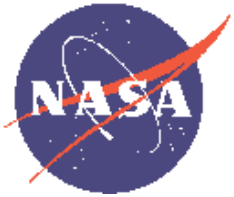
BPROC: A distributed PID space for Beowulf clusters

http://beowulf.gsfc.nasa.gov/software/bproc.html
http://www.beowulf.org/software/bproc.html

Erik Hendriks

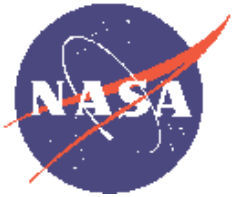
<hendriks@cesdis.gsfc.nasa.gov>



May 1999

What is BPROC?

- **BPROC = Beowulf distributed PROCess space.**
- **A distributed process space which allows processes in the front end's process table to be physically present on other machines in the cluster.**
- **A process migration mechanism to help create those remote processes.**

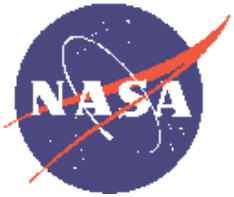


What is a Beowulf?

- A software distribution for building high-performance computing systems
- Dedicated clusters
- Private internal network
- Off-the-shelf commodity hardware
- Open source infrastructure

What is a Distributed PID Space?

- Every machine has a pool of process IDs (PIDs) to allocate to new processes.
- Every machine has a single process tree.
- A distributed PID space allows processes in one machine's process tree to exist on other machines.



Motivation

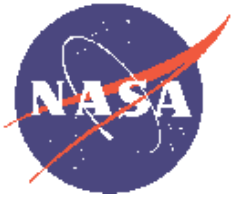
- I hate rsh, rlogin, etc. Remote process creation with BPROC should be a better alternative.
- BPROC will let us present the state of the machine (process-wise) in a single location making a Beowulf seem more like a single system. (single system image)
- BPROC will allow us to reduce the required software for every node.

BPROC setup

- Machines which want to distribute their PID space run a master daemon.
- Machines accepting processes from other machines run a slave daemon.
- Each machine can run multiple slave daemons but only one master daemon.

Starting remote processes, BPROC style

- bproc_rexec(node,file,argv,envp)
  - Semantics similar to exec'ing rsh.
- bproc_move(node,flags)
  - Takes the current process and moves it to another node. Note that open files, SysV IPC, etc are not preserved.
- bproc_execmove(node,flags)
  - Process does an exec followed by a move.
  - A way to run an ordinary binary on a remote host even if the binary is not installed on the remote host.



Starting remote processes, BPROC style

- bproc_rfork(node,flags)
  - A lot like move except that the process spawns a child on remote node. (Same semantics as fork())

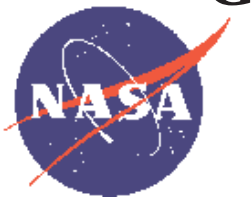
```
if(bproc_rfork(node,flags)==0){
    /* child things (remote) */
} else {
    /* parent things */
}
```

How Does BPROC Work?

- **BPROC consists of 4 basic parts:**
  - **Ghost processes on the front end**
  - **Process ID masquerading on the remote nodes**
  - **Virtual Memory Area dumper (VMADump) to transport processes.**
  - **A master and slave daemon to tie all the pieces together.**

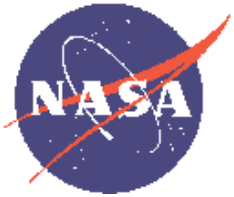
Ghost Processes

- Ghosts are place holders in the process tree on the front end.
- Every remote job has a corresponding ghost process on the front end and vice versa.
- Ghost are real processes which resemble kernel threads like kflushd and kswapd.
  - Ghosts have no memory space.
  - Ghosts have no open files.
  - Ghosts have no file system context.
  - Ghosts do have signal handlers.



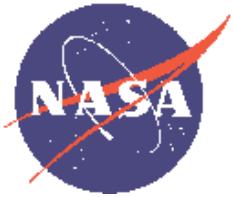
Ghost Processes

- Ghosts can wake up and run locally.
- Ghosts do the following:
  - Forward the signals they receive.
  - Fork when requested.
  - Wait() when requested.
  - Exit with appropriate exit status.
  - That's It!



Ghost Processes

- Ghosts mirror status information of remote processes.
- This information is presented in /proc in place of the ghosts process's real process information.
- Remote process information is updated on demand and no more than once every *n* seconds.
- Update requests take place in context of the process reading /proc.

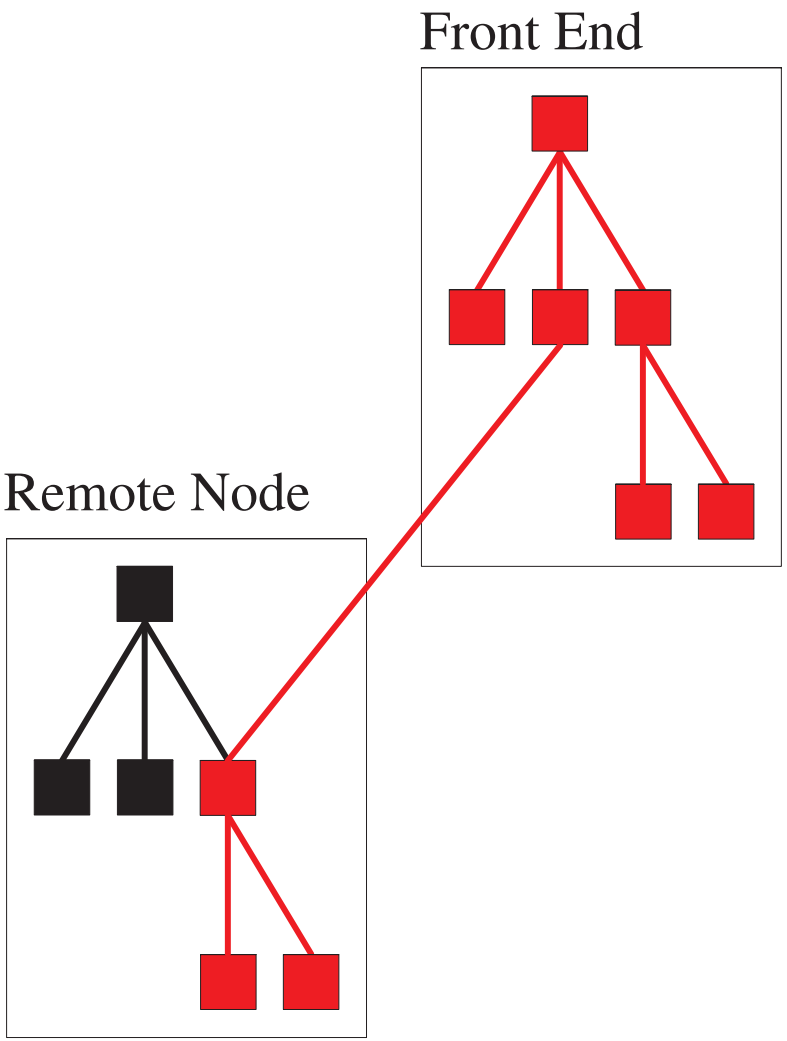


Ghost Processes

- **In-kernel message queues.**
  - **Every ghost has a message queue.**
  - **Another queue for messages going to the master daemon.**
- **Interface to user space**
  - **Master Daemon: /dev/bproc_ghost**
  - **Clients: System call sys_bproc**

Process ID Masquerading

- Remote nodes run a piece of the front end's process tree.
- Remote nodes have a PID space like any other machine.
- Remote jobs must co-exist with other processes.



Process ID Masquerading

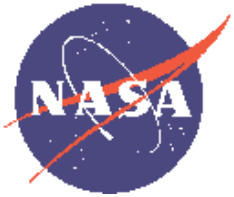
- **When creating processes on remote nodes, the following problems arise:**
  - **When moving a process to a new machine its process ID should not appear to change. (And we cannot guarantee that a given ID will be available on the remote machine.)**
  - **Signals should not always be delivered across PID spaces.**
  - **Not everything can be completely handled locally.**

Process ID Masquerading

- **Solution: Attach an extra PID and PPID to processes and modify the handful of PID related syscalls.**
- **Remote processes are assigned an arbitrary local pid and given the PIDs from the master's PID space as masqueraded PIDs.**
- **For PID related syscalls:**
  - **Arguments are mapped between PID spaces appropriately.**
  - **Return values are always in the master's PID space.**

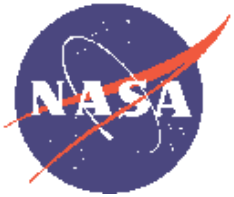
Process ID Masquerading System Calls

- getpid(), getppid(): are straight forward
- fork(): Requires assistance of master node to allocate a new masqueraded process ID for the child process.
- wait(): Arguments are mapped to real PIDs. The result is mapped back.
- kill(): Processes can only signal other processes in the same PID space. The signal may be forwarded.



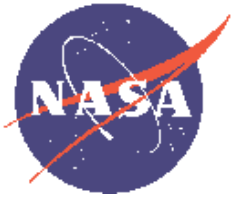
Process ID Masquerading

- **Requests that cannot be handled locally are forwarded to the slave daemon.**
- **Drawbacks:**
  - **fork() will block halfway through new process creation**
  - **kill() can block**



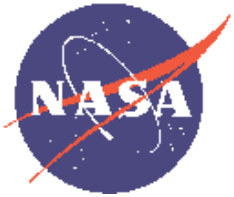
Process ID Masquerading

- **In-kernel message queues much like those for the ghost system.**
- **Interface:**
  - **Slave daemon: /dev/bproc_masq**
  - **Setting new masqueraded PID's is done via a write to this device.**



BPROC Daemons
Connecting it all together

- **Master daemon:**
  - Holds all machine configuration information.
  - Knows where every process is.
  - Acts as a message router.
- **Slave daemon:**
  - Forks to create new remote processes.
  - Acts like a message pipe between the kernel and the remote daemon.
  - Has no knowledge of where other processes exist.



BPROC Daemons
Connecting it all together

- **Ghost and PID masquerading both communicate with the daemons via a misc character device.**
  - **This allows us to notify daemons of new messages via select().**
  - **Avoids adding a system call to the already crowded system call table.**
- **The character device interface is not suitable for doing process migration so a syscall had to be added anyway. (oh well)**

Process Migration (VMADump)

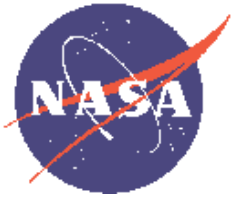
- Virtual Memory Area Dump (VMADump)
- Stores copies of all memory mapped regions to a user supplied file descriptor.
- Stores signal handler states.
- Memory mapped files can be stored as references to files rather than data.
  - Executable (as flagged in struct vm_area_struct)
  - Libraries : things in /lib, /usr/lib
  - Other : any other mmap'ed file

Process Migration (VMADump)

- Interface (via syscall):
  - vma_dump(filedescriptor, flags)
  - vma_undump(filedescriptor)
- A syscall is necessary for this functionality.
- Dumping and undumping involves only writing and reading (respectively).
- (Un)dumping will work on any file which allows read or write - including sockets!

Process Migration (VMADump)

- VMADump has no knowledge of binary formats.
- VMADump module adds a binary format which allows undumping of an image stored in a file without extra utilities.



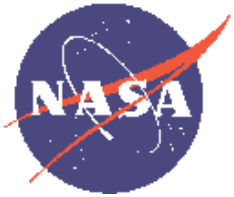
Gratuitous Screen Shot

```
  3:08pm  up  2:25,  7 users,  load average: 0.13, 0.07, 0.07
175 processes: 46 sleeping, 129 running, 0 zombie, 0 stopped
CPU states: 12798.7% user,  8.3% system,  0.0% nice,  0.0% idle
Mem:  128188K av,  57476K used,  70712K free,  23852K shrd,  17168K buff
Swap: 130748K av,      0K used, 130748K free                 23184K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
 1540 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:35 cpumunch
 1541 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:35 cpumunch
 1542 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:35 cpumunch
 1543 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:35 cpumunch
 1544 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:35 cpumunch
 1545 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:35 cpumunch
 1546 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
 1547 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
 1548 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
 1549 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
 1550 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
 1551 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
 1552 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
 1553 hendriks   0   0     0    0     0 RW      0 99.9  0.0  21:34 cpumunch
```

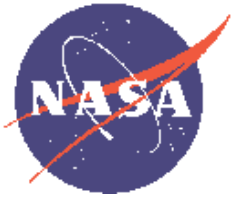
BPROC vs other stuff

- **Previous Beowulf global PID space**
  - **Was a scheme to allocate PIDs to machines.**
  - **Allowed for remote signalling**
  - **No single view of system**
- **Condor**
  - **A resource allocation / load balancing system designed for many machines with distributed ownership.**
  - **Includes user level migration and Mosix-like syscall forwarding.**



BPROC vs MOSIX Process Migration

- **BPROC:**
  - **Non premptive: Processes decide if, when and where they will migrate.**
  - **System image is not preserved, open files are lost.**
  - **No added overhead after moving for any system calls other than fork() and kill().**
- **MOSIX:**
  - **Premptive migration: Processes can be automatically moved by a load balancer.**
  - **System image is preserved.**
  - **High overhead for most system calls after migration.**



Some Problems

- BPROC relies heavily on unique PIDs. As do most things in the kernel. This will likely collide badly with the CLONE_PID flag.
- Scaling: BPROC adds a process table entry for each remote process.
- Details: BPROC was first announced just over a month ago. It's alpha and has many rough edges.
- BUGS: The things I can't call "details". See Details above.

BPROC's Development Roadmap

- Development of libbproc.
- Message passing libraries have to be modified to use BPROC's process creation mechanisms.
- Improved (more complete) status information on the front end.
- Improved error recovery when undumping with VMADump.
- Improved IO forwarding scheme.

Things I'd like to see

- A BPROC-centric node install scheme with a super-small software footprint.
- ptrace-able ghosts. Wouldn't it be great to run be able to run gdb or strace on a ghost?