Clusterproc: Linux Kernel Support for Clusterwide Process Management

Bruce J. Walker
Hewlett-Packard
bruce.walker@hp.com

Laura Ramirez
Hewlett-Packard
laura.ramirez@hp.com

John L. Byrne
Hewlett-Packard
john.l.byrne@hp.com

Abstract

There are several kernel-based clusterwide process management implementations available today, providing different semantics and capabilities (OpenSSI, openMosix, bproc, Kerrighed, etc.). We present a set of hooks to allow various installable kernel module implementations, with a high degree of flexibility and virtually no performance impact. Optional capabilities that can be implemented via the hooks include: clusterwide unique pids, single init, heterogeneity, transparent visibility and access to any process from any node, ability to distribute processes at exec or fork or thru migration, file inheritance and full controlling terminal semantics, node failure cleanup, clusterwide `/proc/<pid>`, checkpoint/restart and scale to thousands of nodes. In addition, we describe an OpenSSI-inspired implementation using the hooks and providing all the features described above.

1 Background

Kernel based cluster process management (CPM) has been around for more than 20 years, with versions on Unix by Locus[1] and Mosix[2]. The Locus system was a general purpose Single System Image (SSI) cluster, with a single root filesystem and a single namespace for processes, files, networking and interprocess communication objects. It provided high availability as well as a simple management paradigm and load balancing of processes. Mosix focused on process load balancing. The concepts of Locus have moved to Linux via the OpenSSI[3] open source project. Mosix has moved to Linux via the openmosix[4] project.

OpenSSI and Mosix were not initially targeted at large scale parallel programming clusters (eg. those using MPI). The BProc[5] CPM project has targeted that environment to speed up job launch and simplify process management and cluster management. More recent efforts by Kerrighed[6] and USI[7] (now Cassat[8]) were also targeted at HPC environments, although Cassat is now interested in commercial computing.

These 5 CPM implementations have somewhat different cluster models (different forms of SSI) and thus fairly different implementations, in part driven by the environment they were originally developed for. The “Introduction to SSI” paper[10] details some of the differences. Here we outline some of the characteristics relevant to CPM. Mosix started as a workstation technology that allowed a user on one workstation to utilize cpu and memory from another workstation by moving running processes (process migration) to the other workstation. The mi-

grated processes had to see the OS view of the original workstation (home node) since there was no enforced common view of resources such as processes, filesystem, ipc objects, binaries, etc. To accomplish the home-node view, most system calls had to be executed back on the home node—the process was effectively split, with the kernel part on the home node and the application part elsewhere. What this means to process ids is that home nodes generate ids which are not clusterwide unique. Migrated processes retain their home node pid in a private data structure but are assigned a local pid by the current host node (to avoid pid conflicts). The BProc model is similar except there is a single home node (master node) that all processes are created on. These ids are thus clusterwide unique and a local id is not needed on the host node.

The model in OpenSSI, Kerrighed and Cassat is different. Processes can be created on any node and are given a single clusterwide pid when they are created. They retain that pid no matter where they execute. Also, the node the process was created on does not retain part of the process. What this means to the CPM implementation is that actions to be done against processes are done where the process is currently executing and not on the creation or home node.

There are many other differences among the CPM implementations. For example, OpenSSI has a single, highly available init process while most/all other implementations do not. Additionally, BProc does not retain a controlling terminal (or any other open files) when processes move, while other implementations do. Some implementations, like OpenSSI, support clusterwide ptrace, while others do not.

With some of these differences in mind, we next look at the goals for a set of CPM hooks that would satisfy most of the CPM implementations.

2 Goals and Requirements for the Clusterproc Hooks

The general goals for the hooks are to enable a variety of CPM implementations while being non-invasive enough to be accepted in the base kernel. First we look at the base kernel requirements and then some of the functional requirements.

Changes to the base kernel should retain the architectural cleanliness and not affect the performance. Base locking should be used and copies of base routines should be avoided. The clusterproc implementations should be installable modules. It should be possible to build the kernel with the hooks disabled and that version should have no impact on performance. If the hooks are enabled, the module should be optional. Without the module loaded, performance impact should be negligible. With the module loaded, one would have a one node cluster and performance will depend on the CPM implementation.

The hooks should enable at least the following functionality:

- optionally have a per process data structure maintained by the CPM module;
- allowing for the CPM module to allocate clusterwide process ids;
- support for distributed process relationships including parent/child, process group and session; optional support for distributed thread groups and ptrace parent;
- optional ability to move running processes from one node to another either at exec/fork time or at somewhat arbitrary points in their execution;

- optional ability to transparently checkpoint/restart processes, process groups and thread groups;
- optional ability to have process continue to execute even if the node they were created on leaves the cluster;
- optional ability to retain relationships of remaining processes, no matter which nodes may have crashed;
- optional ability to have full controlling terminal semantics for processes running remotely from their controlling terminal device;
- full, but optional `/proc/<pid>` capability for all processes from all nodes;
- capability to support either an "init" process per node or a single init for the entire cluster;
- capability to function within a shared root environment or in an environment with a root filesystem per node;
- capability to be an installable module that can be installed either from the ramdisk/initramfs or shortly thereafter;
- support for clusters of up to 64000 nodes, with optional code to support larger;

In the next section we detail a set of hooks designed to meet the above set of goals and requirements. Following that is the design of the OpenSSI 3.0, as adapted to the proposed hooks.

3 Proposed Hook Architecture, Hook Categories and Hooks

To enable the optional inclusion of clusterwide process management (referred also as "clusterproc" or CPM) capability, very small data structure additions and a set of entry points are proposed. The data structure additions are a pointer in the task structure (CPM implementations could then allocate a per process structure that this pointer points to), and 2 flag bits. The infrastructure for the hooks is patterned after the security hooks, although not exactly the same. If `CONFIG_CLUSTERPROC` is not set, the hooks are turned into inline functions that are either empty or return the default value. With `CONFIG_CLUSTERPROC` defined, the hook functions call clusterproc ops if they are defined, otherwise returning the default value. The ops can be replaced, and the clusterproc install-able module will replace the ops with routines to provide the particular CPM implementation. The clusterproc module would be loaded early in boot. All the code to support the clusterwide process model would be under GPL. To enable the module some additional symbols will have to exported to GPL modules.

The proposed hooks are grouped into categories below. Each CPM implementation can provide op functions for all or some of the hooks in each category. For each category we list the relevant hook functions in pseudo-C. The names would actually be `clusterproc_xxx` but to fit here we leave out the `clustproc_` part. The parameters are abreviated. For each category, we describe the general purpose of the hooks in that category and how the hooks could be used in different CPM implementations. The categories are: Init and Reaper; Allocation/ Free; Update Parent; Process lock/unlock; Exit/Wait/Reap; Signalling; Priority and Capability; Setpgid/Setsid; Ptrace; Controlling Terminal; and Process movement;

3.1 Init and Reaper

```
void single_init();
```

```
void child_reaper(*pid);
```

One of the goals was to allow the cluster to run with a single init process for the cluster. The `single_init` hook in `init/main.c`, `init()` can be used in a couple of ways. First, if there is to be a single init, this routine can spawn a "reaper" process that will locally reap the orphan processes that init normally reaps. On the node that is going to have the init, the routine returns to allow init to be exec'd. On other nodes it can exit so there is no process 1 on those nodes. The other hook in this category is `child_reaper`, which is in timer.c, `sys_getppid()`. It returns 1 if the process's parent was the `child_reaper` process. Neither of these hooks would be used in those CPM implementations that have an init process per node.

3.2 Allocation/ Free

```
int fork_alloc(*tsk);
void exit_dealloc(*tsk);
int pid_alloc(pid);
int local_pid(pid);
void strip_pid(*pid);
```

There are 5 hooks functions in this category. First is `fork_alloc`, which is called in `copy_process()` in `fork.c`. This routine is called to allow the CPM to allocate a private data structure pointed to by `clusterproc` pointer which is added to the task structure. Freeing that structure is done via the hook `exit_dealloc()` which is called in `release_task()` in `exit.c` and under error conditions in `copy_process()` in `fork.c` The `exit_dealloc` routine can also be used to do remote notifications necessary for pgrp and session management. All CPM implementations will probably use these hooks. The other 3 hooks deal with pid allocation and freeing and are only used in those implementations presenting a clusterwide single pid space. The `pid_alloc` hook is called in `alloc_pidmap()` in `pid.c`. It takes a locally unique pid and returns a clusterwide unique pid, possibly by encoding a node number in some of the high order bits. The `local_pid` and `strip_pid` hooks are in `free_pidmap()`, also in `pid.c` The `local_pid` hook returns 1 if this pid was generated on this node and the process id is no longer needed clusterwide. Otherwise return 0. The `strip_pid` hook is called to undo the effects of `pid_alloc` so the base routines on each node can manage their part of the clusterwide pid space.

3.3 Update parent

```
int update_parent(*ptsk,*ctsk,
            flag,sig,siginfo);
```

`Update_parent` is a very general hook called in several places. It is used by a child process to notify a parent process if the parent process is executing remotely. In `ptrace.c`, it is called in `__ptrace_unlink()` and `__ptrace_link()`. In the arch version of `ptrace.c` it is called in `sys_ptrace()`. In `exit.c` it is called in `reparent_to_init()` and in `fork.c`, `copy_process()`, it is called in the `CLONE_PARENT` case. Although not all CPM implementations will support distributed ptrace or `CLONE_PARENT`, support for some of the instances of this hook will probably be in each CPM implementation.

3.4 Process lock/unlock

```
void proc_lock(*tsk,base_lock);
void proc_unlock(*tsk,base_lock);
```

The proc_lock and proc_unlock hooks allow the CPM implementation to either use the base tsk->proc_lock (default) or to introduce a sleep lock in their private process data structure. In some implementations, a sleep lock is needed because remote operations may be executed while this lock is held. In addition, calls to proc_lock and proc_unlock are added in exit_notify(), in exit.c, because exit_notify() may not be atomic and may need to be interlocked with setpgid() and with process movement (the locking calls for setpgid and process movement would be in the CPM implementation.

3.5 Exit/Wait/Reap

```
int rmt_reparent_children(*tsk);
int is_orphan_pgrp(pgrp);
void rmt_orphan_pgrp(newpgrp,*ret);
void detach_pid(*tsk,nr,type);
int rmt_thread_group_empty(*tsk,pid,opt);
int rmt_wait_task_zombie(*tsk,noreap,
      *siginfo,*stat_addr, *rusage);
int rmt_wait_stopped(*tsk,int,noreap,
       *siginfo,*stat_addr,*rusage);
int rmt_wait_continued(*tsk,noreap,
      *siginfo,*stat_addr, *rusage);
```

There are several hooks proposed to accomplish all the actions around the exit of a process and wait/reap of that process. One early action in exit is to reparent any children to the child_reaper. This is done in forget_original_parent() in exit.c. The rmt_reparent_children hook provides an entry to reparent those children not executing with the parent. Accurate orphan process group processing can be difficult with other pgrp members, children and parents all potentially executing on different nodes. The "home-node" model implementations will have all the necessary information at the home node. For non–home-node implementations like OpenSSI, two hooks are proposed—is_orphan_pgrp and rmt_orphan_pgrp. is_orphan_pgrp is called in is_orphan_pgrp(), in exit.c. It returns 1 if orphaned and 0 if non-orphaned. If not provided, the existing base algorithm is used. rmt_orphan_pgrp is called in will_become_orphaned_pgrp() in exit.c. It is called if there are no local processes remaining that make the process group non-orphan. In that case it determines if the pgrp will become orphan and if so it effects the standard action on the pgrp members. A detach_pid hook in detach_pid() is proposed to allow CPM implementations to update any data structures maintained for process groups and sessions.

There are 4 proposed hooks in wait. The first, in eligible_child(), exit.c, is rmt_thread_group_empty. This is used to determine if the thread group is empty, for thread groups in which the thread group leader is executing remotely. If it is empty, the thread group leader can be reaped; otherwise it cannot. The other 3 hooks are rmt_wait_task_zombie, rmt_wait_stopped and rmt_wait_continued which are called in wait_task_zombie(), wait_task_stopped() and wait_task_continued() respectively. These hooks allow the CPM implementation to move the respective functions to the node where the process is and then execute the base call there, returning an indication if the child was reaped or if there was an error.

3.6 Signalling

```
void rmt_sigproc(pid,sig,*siginfo,*error);
int pgrp_list_local(pgrp,*flag);
int rmt_sigpgrp(pgrp,sig,*signfo);
void sigpgrp_rmt_members(pgrp,sig,*siginfo,
                                *reg,flag);
int kill_all(sig,*siginfo,*count,*ret,tgid);
void rmt_sig_tgkill(tgid,*siginfo,pid,flag,
                             *tsk*,*error);
void rmt_send_sigio(*tsk,*fown,fd,band);
int rmt_pgrp_send_sigio(pgid,*fown,fd,band);
void rmt_send_sigurg(*tsk,*fown);
int rmt_pgrp_send_sigurg(pgid,*fown);
void timedwait(timeout,*timespec,*ret);
```

There are many places in the kernel that may signal a process, for a variety of of reasons. Hooks in `kill_proc_info()` and `kill_pg_info()` handle many cases. One could define a general hook function that handles many of the cases (process, pgrp, killall, sigio, sigurg, etc.). Doing so would reduce the number of different hook functions but would require a superset of parameters and the op would have to relearn the reason it was called. For now we have proposed them as separate hooks. `rmt_sigproc` is the hook in `kill_proc_info()`, called only if the process is not found locally. It tries to find the process on another node and deliver the signal. For the process group case we currently have 3 hooks in `kill_pg_info()`. Based on the assumption that some node knows the list of pgrp members (or at least the nodes they are on), the first hook (`pgrp_list_local`) determines if such a list is local. If not, it calls `rmt_sigpgrp` which will transfer control to such a node, so the base `kill_pg_info()` can be called. Given we are now executing on the node where the list is, the `pgrp_list_local` hook can lock the list so that no members will be missed during the signal operation. After that, the base code to signal locally executing pgrp members is executed, followed by the code to signal remote members (`sigpgrp_rmt_members`). That hook also does the list unlock. Support for clusterwide kill –1 is provided by a hook in `kill_something_info()`. A CPM implementation could loop thru all the nodes in the cluster, calling `kill_something_info()` on each one. Linux has 2 system calls for thread signalling—`sys_tkill()` and `sys_tgkill()`. The proposed hook `rmt_sig_tgkill` is inserted in each of these system calls to find the thread(s) and deliver the signal, if the threads were not already found locally. The `send_sigio()` function in `fcntl.c` can send process or pgrp signals. The `rmt_send_sigio` or `rmt_pgrp_send_sigio` hook is called if the process or process group is remote. Similar hooks are needed in `send_sigurg()` in `fcntl.c` (with different parameters). The final signal related hook is `timedwait`, which is called from `sys_rt_sigtimedwait()`, in `signal.c`. It is called only if a process was in a scheduled timeout and was woken up to do a migrate. It restarts the `sys_rt_sigtimedwait()` after the migrate.

3.7 Priority and Capability

```
void priority(cmd,who,niceval,*tsk,*err);
int pgrp_priority(cmd,who,niceval,*ret);
int prio_user(cmd,who,niceval,*err);
int capability(cmd,pid,header,data,*reg);
int pgrp_capset(pgrp,*effective,*inherit,
                        *permitted,*ret);
int capset_all(*effective,*inherit,
                        *permitted,*ret);
```

In `sys_setpriority()` (`sys.c`), scheduling priority can be set on processes, process groups or "all processes owned by a given user." A get-priority can be done for a process or a pgrp. The `priority`, `pgrp_priority` and `prio_user` hooks are proposed to deal with distributions issues for these functions. Capability setting/getting (`sys_capset()` and `sys_capget()` in `capability.c`) are quite similar and `capability`, `pgrp_capset` and `capset_all` hooks are proposed for those functions.

3.8 Setpgid/Setsid

```
int is_process_local(pid,pgid);
int rmt_setpgid(pid,pgid,caller,sid);
int verify_pgid_session(pgid,sid);
void pgrp_update(*tsk);
void setpgid_done(*tsk,pid);
void rmt_proc_getattr(pid,*pgid,*siod);
void setsid(void);
```

Setpgid (`sys.c`) may be quite straightforward to handle in the home/master node implementations because all the process, process group and

session information will be at the home/master node. For the more peer-oriented implementations, in the most general case there could be several nodes involved. First, while the setpgid operation is most often done against oneself, it doesn't have to be, so there is a hook set early in `sys_setpgid` to move execution to the node on which the setpgid is to be done (`is_process_local` and `rmt_setpgid`). `is_process_local` can also acquire a sleep lock on the process since setpgid may not be atomic to the `tasklist_lock`. One of the tests in setpgid is to make sure there is someone in the proposed process group in the same session as the caller. If that check isn't satisfied locally, `verify_pgid_session` is called to check the rest of the process group. Given the operation is approved, the `pgrp_update` hook is called to allow the CPM implementation to adjust orphan pgrp information, to create or update any central pgrp member list and to update any cached information that might be at the process's parent's execution site (to allow him to easily do waits). A final hook in `sys_setpgid` (`setpgid_done`) is called to allow the CPM implementation to release the process lock acquired in `is_process_local`.

The `rmt_proc_getattr` hook in `sys_getpgid()` and `sys_getsid()` supplies the pgid and/or sid for processes not executing locally.

The `setsid` hook in `sys_setsid()` can be used by the CPM implementation to update cached information at the parent's execution node, at children execution nodes and at any session or pgrp list management nodes.

3.9 Ptrace

```
void rmt_ptrace(request,pid,addr,data,*ret);
*tsk rmt_find_pid(pid);
int ptrace_lock(*tsk,*tsk)
void ptrace_unlock(*tsk,*tsk);
```

Clusterwide ptrace support is not provided in all CPM implementations (eg. BProc) but can be supported with the help of a few hooks. Unfortunately `sys_ptrace()` is in the arch tree, in `ptrace.c`. The `rmt_ptrace` hook is needed if the process to be ptraced is not local. It reissues the call on the node where the process is running. In `ptrace_attach()`, in the non-arch version of `ptrace.c`, the `rmt_find_pid` hook is used in the scenario that the request was generated remotely. This hook helps ensure that the the process being traced is attached to the process debugging and not to a server daemon acting on behalf of that process. The `ptrace_lock` and `ptrace_unlock` hooks are used in `do_ptrace_unlink()` (`ptrace.c`) and `de_thread()` (`exec.c`). They can be used to provide atomicity across operations that require remote messages.

3.10 Controlling Terminal

```
void clear_my_tty(*tsk);
void update_ctty_pgrp(pgrp,npgrp);
void rmt_vhangup(*tsk);
void get_tty(*tsk,*tty_nr,*tty_pgrp);
void clear_tty(sid,*tty,flag);
void release_rmt_tty(*tsk,flag);
int has_rmt_ctty();
int rmt_tty_open(*inode,*file);
void rmt_tty_write_message(*msg,*tty);
int rmt_is_ignored(pid,sig);
```

Some CPM implementation do not support controlling terminal for processes after they move (eg. BProc). In the home-node style CPM, the task structure on the home node will have `tty` pointer. On the node where the process migrated, the task structure has no `tty` pointer. As long as any interrogation or updating using that pointer is done on the home node, this strategy works. For CPM implementations where system calls are done locally, some hooks are needed to deal with a potentially remote controlling terminal. The proposed strategy is that if the controlling terminal

is remote, the `tty` pointer would be null but there would be information in the CPM private data structure.

Daemonize(), in `exit.c`, normally clears the `tty` pointer in the task structure. Additionally it calls the hook `clear_my_tty` to do any other bookkeeping in the case where the controlling terminal is remote. In `drivers/char/tty_io.c`, the routines `do_tty_hangup()`, `disassociate_dev()`, `release_dev()` and `tiosctty()` all call the hook `clear_tty`, which clears the tty information for all members of the session on all nodes. `release_rmt_tty` is called by `disassociate_ctty()` if the tty is not local; the hook calls `disassociate_ctty()` on the node where the tty is. `get_tty` is called in `proc_pid_stat()` in `fs/proc/array.c` to gather the foreground pgrp and device id for the task's controlling terminal. The hook `update_ctty_pgrp` is called by `tiocspgrp()`, in `drivers/char/tty_io.c` and can be used by the CPM to inform all members of the old pgrp that they are no longer in the controlling terminal foreground pgrp and to inform the new pgrp members as well. Distributed knowledge of which pgrp is the foreground pgrp is important for correct behavior in the situation when the controlling terminal node crashes. `Sys_vhangup()`, in `fs/open.c`, has a call to `rmt_vhangup()` if `tty` is not set (if there is a remote tty, the CPM can call `sys_vhangup()` on that node). In `drivers/char/tty_io.c`, `tiosctty()` and `tty_open()` call the hook `has_rmt_ctty` to determine if the process already has a controlling terminal that is remote. Also in `drivers/char/tty_io.c`, the `tty_open()` function calls `rmt_tty_open` for opens of `/dev/tty` if the controlling terminal is remote. The `is_ignored()` function in `drivers/char/n_tty.c` calls `rmt_is_ignored` if it is called by an agent for a process that is actually running remotely. Finally, `rmt_tty_write_message` is called in `kernel/printk.c`, `tty_write_message()` if the tty it wants to write to is remote.

3.11 Process movement

```
int do_rexec(*char,*argv,*envp,
                    *regs,*reg);
void rexec_done();
int do_rfork(flags,stk,*regs,size,
             *ptid,*ctid,pid,*ret);
int do_migrate(*regs,signal,flags);
```

As mentioned in the goals, the hooks should allow for process movement at exec() time, at fork() time and during execution. Earlier versions of OpenSSI accomplished this via new system calls. The proposal here does not require any system calls although that is an option. For fork() and exec(), a hook is put in `do_fork()` and `do_execve()` respectively. Ops behind the hooks can determine if the operation should be done on another node. A load balancing algorithm can be consulted or the process could have been marked (eg. via a procfs file like `/proc/<pid>/goto`) for remote movement. An additional hook, `rexec_done` is provided so the CPM implementation can get control after the exec on the new node has completed but before returning to user mode, so that process setup can be completed and the original node can be informed that the remote execve() was successful.

A single hook is needed for process migration. The proposed mechanism is that via `/proc/<pid>/goto` or a load balancing subsystem, processes have `TIF_MIGPENDING` flag (added flag in flags field of `thread_info` structure) set if they should move. That flag is checked just before going back to user space, in `do_notify_resume()`, in `arch/xxx/kernel/signal.c` and calls the `do_migrate` hook. Checkpoint and restart can be invoked via the same hook (migrate to/from disk).

Determining if these hooks are sufficient to allow an implementation that satisfies the goals and requirements outlined earlier is best done by implementing a CPM using the hooks. The OpenSSI 3.0 CPM, which provides almost all the requirements, including optional ones, has been adapted to work via the hooks described above. Work to ensure that other CPM implementations can also be adapted needs to be done. The OpenSSI 3.0 CPM design is described in the next section.

4 Clusterproc Design for OpenSSI

In this section we describe a Cluster Process Management (CPM) implementation adapted from OpenSSI 2.0. It is part of a functional cluster which is a subset of OpenSSI. The subset does not have a cluster filesystem, a single root or single init. It does not have clusterwide device naming, a clusterwide IPC space or a cluster virtual ip. It does not have connection or process load balancing. All those capabilities will be subsequently added to this CPM implementation to produce OpenSSI 3.0.

To allow the CPM implementation to be part of a functional cluster, several other cluster components are needed. A loadable membership service is needed, together with an intra-node communication service layered on tcp sockets. To enable the full ptrace and remote controlling terminal support, a remote copy–to/from–user capability is needed. Also, a set of remote file ops is needed to allow access to remote controlling terminals. Finally, a couple of files are added to `/proc/<pid>` to provide and get information for CPM. Implementations of all needed capability are available and none require significant hooks. Like the clusterproc hooks, however, these hooks must be studied and included if they are general and allow for different implementations.

In this section we describe the process id and process tracking design, the module initialization, and per process private data. Then we describe how all the process relationships are managed clusterwide, followed by sections on `/proc` and process movement.

4.1 Process Ids and Process Tracking

As in OpenSSI, process ids are created by first having the local base kernel generate a locally unique id and then, using the hooks, adding the local node number in the higher order bits of the pid. This is the only pid the process will have and when the pid is no longer in use, the locally unique part is returned to the pool on the node it was generated on. The node who generated the process id (creation or origin node) is responsible for tracking if the process still exists and where it is currently running so operations on the process can be routed to the correct node and so ultimately the pid can be reused. If the origin node leaves the cluster, tracking is taken over by a designated node in the cluster (surrogate origin node) so processes are always findable without polling.

4.2 Clusterproc Module Initialization and Per Process Clusterproc Data Structure

The clusterproc module is loaded during the ramdisk processing although it could be done later. It assumes the membership, intra-node communication remote copy-in/copy-out and remote file ops modules are already loaded and registers with them. It sets up its data structures and installs the function pointers in the clusterproc op table. It also allocates and initializes clusterproc data structures for all existing processes, linking the structures into the task structure. After this initialization, each new process created will get a private clusterproc data structure via the `fork_alloc` hook.

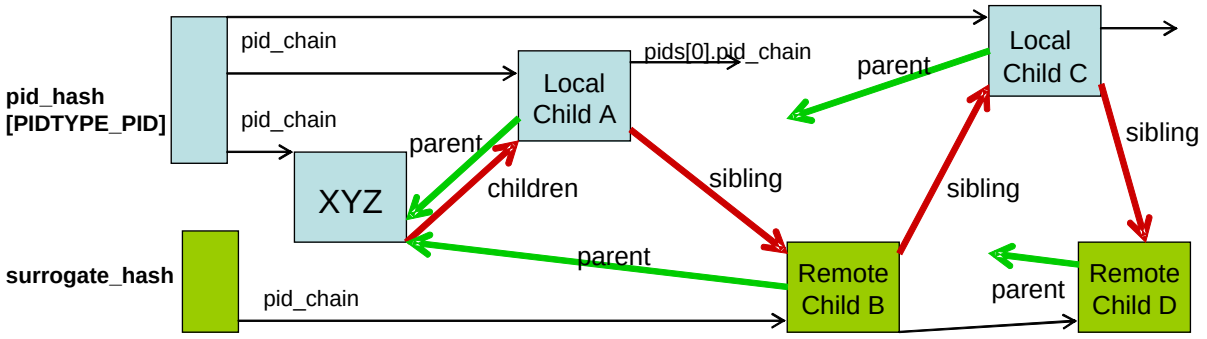

Figure 1: Parent xyz's execution node

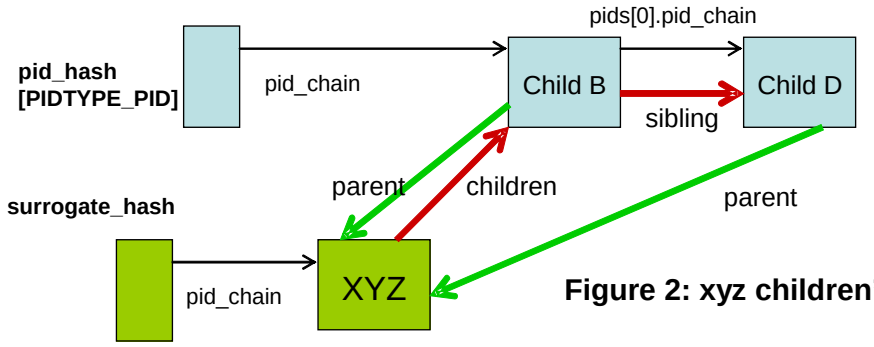

Figure 2: xyz children's execution node

4.3 Parent/Child/Ptrace Relationships

To minimize hooks and changes to the base Linux, the complete parent/child/sibling relationship is maintained at the current execution node of the parent, using surrogate task structures for any children that are not currently executing on that node. Surrogate task structures are just struct `task_struct` but are not hashed into the any of the base pid hashes and thus only visible to the base in the context of the parent/child relationship. Surrogate task structures have cached copies of the fields the parent will need to execute `sys_wait()` without having to poll remote children. The reap operation does involve an operation to the child execution node. The `update_parent` hook is used to maintain the caches of child information. For each node that has children but no parent, there is a surrogate task structure for the parent and a partial parent/child/sibling list. Surrogate task structures are hashed off a hash header private to the CPM module. Figure 1 shows how parent process XYZ is linked to his children on his execution node and Figure 2 shows the structures on a child node where XYZ is not executing.

Ptrace parent adds some complexity because a process's parent changes over time and `real_parent` can be different from `parent`. The `update_parent` hook is used to maintain all the proper links on all the nodes.

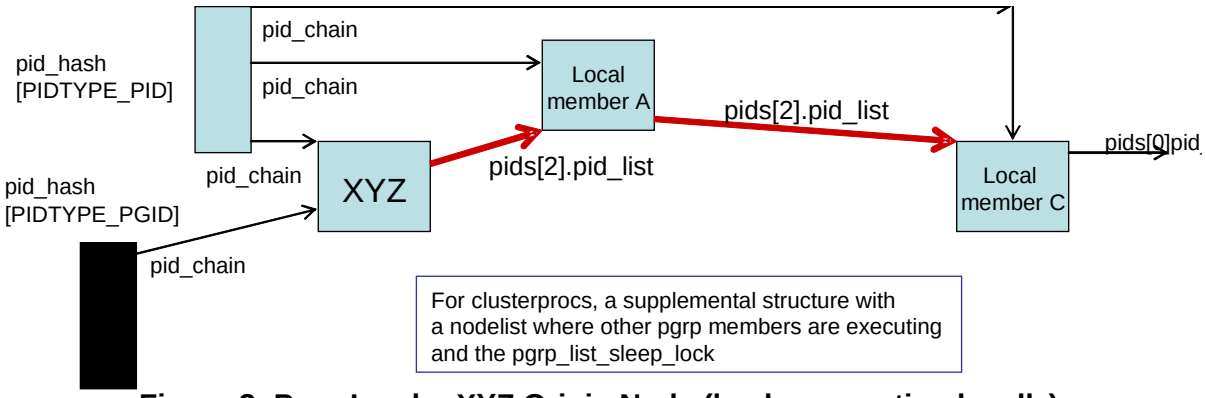

Figure 3: Pgrp Leader XYZ Origin Node (leader executing locally)

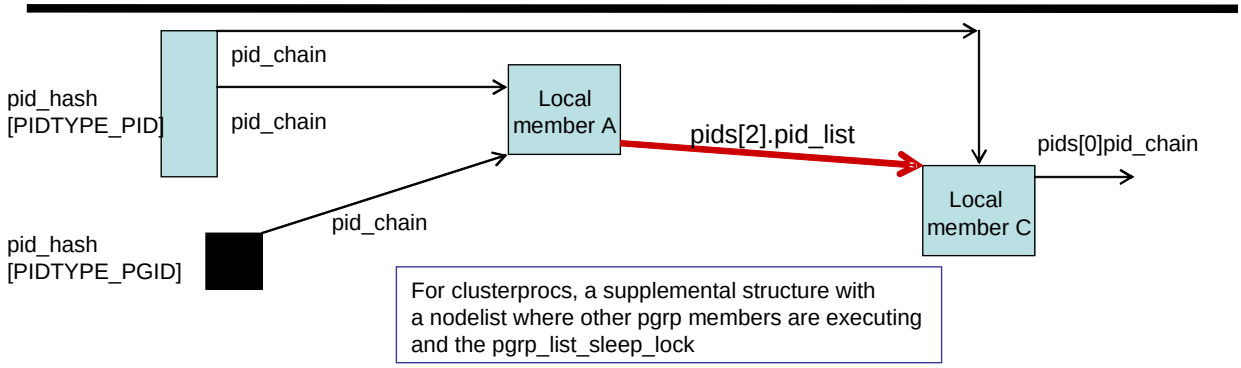

Figure 4: Pgrp Leader XYZ Origin Node (leader not executing locally)

4.4 Process Group and Session Relationships

With the process tracking described above, actions on an individual process is pretty straightforward. Actions on process groups and sessions are more complicated because the members may be scattered. For this CPM implementation, we keep a list of nodes where members are executing on the origin/surrogate origin node for the pid that is the name of the pgrp or session. On that origin node any local members are linked together as in the base but an additional structure is maintained that records the other nodes where members are on. This structure also has a sleep lock in it to make certain pgrp or session operations are atomic. Figures 3 and 4 shows the data structure layout on the origin node with and without the pgrp leader executing on that node. As with process tracking, this origin node role is assumed by the surrogate origin if the origin node fails and is thus not a single point of failure.

Operations on process groups are directed to the origin node (aka the list node). On that node the operation first gets the sleep lock. Then the operation can be done on any locally executing members by invoking the standard base code. Then the operation is sent to each node in the node list and the standard base operation is done for any members on that node.

A process group is orphan if no member has a parent in a different pgrp but with the same session id (sid). Linux needs to know if a process group is orphan to determine if processes

can stop (SIGTSTP, SIGTTIN, SIGTTOU). If a process group is orphan, they cannot. Linux also needs to know when a process group becomes orphan, because at that point any members that are stopped get SIGHUP and SIGCONT signals. A process exit might effect its own pgrp and the pgrp on all its children, which could involve looking at all the pgrp members (and their parents) of all the pgrps of all the exiting process's children. When all the pgrps and processes are distributed, this could be very expensive. The OpenSSI CPM, through the described hooks, has each pgrp list cache whether it is orphan or not, and if not, which nodes have processes contributing to its non-orphaness. Process exit can locally determine if it necessary to update the current process's pgrp list. Each child must be informed of the parents exit, but they can locally determine if they have to update the pgrp list orphan information. With a little additional information this mechanism can survive arbitrary node failures.

4.5 Controlling Terminal Management

In the OpenSSI cluster, the controlling terminal may be managed on a node other than that of the session leader or any of the processes using it. There is a relationship in that processes need to know who their controlling terminal is (and where it is) and the controlling terminal needs to know which session it is associated with and which process group is the foreground process group.

In the base linux, processes have a `tty` pointer to their controlling terminal. The `tty_struct` has a `pgrp` and a `session` field. In clusterproc, the base structures are maintained as is, with the pgrp and session fields in the tty structure and the tty pointer in the task's signal structure. The tty pointer will be maintained if the tty is local to the process. If the tty is not local, the clusterproc structure will have cttynode and cttydev fields to allow CPM code to determine if and where the controlling terminal is. To avoid hooks in some of the routines being executed at the controlling terminal node, svrprocs (agent kernel threads akin to nfsd's) doing opens, ioctls, reads and writes of devices at the controlling node will masquerade as the process doing the request (pid, pgrp, session, and tty). To avoid possible problems their masquerading might cause, svrprocs will not be hashed on the `pid_hash[PIDTYPE_PID]`.

4.6 Clusterwide `/proc`

Clusterwide `/proc` is accomplished by stacking a new pseudo filesystem (`cprocfs`) over an unmodified `/proc`. Hooks may be needed to do the stacking but will be modest. In addition, a couple of new files are added to `/proc/<pid>`—a `goto` file to facilitate process movement and a `where` file to display where the process is currently executing. The proposed semantics for `cprocfs` would be that:

- readdir presents all processes from all nodes and other proc files would either be an aggregation (sysvipc, uptime, net/unix, etc.) or would pass thru to the local `/proc`
- `cprocfs` function ships all ops on processes to the nodes where they are executing and then calls the `procfs` on those nodes;
- `cprocfs` inodes don't point at task structures but at small structures which have hints as to where the process is executing.
- `/proc/node/#` directories are redirected to the `/proc` on that node so one can access all the hardware information for any node.

- readdir of /proc/node/# only shows the processes executing on that node.

4.7 Process Movement

Via the hooks described earlier, the OpenSSI CPM system provides several forms of process movement, including a couple of forms of remote exec, an rfork and somewhat arbitrary process migration. In addition, these interfaces allow for transparent and programmatic checkpoint/restart.

The external interfaces to invoke process movement are library routines which in turn use the /proc/<pid>/goto interface to affect how standard system calls function. Writes to this file would take a buffer and length. To allow considerable flexibility in specifying the form of the movement and characteristics/functions to be performed as part of the movement, the buffer consists of a set of stanzas, each made up of a command and arguments. The initial set of commands is: rexec, rfork, migrate, checkpoint, restart, and context, but additional commands can be added. The arguments to rexec, rfork and migrate() are a node number. The argument to checkpoint and restart are a pathname for the checkpoint file. The context command indicates whether the process is to have the context of the node it is moving to or remain the way it was. Do_execve() and do_fork() have hooks which, if clusterproc is configured, will check the goto information that was stored off the clusterproc structure, and if appropriate, turn an exec into an rexec or a fork into an rfork.

The goto is also used to enable migrations. Besides saving the goto value, the write to the goto sets a new bit in the thread_info structure (TIF_MIGPENDING). Each time the process leaves the kernel to return to user space (did a system call or serviced an interrupt), the do_notify_resume() function is called if any of the flags in thread_info.flags are set (normally there are none set). The function do_notify_resume() now has a hook which will check for the TIF_MIGPENDING flag and if it is set, the process migrates itself. This hook only adds pathlength when any of the flags are set (TIF_SIGPENDING, etc.), which is very rarely.

OpenSSI currently has checkpoint/restart capability and this can be adapted to use the goto file and migration hook. Two forms of kernel-based checkpoint/restart have been done in OpenSSI. The first is transparent to the process, where the action is initiated by another process. The other is when the process is checkpoint/restart aware and is doing the checkpoint on itself. In that case, the process may wish to "know" when it is being restarted. To do that, we propose that the process open the /proc/self/goto file and attach a signal and signal handler to it. Then, when the process is restarted, the signal handler will be called. Checkpoint/restart are variants of migrate. The argument field to the goto file is a pathname. In the case of checkpoint, the TIF_MIGPENDING will be set and at the end of the next system call, the process will save its state in the filename specified. Another argument can determine whether the process is to continue or destroy at that point. Restart is done by first creating a new process and then doing the "restart" goto command to populate the new process with the saved image in the file which is specified as an argument.

A more extensive design document is available on the OpenSSI web site[9].

5 Summary

Process management in Linux is a complicated subsystem. There are several differ-

ent relationships—parent/child, process group, session, thread group, ptrace parent and controlling terminal (session and foreground pgrp). There are some intricate rules, like orphan process groups and `de_thread` with ptrace on the thread group leader. Making all this function in a completely single system way requires quite a few different hook functions, as defined above (some could be combined to reduce this number), but there is no performance impact and the footprint impact on the base kernel is very small (patch file touches 23 files with less than 500 lines of total changes, excluding the new `clusterproc.h` file).

References

[1] Popek, G., Walker, B. *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[2] Barak, A., Guday, S., Wheeler, R. *The MOSIX Distributed Operating System, Load Balancing for UNIX* volume 672 of Lecture Notes in Computer Science, Spinger-Verlag, 1993

[3] `http://www.openssi.org`

[4] `http://www.openmosix.org`

[5] `http://bproc.sourceforge.net`

[6] Valle'e, G., Morin, C., et.al., *Process migration based on gobelins distributed shared memory*, in proceedings of the workshop on Distributed Shared Memory (DSM'02) in CCGRID 2002, pg. 325–330, IEEE Computer Society, May 2002.

[7] Private communication

[8] `http://www.cassatt.com`

[9] `http://openssi.org/proc-hooks/proc-hooks.pdf`

[10] `http://openssi.org/ssi-intro.pdf`

Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*