually useful for some applications. The line manager may wish to push a cryptographic module; for security reasons, it may be desirable to prevent the user from ever removing it.)

Finally, the child process *exec*()s the user's shell.

Once the fork has taken place, the session manager should close the physical device passed to it by the line manager. For each session, it must record the session device, the real device, the ''magic cookie'' passed back by the `I_LINK` call, and the process id of the child, which will ultimately become the user's shell.

After all of the juggling has taken place, the configuration looks like this. The dotted arrows indicate information; the solid arrows indicate an open file descriptor or some other active data channel.



As noted, the `sesstty` driver passes most data and control messages through without modification or hindrance. An exception is made for one event: an `M_HANGUP` indication coming from the real device. `M_HANGUP` is an actual stream message, and may be diverted if we choose. If left to propagate upward, it will cause `SIGHUP` to be generated; as we have seen, though, this may not accomplish everything we want. Accordingly, the message is diverted, and a status message is sent instead to the session manager. The session manager in turn issues `I_UNLINK`, to sever the connection to the real device; if necessary, it also notifies the line manager to re-enable the device. Finally, the session manager must issue a command (via the control channel) to cause a real M_HANGUP message to be sent to the session device. (We shall see later why we do not simply make a copy of the M_HANGUP to send to the session manager, rather than diverting it.)

Setting up a session is straight-forward. Tearing one down, however, depends on three independent events: termination of the session process (which we will refer to as *PEND),* a close of the session device (*SCLOSE*), and a M_HANGUP of the real device. These may occur in any order. If we judiciously use *sighold*() to block SIGCLD, all of these events will appear to be synchronous to the session manager, thus avoiding potential race conditions.

The cleanup rules can be stated fairly simply.

1. When the first termination event comes in for any session, sever the connection between the session device and the real device, and tell the line manager to re-enable the line.
2. If the first event is M_HANGUP, send an M_HANGUP message up the session device to the user processes.
3. When both PEND and SCLOSE have occurred, the session is over; clean up the utmp entry.
4. Whenever M_HANGUP arrives, send a re-enable message. Hardware hangup signals are external to the machine, and hence asynchronous; depending on internal timing, one may arrive while a re-enable is in transit, and hence its effect may be lost.

We must also deal with the problem of commands that (a) close all of their file descriptors, and (b) persist after the shell itself has exited. This would cause the session device to be closed, which in turn would cause the session entry in /etc/utmp to be deleted. Such processes would then become invisible in some sense. We deal with this by changing the implementation of **/dev/tty** slightly. Rather than storing a pointer to the tty structure in the ‘‘u.’’ area, we instead store a pointer to the file table entry for the session device. That is, when a process acquires a session device as its controlling terminal, the kernel would in effect perform a *dup*() operation on the relevant file descriptor, storing the result in u.u_ttyd. Thus, the session device could not close until all processes in that session have terminated. (The close notification may be delayed even further, if other process outside the session are using the device. The write command is one case in point.)

There is one anomalous case of line re-enabling that must be dealt with. If the child of the line manager does not validate the login id successfully, it will never hand off the connection to the session manager. Consequently, no ordinary re-enable message will ever be sent. There are a number of ways around this problem, such as having the line managers look for special exit codes from their children, or having the children always send re-enable requests to the parent via the session manager.

Let us revisit Table 1 and examine the new division of functionality:

TABLE 2. New Division of Responsibility

Program	*Function*	*External triggers*
`/etc/init`	Fork and invoke session manager and line managers	Previous process died
Line manager	Contact session manager Open set of lines Fork per-line child	 Line open succeeds
Line manager child	Hardware environment setup Collect and validate login id Hand off line to session manager	
Session manager	Create session device Create `utmp` entry Fork/exec `loginenv` Disconnect session Clean up `utmp`	Message from line manager Message from line manager child Line hangup or child process exit Session ends
`/etc/loginenv`	Software environment setup Exec user shell	
`/bin/sh`	(user session) exit receive `SIGHUP`	 User logs off session manager message (after session disconnect)

3. BELLS, GONGS, AND WHISTLES

In addition to fixing some bugs, using the session manager makes it easy to add some new functionality to the system. The most interesting new feature is the ability to have a session survive the disconnection of the physical terminal. Instead of sending `M_HANGUP` to a session when the physical line drops, nothing would be done. Read requests on the session device would eventually block due to lack of input data; write requests would block because of flow control on the stream. Alternatively, a `STOP` signal could be sent to the session's process group instead; that way, a user who disconnected because of stubborn malfunctioning processes would not have to worry about them eating the CPU alive until they were killed manually.

To resume a suspended session, a user would log in and issue a `reconnect` command. This command would ask the session manager to sever the link between the real device and the new session, and then splice the real device to the `sesstty` from the old session. The new session would probably be terminated, though as an option it, too, could be suspended. This would allow a user to switch back and forth between multiple sessions, possibly even on different machines.[4]

It isn't clear how long a disconnected session should be allowed to persist. Permitting such session to live indefinitely would tend to choke a system on the bodies of undead[5] processes. Presumably, a timeout interval would be set by the system administrator; users who wished to suspend a session for a longer time could override the limit via an *ioctl*() call.

Instead of having the output of a disconnected session pile up, we may wish to discard it. The obvious model is a ''streams **/dev/null**''; unfortunately, that's not as simple as it sounds. It's certainly easy enough to implement a pseudo-driver that discards any data flowing into it; however, there is no way for a driver to pass a ''hard EOF'' back to the stream head. If such an enhancement were made, the session device could be spliced to a clone of the null stream. Alternatively, some large number of 0-length

4. Any resemblance between this concept and the `sxt` driver is *not* coincidental; we discuss the similarities below.

5. I can't call them zombies...

records could be sent upstream.

Earlier, we alluded to the resource allocation problem. Briefly, a user may wish to acquire some resource for a span longer than a single command. A tape drive is a good example; the user may need to create a `cpio` image on the tape, verify it, recreate it when the verification shows that the file is incorrect, etc. In a closed-shop environment, assignment of the tape drive to that user may be handled by a tape drive manager daemon. Apart from allocating the drive, this daemon would also request that the operator mount the user's tape. The problem is when to release the drive for allocation to other users. While a time limit is one solution (and probably part of any solution), noting when a session has ended is another. Today, detecting logout is insufficient; the process using the drive may be running in the background. Sessions, however, persist until the last process is gone.

Session-based ttys are also useful to B2-level secure UNIX systems[2]. Such systems require the existence of a *trusted path* — a mechanism invoked by the user that will reliably connect that user to a trusted program. The trusted path must be used at login time; that way, the user is guaranteed that the `Password:` prompt is not coming from a Trojan horse program. Obviously, a requirement for the trusted path is that all other processes talking to the terminal must be killed or have their access rights revoked. Sheerly in terms of implementation problems, this is a complex task. The list of open files for a process is in the process's pageable ''`u.`'' area; thus, it is not easily accessible to the kernel at interrupt time. Session ttys inherently solve the problem. When carrier drops, an `M_HANGUP` message is generated. Upon receipt of the message, the session manager will *always* disconnect the physical device from the session device, and *always* ask the line manager — a trusted process — to re-enable the line. The only change needed to the session manager is that the line may not be re-enabled until `M_HANGUP` is received; presenting a new `login:` prompt upon session head exit violates the DoD requirement that the user must initiate access to the trusted path.

Session heads need not be bound to tty devices; rather, they can be linked to any underlying stream. This gives us the ability to bring other services under the same umbrella. For example, RFS mount operations could be listed in `/etc/utmp`; that would permit easy monitoring of start times, exit status, etc. Other network sessions — file transfer operations, remote execution requests, etc. — could be listed as well. Some care should be exercised in deciding what types of sessions should be listed, of course; things like directory queries or time server queries are likely to be too transient to be worth the overhead.

One essential difference between the current `/etc/init` and a line manager is that the latter knows when an active session is in progress. All `init` can say is that something is running; it cannot tell whether or not the line is actually open. But this knowledge is extremely important to programs that wish to use the line for outgoing calls. Current solutions are not particularly clean. `Uucico`, for example, requires a special version of `getty` known as `uugetty` for shared lines. A better solution would be for it to ask the line manager for a line — any available line — rather than relying on heuristics to determine what is happening. Ideally, in fact, such negotiations would be handled by a connection agent, rather than by each individual client program.

4. ERROR RECOVERY

A serious disadvantage to the session manager is that is creates new critical processes. If the session manager crashes, or if a line manager gets confused, it would be impossible to log in to the system. It is thus necessary to plan for such occurrences.

It is comparatively easy to recover from a line manager crash. A tty line manager needs one essential piece of information: which lines are currently in use, and hence which other lines should be enabled for login. We accomplish this at line manager startup by having it ''log in'' to the session manager. As part of this dialog, it indicates whether or not it cares about lines currently in use; if it does care, the session manager will send it a list. Ports assigned to that line manager are enabled if not currently in use.

Network-based line managers are much simpler. They rarely care which ports are in use; that is generally the concern of the transport layer. Thus, their login sequence does not request notification.

In either case, it is possible to kill and restart the line manager without loss of state. This suggests that for redundancy, at least two line managers should run on each system. Typically, one would handle the console only, while a second would handle all other tty lines. Additional line managers would be used for each network type.

Recovery from a session manager crash is more difficult. When the session manager crashes, the `sessctl` device is closed, thereby automatically unlinking all of the session devices from the physical lines. Users are disconnected, but their processes can remain suspended; when the session manager is restarted, reconnection is possible. Unfortunately, if the session manager is the sole means of access to a system, it isn't possible to log in and restart it. Some crashes can be handled by having a daemon periodically query the session manager, and restart it if necessary; obviously, though, there are many failure modes immune to this sort of automated detection. Perhaps the best solution is to have a standing network server that will, on request, kill and restart the session manager.[6]

5. SESSION DEVICES WITHOUT SESSION MANAGERS

In some environments, a simplified version of this mechanism can be used. We can eliminate the session manager and the line managers, and simply change `getty` to talk to the session driver.

Assume that all access to the system is via a network, or via a port selector that will connect a user to any free port that has DTR enabled. Assume further that there are sufficiently many ports on the system that a short-term loss of some is not a serious problem, but that revoking access to the terminal is important. In that case, when `getty` answers a line, it would open the clone driver `/dev/session` and splice the physical line below it. All further login proceedings would take place on the session device.

We now have two end conditions: the login shell exiting, and carrier dropping on the physical line. When the former happens, a new `getty` will be spawned by `init`. If there are no left-over processes, the session device will be closed, which in turn will cause the physical line to be unlinked. The `getty` can then proceed normally. If there are left-over processes, `getty` would open the old session device and manually unlink the physical line; life could then proceed as before.

Carrier drops are more problematic. If the login shell does not exit, no new `getty` is spawned; hence no process issues the `I_UNLINK` call. However, the `M_HANGUP` message is detectable by the session driver. It cannot unlink the line — for assorted complex reasons, that cannot be done at interrupt time — but it can block output to the physical device from the session device. Additionally, the physical device driver can be modified to drop DTR if carrier drops; thus, the line will not be available for reuse until it is reopened by some future `getty`.

This simplified version is very susceptible to denial of service attacks. An enemy could log in and then drop the line, thereby tying up a session device. Some provision must be made to detect too many session devices being assigned to one user.

Despite appearances, it may be possible to use this version with a modem pool. Many modems can be configured to busy out the line if DTR is low; thus, a disabled line will be skipped by the phone network.

6. IMPACT ON OTHER SYSTEM COMPONENTS

The session driver requires remarkably few changes to other system components. Obviously, the entire implementation depends on the tty subsystem being converted to use streams instead of `clist`s. Most of the changes are to `getty` and `login`. At a minimum, the version of `login` invoked by `getty` must be changed to communicate with the session manager after collecting the login name. More likely, we could remove the password-checking code from `login`, and let `getty` — a line manager —

---

6. A better solution could be contrived if someone would bring back console bit switches and lights....

invoke an authentication routine. It is important to perform authentication before discarding knowledge of the physical device.

`Getty` also needs to be integrated with a line manager. That can be done in either of two ways. First, we could write a separate line manager program; it would fork and invoke per-line `getty` processes, much as `init` does now. Alternatively, we could write a ‘‘super-`getty`’’, a program that would have *open*() requests outstanding on multiple lines at once, and would only *fork*() when the *open*() succeeded. This latter approach reduces clutter in the system process table, at the cost of greater complexity and lesser robustness. Super-`getty` is the only solution if we wish to implement two-way lines.

If `login` loses its authentication code, and hence becomes unprivileged, the code in `sh` that *exec*()s `/bin/login` must be changed. Conceptually, what must be done is to disconnect from the current session, while asking the line manager to re-enable the line without dropping it first. It is quite conceivable that some line managers will not be willing or able to do this. More thought needs to be given to this question.

Existing listener processes will need to be converted to talk to the session manager. Currently, there are only two major ones for login sessions, the DKHOST listener for the Datakit® VCS and `/usr/net/listen` when used with STARLAN.

The **/dev/tty** change described earlier is actually independent of the other changes described here. In fact, it could be done with the current system, though there would be added complexity when handling end conditions such as an orphaned background process. Still, those problems are not insurmountable. Apart from changes to the **/dev/tty** driver itself, the *fork*() code would have to be changed to perform the *dup*() operation, and *exit*() would have to perform an additional *close*().

The current accounting routine records the device number of the controlling tty. Under sessions, that value would be the device number of the session head, which has no physical reality. If the physical device number is desired there, some mechanism to announce it must be provided. On the other hand, it may not pay to change that; network connections have no true physical device number available, and the session device number serves to group together commands from a given session.

It would be nice to add some new fields to the `utmp` structure, to record a link to the physical device or source host; compatibility considerations may preclude this, however. If `utmp` cannot be changed, a parallel file containing this information must be set up. A few more type codes for `utmp` are certainly needed to indicate disconnected sessions and the like.

Use of the session manager will exacerbate an existing incompatibility. There are a few programs, including some versions of the Korn shell, that use *getppid*() to determine if they are direct children of `init`. Under this new scheme, almost nothing will be. Of course, the same is true now for children of listener processes.

7. FUTURE DIRECTIONS

As noted, the ability to bounce back and forth between different sessions is very similar to functionality provided by the `sxt` driver. Indeed, by rewriting `shl` we could eliminate the driver entirely. The new `shl` command would act as a line manager, and set up a new session for each shell layer created. There is some problem with implementing `noloblk` in a disconnected session; one solution is to have `shl` splice a stream pipe to all of its session heads, and let it perform the multiplexing at user level. A better idea might be to install real job control, and discard `shl` entirely.

A similar technique could be used to replace the `xt` driver; `layers` could do the multiplexing and handle the line protocol to the DMD itself, much as `mux` in 9th Edition and `mpx` in 4.3BSD do. This would shrink and simplify the kernel, and allow easier modifications to the protocol handler.

There is certainly some efficiency loss in having this multiplexing done at user level. On the other hand, it has always been the UNIX system philosophy that the kernel should provide the basic primitives and plumbing, while letting application programs provide the richness and complexity. This is perhaps best-illustrated by the duplication of code represented by having both `xt` and `sxt` present in the same system.

If efficiency is a major concern, the streams mechanisms provide several handles to resolve the problem. Perhaps the nicest solution for `layers` is to adopt `mux`'s approach: to move the tty line discipline into the terminal. Thus, only complete lines are passed to user level, thereby avoiding the expense of several packets and context switches per character typed. Alternatively, a ''buffer'' line discipline could delay characters sent from the terminal until a complete packet was received, again reducing the context-switch overhead. (Obviously, such a line discipline would need to turn itself off if a raw-mode application such as `vi` were running.)

Letting each layer be a session has a curious implication: each such layer would (or at least, could) have a `utmp` entry. This is both good and bad. On the one hand, it provides a clearer picture of who is doing what, and eliminates such oddities as the `relogin` command. On the other hand, if each layer appears to be a true login session, commands that scan `utmp` need to be modified to realize this. It isn't pleasant to receive six separate copies of a `wall` message, for example. (It turns out that `wall` is a particularly nasty case. Programs that are split between a DMD and the host, such as `jim` and `cip`, can get very confused if they receive data that isn't part of their protocol. Under the current scheme on System V, that is only a problem if one is sufficiently incautious as to invoke such a program in the designated login window. One can bypass this specific problem by modifying `wall` to ignore lines that are in raw mode; this does not solve the general problem, however.) On balance, we tend to like having each layer appear in `utmp`, but it is certainly a debatable point.

8. IMPLEMENTATION QUIRKS

The *Careful Reader* will have noticed that this design uses several features — stream pipes, named stream pipes, and a stream tty line discipline — that do not appear to be part of System V Release 3. This perception is somewhat correct in practice, and entirely correct in principle.

The stream tty line discipline is a prototype adapted by others from the STARLAN support.

Stream pipes are currently supported in a strange fashion. By opening the clone device `/dev/spx`, a process receives half of a stream pipe. To create a full stream pipe, `/dev/spx` must be opened again, and the two halves spliced together via the `I_FDINSERT` *ioctl*() call. In the distributed system, only `root` may open `/dev/spx`. The `/dev/spx` driver is not considered to be a public interface, and may not be present in this form in future releases of System V. It does work today.

Creating named stream pipes is even more bizarre. First, one creates a stream pipe. An *fstat*() call is used to determine the major and minor device numbers used for one end of the stream pipe. (Recall that these are actual devices as far as the kernel is concerned.) A *mknod*() call is then used to create a new, named `i-node` corresponding to that pipe end. Other processes may open this filename and receive a file descriptor equivalent to the original pipe end.

All processes that open a named stream pipe receive the same pipe end. That is, data written by them may be intermixed, and no identifying information is transmitted to the far end. Creating unique connections from the line managers to the session manager is a key problem.

We accomplish this by using the named pipe solely to pass file descriptors. That is, each line manager, at startup, creates a new stream pipe and passes one end of it to the session manager via the named pipe. All further communication takes place on this new pipe. A facility similar to 4.2BSD's *UNIX-domain sockets* would simplify matters.

When a session is starting up, each line manager creates yet another stream pipe for the per-line child process to pass to the session manager via the line manager's pipe. This separate connection is created for two reasons. First, there is the aforementioned problem of having multiple simultaneous conversations on a single pipe. Second, when `I_SENDFD` is used to pass file descriptors, these file descriptors are sent immediately to the head of the receiving process's read queue. They are not associated with any data message, nor are they sent synchronously with one. The session manager is thus incapable of telling which physical line goes with which session. 4.2BSD's mechanism for passing access rights to a file along with some text would alleviate this problem.

9. UNSOLVED PROBLEMS

The current design is very heavily oriented towards login-type sessions. That is, it assumes that a shell is the desired end-point of any connection. Although this is certainly true for login sessions, it is not true for some other types of session. For example, a remote execution request would need very different parameters to that shell than an interactive request. An FTAM session would not even use a shell. The current session manager protocol will have to be redesigned or extended so that the line manager (or rather, the program acting as a line manager in this context) can retain ownership of the session device. Some possible new designs would make the session manager disappear entirely as a separate program.

The issue of subsessions, for layers in particular, needs more thought. The effect of a disconnection of the physical line must be propagated to all subsessions. Since sessions can only be created by `root`, it may be sufficient to inform the primary session of hangups; on the other hand, privileged programs are not, in general, immune to bugs. If we are using sessions for resource management, we need a way to bind the resource to the real session, rather than any subsession; it is certainly reasonable to use a tape drive in a window other than the one it was allocated from.

The biggest wart in the current design is the nature of the communications channel between the session manager and the line manager. The current scheme requires that a large number of file descriptors be kept open by the session manager; if each invocation of `layers` is in effect a line manager, the session manager could exceed the system limit.

10. SUMMARY

The session manager is a single concept that solves several problems:

1. It introduces a strong concept of a session into the UNIX system.
2. Each session is isolated from all other sessions on the same physical line.
3. The handling of modem control signals can be made reliable, despite ill-behaved user processes.
4. Non-login connections and windows can be treated as sessions, if desired.

11. ACKNOWLEDGEMENTS

I’d like to thank Steve Albert for reading and commenting on an early draft of this document. He, Paul Lustgarten, Nancy Mintz, and Ed Whelan gave me several opportunities to explain the entire concept, thereby helping me clarify the designs; they also helped me focus on the essential parts of it. Additionally, Dennis Ritchie had a number of useful suggestions.

GLOSSARY

clone driver — A clone driver provides a means to assign a unique minor device to each process using the driver. When a clone device is opened, an idle minor device is selected, and the `i–node` used is modified to point to this minor device rather than the clone device `i–node`.

I_LINK — An `ioctl`() command used to set up the linkage between a multiplexor driver and a stream.

I_SENDFD — An `ioctl`() command used to send an open file descriptor across a stream pipe.

I_UNLINK — An `ioctl`() command used to tear down the linkage between a multiplexor driver and a stream.

listener — A process that receives and dispatches connection requests arriving via a network.

multiplexor driver A multiplexor driver is a pseudo-device driver that permits any open stream to be linked beneath it. It can be used to layer a protocol — say, IP — on top of a physical device. Thus, IP could be implemented as a multiplexor driver, while the Ethernet interface would be a stream to be linked beneath it.

named stream pipe A stream pipe with a name in the file system that can be opened by other processes. Akin to a `FIFO`, but full-duplex.

session manager See pp. 1-16.

stream pipe Similar to an ordinary pipe, except that it is implemented via streams. Thus, it is full-duplex. A stream pipe may be used to transmit file descriptors via `I_SENDFD` and `I_RECVFD`. It is possible to push modules onto a stream pipe, but that feature is not used by the session manager.

UNIX-domain socket A mechanism for intra-machine, inter-process communication on 4.2BSD and 4.3BSD. Addresses in this domain look like file names. Both data and access rights — file descriptors — may be passed across UNIX-domain connections.

REFERENCES

1. Ritchie, D.M. ‘‘A Stream Input-Output System’’. *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, part 2 (October 1984), pp. 1897-1910.

2. DoD Computer Security Center. *DoD Trusted Computer System Evaluation Criteria*, 1983, CSC-STD-001-83.

done with aps