

Bproc kickstart

Nicholas Mc Guire

Opentech EDV-Research GmbH

June 12, 2004

Contents

1. System preparation	1
1.1. Patching the kernel	1
1.2. Configuring	2
1.3. Compiling	2
1.4. Installing	3
1.5. Checking the new system	5
2. Starting the ghosts	6
2.1. Compiling user-space	6
2.2. Starting in all up	7
2.2.1. Setting up the daemon	7
2.2.2. Slave setup	10
2.2.3. Testing	11
2.3. Bproc management apps	12
2.3.1. /usr/bin/bpsh	12
2.3.2. /usr/bin/bpstat	12
2.3.3. /usr/bin/bplib	13
2.3.4. /usr/bin/bpcp	13
2.3.5. /usr/bin/bpctl	13
2.4. A few error cases	14
3. Simple examples	16
4. List of Acronyms	19

Contents

Version	Author	Date	Comment
1.0	Nicholas Mc Guire	28 May 2004	First shot

1. System preparation

Note: The bproc distribution comes with a kernel patch - in the 3.2.6 release it was for the 2.4.21 kernel - as Slackware uses 2.4.22 by default we applied the bproc patch to 2.4.22 - looks like this is fine - but if you run into problems use the "official" bproc kernel.

1.1. Patching the kernel

```
hofrat@rtl21:~$ tar -xzf bproc-3.2.6.tar.gz
hofrat@rtl21:~$ mkdir linux
hofrat@rtl21:~$ cd linux/
```

download linux-2.4.22.tar.bz2 via ftp from:

```
ftp://ftp.kernel.org/pub/linux/kernel/v2.4/
```

```
hofrat@rtl21:~/linux$ tar -xjf linux-2.4.22.tar.bz2
hofrat@rtl21:~/linux$ cd linux-2.4.22
hofrat@rtl21:~/linux/linux-2.4.22$ patch -p1 --dry-run < /home/hofrat/bproc-3.2.6/patches
patching file fs/exec.c
Hunk #2 succeeded at 953 (offset 22 lines).
patching file fs/binfmt_script.c
patching file fs/binfmt_elf.c
patching file fs/proc/base.c
Hunk #2 succeeded at 341 (offset 36 lines).
Hunk #3 succeeded at 959 (offset -6 lines).
```

The lines starting with "Hunk ..." indicate that the code sequence that needed changing was not at the same location that the original authors expect, this is due to us patching the 2.4.22 kernel (which is the default on Slackware 9.1) with the bproc patch for 2.4.21 (which was the latest at time of writing). Check the bproc home page for later patches, and don't forget to check the lam home page for the latest bproc version supported by lam (at time of writing bproc 4 was being discussed on the lam mailing list but not yet available).

The patch command above did not yet patch anything - it just did a dry-run - so now that we know nothing will fail, we can do the actual patch. Note though that it well may be that a patch that applies still does not work properly - so if you want to be absolutely sure, then use the 2.4.21 kernel not the 2.4.22 !

```
hofrat@rtl21:~/linux/linux-2.4.22$ patch -p1 < /home/hofrat/bproc-3.2.6/patches/bproc-pa
```

1.2. Configuring

We assume that you have a working kernel configuration for your system and all you need to do here is to add the configuration for bproc - if this is not the case then you need to add in your hardware specific settings along!

```
hofrat@rtl21:~/linux/linux-2.4.22$ make menuconfig
```

```
...
General setup  --->
  ...
  [*] Beowulf Distributed Process Space (NEW)
  ...
```

```
    < Exit >
```

```
  < Exit >
```

```
Do you wish to save your new kernel configuration?
```

```
    < Yes >
```

That's it for configuring bproc support. You should store your configuration file in a safe place so that you can later rebuild the system - we simply copy it to config_bproc so we know what this is - consult your configuration management plan (CMP) on where to put it in the repository.

```
hofrat@rtl21:~/linux/linux-2.4.22$ cp .config config_bproc
```

1.3. Compiling

Before you proceed with the compilation and installation we must make a little change to the top level Makefile, the bproc patch leaves the kernel's extraversal unchanged, so we would now overwrite our modules in /lib/modules/2.4.22/ with the new set - which is not what we want - we always want to have the fallback installation that was unmodified to cross check any problems !

So we set the following in the Makefile:

```
hofrat@rtl21:~/linux/linux-2.4.22$ vi Makefile
...
EXTRAVERSION = -bproc
..
```

Now all modules will go into `/lib/modules/2.4.22-bproc` and we have no collision, furthermore when we do a `uname -a` on the system later we could not tell the bproc patched kernel from the unmodified kernel (other than by inspecting the symbol table or remembering the date of compilation...).

The rest is standard procedure for the kernel compilation:

```
hofrat@rtl21:~/linux/linux-2.4.22$ make dep
hofrat@rtl21:~/linux/linux-2.4.22$ make modules
hofrat@rtl21:~/linux/linux-2.4.22$ make bzImage
...
Boot sector 512 bytes.
Setup is 2516 bytes.
System is 1105 kB
warning: kernel is too big for standalone boot from floppy
make[1]: Leaving directory '/home/hofrat/linux/linux-2.4.22/arch/i386/boot'
hofrat@rtl21:~/linux/linux-2.4.22$
```

This will build it all (kernel and modules), but not install it as the installation can only be performed by the root user. Note the above warning about the kernel being too big - it only means that this kernel will not be able to boot from a floppy - you can safely ignore it in our setup.

1.4. Installing

To install we become root using `su`, note that we don't use `su -`, that is we don't want to have roots environment at this point, only the privileges.

```
hofrat@rtl21:~/linux/linux-2.4.22$ su
Password:
root@rtl21:/home/hofrat/linux/linux-2.4.22# make modules_install
root@rtl21:/home/hofrat/linux/linux-2.4.22# cp arch/i386/boot/bzImage
/boot/bproc
root@rtl21:/home/hofrat/linux/linux-2.4.22# cp System.map
/boot/System.map.bproc
```

1. System preparation

The system map is not really needed for normal operation, but if we get some oops and need to decode it we will need it!

Now that the kernel and modules are in place we need to set up the lilo configuration file `/etc/lilo.conf`. We will add an entry for the new kernel, and not make it the default (yet).

We can simply copy the original Slware entry for Linux and edit it - you should end up with the new entry looking like:

```
image = /boot/vmlinuz
  root = /dev/hda2
  label = Linux
  read-only
image = /boot/bproc
  root = /dev/hda2
  label = bproc
  read-only
```

Next we run the `lilo` command, which should give us:

```
root@rtl21:/home/hofrat/linux/linux-2.4.22# lilo
Added Linux *
Added bproc
```

To boot our new kernel on the node - which is assumed to have no screen - for test purposes we use:

```
root@rtl21:/home/hofrat/linux/linux-2.4.22# lilo -R bproc
```

This will instruct lilo to boot the bproc kernel the next time power is cycled, but leave the defaults unchanged, so if the box does not come on-line again you can power cycle it again and it will boot the default linux again. So let's now boot into the new kernel:

```
root@rtl21:/home/hofrat/linux/linux-2.4.22# reboot
```

```
Broadcast message from root (pts/1) (Sun May 30 12:45:41 2004):
```

```
The system is going down for reboot NOW!
```

1.5. Checking the new system

After the box comes on-line again we log in as root (or as user and `su -` to root).

```
piglet:~ # ssh -l hofrat rtl21
hofrat@rtl21's password:
Warning: Remote host denied X11 forwarding, perhaps xauth program could not be run on the
Last login: Sun May 30 12:46:47 2004 from piglet.hofr.at
Linux 2.4.22-bproc.
```

The first change you should notice is that the system reports Linux 2.4.22-bproc, and `uname -a` gives us

```
Linux rtl21 2.4.22-bproc #1 Sun May 30 12:35:02 CEST 2004
i686 unknown unknown GNU/Linux
```

After becoming root we can check the kernel symbols and those will show us that bproc is actually successfully compiled in:

```
root@rtl21:~# ksyms -a | grep bproc_hook
c02fc8a0 bproc_hook_do_execve_hook
c02fc8a4 bproc_hook_load_script_hook
c02fc8a8 bproc_hook_refresh_status_hook
c02fc8ac bproc_hook_get_task_state_hook
c02fc8b0 bproc_hook_proc_pid_hook
c02fc8b4 bproc_hook_proc_ppid_hook
c02fc8c0 bproc_hook_proc3_hook
...
```

So these are the hooks for the gost process' used to manage the remote apps on the local system.

Now we need to set up bproc on at least one more system so we can play with remote execution via migrated processes. Basically the same steps as above apply, if you have two **identical systems then you can copy the kernel setup to the second system. To do that cleanly we recommend transferring the kernel tree and rerunning the install sections:**

Below rtl21 is the system we built it on already and rtl20 is an identical system with Slackware installed but no bproc yet.

2. Starting the ghosts

```
hofrat@rtl21:~$ tar -cf - linux | ssh -l hofrat rtl20 tar -xf -
```

Don't use scp or the links will be broken - this is a bit of a lengthy command but it will do the job. If you don't like that you can make a tar file of the kernel tree you have and scp it to the second host and then unpack it there. Any way - once you got the tree copied to rtl20 then you can log in there and simply rerun the install steps

```
hofrat@rtl21:~$ ssh -l root rtl20
root@rtl20's password:
root@rtl20:~# cd /home/hofrat/linux/linux-2.4.22
root@rtl20:/home/hofrat/linux/linux-2.4.22# make bzImage
root@rtl20:/home/hofrat/linux/linux-2.4.22# make modules_install
```

Edit /etc/lilo.conf as noted above on the second box and reboot it aswell.

2. Starting the ghosts

Well not quite yet - the system(s) are configured but we don't have the libs and apps ready. The above setup added the kernel level management to the system, now we need to configure and compile the user-side.

2.1. Compiling user-space

```
hofrat@rtl21:~$ cd bproc-3.2.6
hofrat@rtl21:~/bproc-3.2.6$ vi Makefile.conf
# Hey emacs, this is a -*- makefile -*-.
# $Id: content.tex,v 1.5 2004/06/04 12:51:31 hofrat Exp $
LINUX=/home/hofrat/linux/linux-2.4.22
```

That tells the bproc build process where to find the pathed kernel, that should be all you need to configure - now its just a make to build it. There seems to be a minor glitch in bproc-3.2.6 that causes make install to fail, below is a "quick-and-dirty" work around.

```
...
make[1]: *** No rule to make target '/usr/src/linux/include/linux/bproc.h', needed by 'h
make[1]: Leaving directory '/home/hofrat/bproc-3.2.6/kernel'
make: *** [install-modules] Error 2
```

2. Starting the ghosts

```
root@rtl21:/home/hofrat/bproc-3.2.6# cd /usr/src
root@rtl21:/usr/src# rm linux
root@rtl21:/usr/src# ln -s /home/hofrat/linux/linux-2.4.22 linux
root@rtl21:/usr/src# cd /home/hofrat/bproc-3.2-6#
root@rtl21:/home/hofrat/bproc-3.2.6# make install
...
mkdir -p /usr/include/sys
install -m 644 vmadump.h /usr/include/sys
make[1]: Leaving directory '/home/hofrat/bproc-3.2.6/vmadump'
root@rtl21:/home/hofrat/bproc-3.2.6# ldconfig
```

Looks better - bproc is now installed ! You could reboot, the Slacware would set up the linkern loader config or you simply run ldconfig to update it. Last we check if the new bproc lib is accessible:

```
root@rtl21:/home/hofrat/bproc-3.2.6# ldconfig -p | grep bproc
libproc.so.2.0.16 (libc6) => /lib/libproc.so.2.0.16
libbproc.so.2 (libc6) => /usr/lib/libbproc.so.2
```

That should be ok now.

2.2. Starting in all up

The kernel is configure, the libs are in place, now we are left with the environment and the configuration - and then we can bother with executables :)

2.2.1. Setting up the daemon

The bproc daemon needs to be running on both systems, it's main responsibilities are:

- forwarding signals
- returning status
- redirect I/O (i.e. stdin,stdout,stderr)

2. Starting the ghosts

On the slave nodes, the ones that execute the actual tasks, the `bpslave` must be running. On the master node, the one that manages the remote aps via the local ghosts, the `bpmaster` daemon must be running.

The daemon is already compiled (it was compiled with the above `make`) we need to now configure the hosts. The default configuration file is located in `/etc/beowulf/config` - but before setting up the system wide config file we will test it by passing it to the `bpmaster/bpslave` on the command line.

We need to collect some information about the nodes first and put that together into the config file:

- The masters interface to use:
Our test system only has one interface - so that is `eth0`, if you don't know which interface is used to reach the slave node, you can use:

```
root@rtl21:~# route -n
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
192.168.1.0      0.0.0.0         255.255.255.0  U        0      0      0 eth0
0.0.0.0          192.168.1.8    0.0.0.0        UG       1      0      0 eth0
```

Which in our case shows us that `192.168.1.X` is on `eth0` with the slave being at `192.168.1.40` we know the masters interface is `eth0` - now we grab the interface information with:

```
root@rtl21:~# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:30:1B:30:74:4C
          inet addr:192.168.1.41  Bcast:192.168.1.255  Mask:255.255.255.0
          ...
```

- Number of nodes:
That's simple here - it's only one for now - the master node is not counted (note that some of the examples will not run in a one-slave setup!).
- IP Range:
This is a list of IP addresses that you want to assign to the nodes that are going to receive processes via `bproc`. We only have one slave node so our range is a single address.

2. Starting the ghosts

- MAC address:

```
root@rtl21:~# ping -c 1 rtl20
PING rtl20.hofr.at (192.168.1.40) 56(84) bytes of data.
64 bytes from rtl20.hofr.at (192.168.1.40): icmp_seq=1 ttl=64 time=0.121 ms

--- rtl20.hofr.at ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.121/0.121/0.121/0.000 ms
root@rtl21:~# arp
Address                  HWtype  HWaddress          Flags Mask            Iface
rtl20.hofr.at            ether    00:30:1B:2F:F4:4E  C                     eth0
root@rtl21:~#
```

So rtl20 is at 00:30:1B:2F:F4:4E

- library list:

This is the list of all libraries that may be used in the applications - they must all be listed here explicitly, note that no symbolic links to libs are permitted it must be the real file (symlinks are ignored (bad idea)). you can use `ldd` to check what libs you need for your application. Note though that `ldd` does not necessarily list all libs (it does not list interlibrary dependencies !) - this is covered in [?] in detail.

```
root@rtl21:/home/hofrat/bproc-3.2.6# ldd tests/ping-pong
libbproc.so.2 => /usr/lib/libbproc.so.2 (0x40020000)
libc.so.6 => /lib/libc.so.6 (0x40025000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Here is a sample config file from the above information that we call `mycofig` from now on, it only has a single slave node - so we have the simplest possible master-slave setup.

```
interface eth0 192.168.1.41 255.255.255.0
nodes 1
iprange 0 192.168.1.40 192.168.1.40
node 0 00:30:1B:2F:F4:4E
```

```
libraries /lib/ld-*
```

2. Starting the ghosts

```
libraries /lib/libc-*
libraries /lib/libproc*
libraries /lib/libtermcap*
libraries /usr/lib/libbproc*
```

The configuration file should do for a first try - now we need to load the kernel modules to actually provide the necessary kernel services.

Bproc comes with two kernel modules that are in the bproc source tree in the kernel directory after compilation, if you ran the install then they end up in the non-standard directory `/lib/modules/2.4.22-bproc/bproc/*.o` (bad). So we simply manually load them for now to do some testing.

```
root@rtl21:~# insmod /lib/modules/2.4.22-bproc/bproc/vmadump.o
root@rtl21:~# insmod /lib/modules/2.4.22-bproc/bproc/bproc.o
root@rtl21:~# lsmod
Module                Size  Used by    Not tainted
bproc                 64936  2
vmadump              11640  0 [bproc]
8139too              15080  1
root@rtl21:~# dmesg
...
vmadump: 1.69 Erik Hendriks <erik@hendriks.cx>
bproc: Beowulf Distributed Process Space Version 3.2.6
bproc: (C) 1999-2002 Erik Hendriks <erik@hendriks.cx>
```

Any half way decent kernel module should reflect in the kernel messages viewed with `dmesg`.

TODO: what each module provides.

Now we can launch the master daemon on the master node

```
root@rtl21:/home/hofrat/bproc-3.2.6/daemons# ./bpmaster -v -c myocnfig
Creating FD sets fdset_size = 1024
Creating FD sets fdset_size = 1024
```

2.2.2. Slave setup

The slave setup is somewhat simpler - all it needs to know is who is the master node. So its loading the kernel modules and launching the slave daemon to listen for request from `rtl21`.

2. Starting the ghosts

```
root@rtl20:~# insmod /lib/modules/2.4.22-bproc/bproc/vmadump.o
root@rtl20:~# insmod /lib/modules/2.4.22-bproc/bproc/bproc.o
root@rtl20:~# cd /home/hofrat/bproc-3.2.6/daemon
root@rtl20:/home/hofrat/bproc-3.2.6/daemons# ./bplslave rtl21
root@rtl20:/home/hofrat/bproc-3.2.6/daemons#
```

So that step is simple. There is no configuration file involved on the slave, and basically you need not provide any other application on the slave node than `bplslave`, everything else can be sent over on demand !

2.2.3. Testing

Bproc comes with a quite large set of test apps - so we will look at a few. First we switch back to the non-root user with an `exit` on the master node. Then we can launch the ping-pong application.

```
root@rtl21:/home/hofrat/bproc-3.2.6/tests# ./ping-pong
```

-

Thats not very exciting is it :)

On the slave node we can see the active migrated process

```
root@rtl20:/~# pidof ping-pong
1302
```

Looks normal with `pidof` (or `ps`, `top`, etc.) - until we try to check the process details:

```
root@rtl20:/~# ls /proc/1302
/usr/bin/ls: /proc/1302: No such file or directory
```

The migrated process is not represented in the slave nodes `/proc` filesystem, other than that is a full local process.

2.3. Bproc management apps

2.3.1. /usr/bin/bpsh

```
root@rtl21:/home/hofrat/bproc-3.2.6# bpsh 0 pwd
/home/hofrat/bproc-3.2.6
root@rtl21:/home/hofrat/bproc-3.2.6# bpsh 0 hostname
rtl20
```

Note that the shell is active in the same location on the remote system as on the local system ! This is a bit of a bproc strangeness that makes a lot of sense for clusters as they generally are assumed to have a consistent setup. If bpsh is started from a directory that is not available on the target node then we end up in /.

```
root@rtl21:/home/hofrat/bproc-3.2.6# cd junk
root@rtl21:/home/hofrat/bproc-3.2.6/junk# bpsh 0 pwd
/
```

2.3.2. /usr/bin/bpstat

As the name says we can check the status of the nodes active - in our setup this is only one system.

```
root@rtl21:/home/hofrat/bproc-3.2.6/junk# bpstat
Node(s)          Status      Mode      User      Group
0                up         ---x--x--x root      root
root@rtl21:/home/hofrat/bproc-3.2.6/tests# bpstat -l
Node Address      Status      Mode      User      Group
  0 192.168.1.40   up         ---x--x--x root      root
```

bpstat can display a list of migrated processes - but will do so using the local pid's, that is, on the master you get the ghost pids that don't tell you much about the clients.

```
root@rtl20:/home/hofrat/bproc-3.2.6/tests# ./simple &
[1] 1025
root@rtl20:/home/hofrat/bproc-3.2.6/tests# bpstat -p
PID      Node
1025     0
```

2.3.3. /usr/bin/bplib

In the config file we dropped in a list of libraries that bproc

2.3.4. /usr/bin/bpcp

Not much to say to this - its the bproc version of copy.

```
root@rtl21:/etc/# bpcp profile 0:/tmp/profile
```

This will copy the file profile from the current directory on rtl21 to /tmp/profile on node 0.

2.3.5. /usr/bin/bpctl

The control of the nodes is basically possible via bps_h, simply execute any administrative commands in the shell on the node. Beyond this there are some administrative tasks directly related to bproc that can't be done via the shell interface, like change the state of the node, move the master node, or change the nodes group/user to use for commands, etc.

```
root@rtl21:/etc# bpctl -S 0 --user hofrat
root@rtl21:/etc# bpstat
Node(s)                Status      Mode      User      Group
0                       up         ---x---x--x hofrat    root
root@rtl21:/etc# bpctl -S 0 --state down
root@rtl21:/etc# bpstat
Node(s)                Status      Mode      User      Group
0                       down       ----- root      root
root@rtl21:/etc# bpctl -S 0 --state up
0: Node is down
root@rtl21:/etc# ssh -l root rtl20
root@rtl20's password:
root@rtl20:# bpslave rtl21
```

In the above sequence we first change the user of node 0 to hofrat. Next we mark node 0 down (note the mode change back to root), basically this means that we shut down bpslave on node 0 (it is still running but it will not respond). To restart it we log in via ssh and relaunching it on the commandline will bring rtl20 back into the bproc cluster. To temporarily remove a node from a bproc cluster change the state to unavailable and then when needed bring it back in with --state up

2. Starting the ghosts

```
root@rtl21:~# bpctl -S 0 --state unavailable
root@rtl21:~# bpstat
Node(s)                Status           Mode           User           Group
0                      unavailable     ---x--x--x    root           root
root@rtl21:~# bpctl -S 0 --state up
root@rtl21:~# bpstat
Node(s)                Status           Mode           User           Group
0                      up              ---x--x--x    root           root
root@rtl21:~#
```

Note though that there seems to be a problem in version 3.2.6 as marking a node unavailable had no obvious effect (other than changing the state returned) - not sure if this is a bug or I simply did something wrong here...

Futher the 3.2.6 release shows a very unfriendly behavior when bpstat or bpctl is called on the slave nodes

```
root@rtl20:~# bpctl -S 0 -u hofrat
0: Input/output error
root@rtl20:~# bpstat
bproc_nodelist: Input/output error
```

But there is no actual problem with this behavior - the commands are not thought be executed on the slave nodes and instead of giving polite errors just pounce on the users ;)

2.4. A few error cases

This will not be complete, some of the hopefully more common errors and there fixes. If you encounter other problems, drop me a note jder.herr@hofr.at.

```
hofrat@rtl21:~/bproc-3.2.6$ cd tests/
hofrat@rtl21:~/bproc-3.2.6/tests$ make ping-pong
gcc -Wall -g -O2 -I../clients -o ping-pong ping-pong.c -L../clients -lbproc
hofrat@rtl21:~/bproc-3.2.6/tests$ ./ping-pong
./ping-pong: error while loading shared libraries: libbproc.so.2: cannot open shared obj
hofrat@rtl21:~/bproc-3.2.6/tests$
```

...compiling went smoth so probably we forgot to register the new lib...

2. Starting the ghosts

```
hofrat@rtl21:~/bproc-3.2.6/tests$ su -
Password:
root@rtl21:~# ldconfig
root@rtl21:~# ldconfig -p | grep bpro
    libproc.so.2.0.16 (libc6) => /lib/libproc.so.2.0.16
    libbproc.so.2 (libc6) => /usr/lib/libbproc.so.2
root@rtl21:~# exit
hofrat@rtl21:~/bproc-3.2.6/tests$ ./ping-pong
bproc_move: Function not implemented
waitpid: Function not implemented
```

...no daemon running, lets try to launch teh daemon...

```
hofrat@rtl21:~/bproc-3.2.6/daemons$ ./bpmaster -v -c myocnfig
Creating FD sets fdset_size = 1024
./bpmaster: BPROC_SYS_VERSION: Function not implemented
```

...daemon does not want to start ? kernel modules not loaded !

```
root@rtl21:~# modprobe bproc
modprobe: Can't locate module bproc
root@rtl21:~#
```

...modules are not in the default location - you can move them or you can modify your `/etc/modules.conf` to include the non-standard path `/lib/modules/2.4.22-bproc/bproc` - you can manually install the modules by passing ther absolute location:

```
root@rtl21:~# insmod /lib/modules/2.4.22-bproc/bproc/vmadump.o
root@rtl21:~# insmod /lib/modules/2.4.22-bproc/bproc/bproc.o
root@rtl21:~#
```

You have two options to fix this problem permanently

- modify `/etc/modules.conf` to include the non standard path to `/lib/modules/2.4.22-bproc/bproc` and rerun `modprobe`.
- move the two `bproc` modules to a default location like `/lib/modules/misc`

Any way your daemon should now do something like:

```
root@rtl21:/home/hofrat/bproc-3.2.6/daemons# ./bpmaster -v -c myocnfig
Creating FD sets fdset_size = 1024
Creating FD sets fdset_size = 1024
```

if not - you are out of luck here - end of configuration section.

3. Simple examples

The directory `tests` in the `bproc` distribution contains a number of test applications. Before looking at those we want to present some really simple starters. First we do a remote "hello world".

```
#include <stdio.h>      /* printf */
#include <sys/bproc.h> /* bproc_move */

int main(int argc, char *argv[]) {

    /* move it to node 0 */
    bproc_move(0);

    printf("hello remote world !\n");
    fflush(stdout);
    return 0;
}
```

The obvious difference to the original "hello world" is the `bproc_move(0);` line in the main routine. This is what causes `bproc` to migrate the application to node 0 and continue execution there. The consequence of running on a remote node is that `stdout` behaves a bit different than on a local terminal, to get the output in a controlled manner we use `fflush(stdout)`, if all of your output statements end with a newline character then you could also make the application line-buffered by putting a `setlinebuf(stdout);` in.

A slightly modified version that shows some simple problems of remote program executions is:

3. Simple examples

```
#include <stdio.h>    /* printf */
#include <unistd.h>   /* sleep */
#include <sys/bproc.h> /* bproc_move */

int main(int argc, char *argv[]) {

    /* move it to node 0 */
    bproc_move(0);

    while(1){
        printf("hello remote world !\n");
        sleep(1);
    }
    return 0;
}
```

The first thing we notice when it is launched is that nothing happens for a minute or so and then we get a full screen of messages - not one at a time. If we remove the `sleep(1)` (bad version 1 below) then we get a similar effect just it takes only a second or so until we get the first screen full, but the version without the `sleep(1)`; in it shows another problem: if we send it a `<CNTRL>-<C>` it does not seem to react - it actually can take quite some time until `stdout` is flushed and the application can terminate.

bad version 1:

```
while(1){
    printf("hello remote world !\n");
}
return 0;
}
```

If you have a lot of output then the performance penalty can be quite impressive, bad version 2 below will seemingly not react to `<CNTRL>-<C>` any more, actually it does it just takes for ever ! - so as a general rule remote `stdout` usage should be limited as much as possible. Any way if a remote application generates tones of output then there most likely is a low-level design error lurking around that needs fixing ...

bad version 2:

3. *Simple examples*

```
while(1){
    printf("hello remote world !\n");
    fflush(stdout);
}
return 0;
}
```

TODO: details of test programs.

4. List of Acronyms

CVS - Concurrent Version Control