1 Introduction

Xcpu is a remote process execution system that represents execution and control services as a set of files in a hierarchical file system. The file system can be exported and mounted remotely over the network. Xcpu is slated to replace the aging B-proc cluster management suite.

A cluster that uses Xcpu has one or more control nodes. The control nodes represent the global view of the cluster and can be used to execute, view and control the distributed programs. The rest of the cluster nodes known as compute nodes and are used to run distributed applications as guided by the control node.

Xcpu is responsible not only for the execution of the programs, but also for their distribution to the compute nodes. It allows an arbitrary number of files (shared libraries, configuration files) to be pushed with the executable. In order to avoid the network bandwidth bottleneck between the control node and the compute nodes, Xcpu uses some of the compute nodes for the program distribution by dedicating them as distribution nodes for the duration of the program startup. This scheme, borrowed from B-proc, decreases significantly the start-up time for distributed applications.

The usage of a standard file-system interface makes the system easy to understand and operate. Furthermore, the ability to mount a compute node over the network to the local file system is a significant departure from the antiquated "remote execution" model in which little or no control over the application is given to the end user.

Below is a sample session to an Xcpu compute node, which launches a single process and displays its output on the console:

```
$ mount -t 9p 192.168.100.101 /mnt/xcpu/1 -o port=666
$ cd /mnt/xcpu/1
$ ls -l
total 0
-r--r--r-- 1 root root 0 Jul 25 10:19 arch
-r--r--r-- 1 root root 0 Jul 25 10:19 clone
-rw-r--r-- 1 root root 0 Jul 25 10:19 env
-r--r--r-- 1 root root 0 Jul 25 10:19 procs
-r--r--r-- 1 root root 0 Jul 25 10:19 state
$ tail -f clone &
[1] 8237
0tail: clone: file truncated
$ cd 0
$ ls -l
total 0
-rw-rw-rw- 1 nobody nobody 0 Jul 25 12:58 argv
-rw-rw-rw- 1 nobody nobody 0 Jul 25 12:58 ctl
-rw-rw-rw- 1 nobody nobody 0 Jul 25 12:58 env
-rw-rw-rw- 1 nobody nobody 0 Jul 25 12:58 exec
```

```
drwx------ 1 nobody nobody 0 Jul 25 12:58 fs
-r--r--r-- 1 nobody nobody 0 Jul 25 12:58 state
-r--r--r-- 1 nobody nobody 0 Jul 25 12:58 stderr
-rw-rw-rw- 1 nobody nobody 0 Jul 25 12:58 stdin
-rw-rw-rw- 1 nobody nobody 0 Jul 25 12:58 stdio
-r--r--r-- 1 nobody nobody 0 Jul 25 12:58 stdout
-rw-rw-rw- 1 nobody nobody 0 Jul 25 12:58 wait
$ cp /bin/date fs
$ echo exec date > ctl
$ cat stdout
Tue Jul 25 12:59:11 MDT 2006
$
```

First, the xcpufs file system is mounted at `/mnt/xcpu/1`. Reading from the `clone` file creates a new session and returns its ID. The user can copy an arbitrary number of files to the `fs` directory. The execution of the program is done by writing `exec` *progname* to `ctl` file.

2 Xcpufs

Xcpufs is a file server that runs on all compute nodes and exports an interface for program execution and control as a file hierarchy. The file server uses Plan9's 9P2000 protocol. Xcpufs can be mounted on a Linux control node using v9fs, or can be accessed directly using clients that speak 9P2000.

Xcpufs manages the processes it executes in sessions. In order to execute a program, the user creates a session, copies all required files, including the executable, sets up the argument list and the program environment and executes the program. The user can send data to program's standard input, read from its standard output and error, and send signals.

Only one program can be executed per session. The process of this program is known as *main session process*. That process may spawn other processes on the compute node. Xcpufs can control (send signals, destroy) only the main session process.

There are two types of sessions – normal and persistent. The *normal* session (and the subdirectory that represents it) exists as long as there is an open file in the session directory, or the main session process is running. The *persistent* session doesn't disappear unless the user manually *wipes* it.

Normally the program will run as the user that attached the filesystem (field `uname` in Tattach 9P2000 message). The program can be executed as somebody else if the ownership of the session subdirectory is changed to different user. The files in the session directory are accessible by the owner and the members of the owners default group.

All files that are copied to a session are stored in a temporary directory on the compute node. Before executing the program, xcpufs changes the process' current directory to the session directory, and sets XCPUPATH envrironment

variable to it. All files in the temporary storage are deleted when the session is destroyed.

In addition to the environment set through the env file, xcpufs adds three more variables:

XCPUPATH contains the full path to session's temprary directory

XCPUID contains the global ID of the session (same as the `id` file

XCPUSID contains the local session ID (same as the session directory)

There are two groups of files that xcpufs exports – top-level files that control the xcpufs, and session files that control the individual sessions.

2.1 Top-level files

```
        arch
        clone
        env
procs
state
```

`Arch` is a read-only file, reading from it returns the architecture of the compute node in a format *operating-system/processor-type*.

`Clone` is a read-only file. When it is opened, xcpufs creates a new session and exports a new session directory in the file-system. It also copies the content of the global `env` file to the session one. The open can fail if the content of the global `env` file is not formatted correctly.

Reading from the file returns the name of the session directory.

`Env` contains the global environment to be used for all new sessions. When a new session is created, the content of `env` is copied to the sessions `env` file.

Reading from the file returns the current environment. Writing to the file changes the environment.

The content of the global and session environment files have the following format:

```
environment = *env-line
env-line = env-name ``='' env-value LF
env-name = ALPHA *[ALPHA | DIGIT]
env-value = ANY
```

If the `env-value` contains whitespace characters (SP, TAB or LF) or single quotes, it is enclosed in single quotes and the original single quotes are doubled (i.e. ' becomes '').

`Procs` is a read-only file. The content of the file is a s-expression that list all processes running on the compute node. The first subexpression contains list of the fields returned for the processes. The list is architecture dependent.

`State` file contains the state of the node. The user can write any string to the file.

2.2 Session directory

```
        argv
        ctl
        exec
        env
fs
        state
        stdin
        stdout
        stderr
        stdio
        wait
id
```

2.2.1 `Ctl` file

The `ctl` file is used to execute and control session's main process.

Reading from the file returns the main process pid if the process is running and -1 otherwise.

The operations on the session are performed by writing to it. `Ctl` commands have the following format:

```
ctl = *cmd-line
cmd-line = command `` `` *[argument] LF
command = ``exec'' | ``clone'' | ``wipe'' |
  ``signal'' | ``close'' | ``type''
argument = ANY
```

If the `argument` contains whitespace characters (SP, TAB or LF) or single quotes, it is enclosed in single quotes and the original single quotes are doubled (i.e. ' becomes '').

Writing to `ctl` ignores the specified offset and always appends the data to the end of the file. It is not necessary a write to contain a whole (or single) command line. Xcpufs appends the current write data to the end of the buffer, and executes all full command lines. The write operation returns when all valid commands are finished.

`Ctl` supports the following commands:

exec *program-name directory* Execute the program. For backward compatibility, if program name is not specified, xcpufs executes the program named "xc" from the session directory. If the *directory* is specified, xcpufs sets the current directory to that value before executing the binary. If it is not specified, the session directory is used.

If the program-name is a relative path (i.e. doesn't start with '/' character, the session directory path is appended in front of it.

clone *max-sessions address-list* Copies the current session content (argument list, environment and files from `fs` directory) to the specified sessions. If `max-sessions` is greater than zero, and the number of the specified sessions is bigger than `max-sessions`, `clone` pushes its content to up to max-sessions and issues `clone` commands to some of them to clone themselves to the remaining sessions from the list.

The format of the session-address is:

```
address-list = 1*(session-address ``,'')
session-address = [``tcp!''] node-name [``!''port]
          ``/'' session-id
node-name = ANY
port = NUMBER
session-id = ANY
```

wipe Closes the standard I/O files, if the main session process is still alive, kills it (SIGTERM) and frees all objects used by the session. This command is normally used to terminate persistent sessions.

signal *sig* Sends a signal to the main session process. The signal can be specified by number, or name. The supported signal names may depend on the node's architecture.

type `normal` | `persistent` Changes the type of the session.

close `stdin` | `stdout` | `stderr` Closes the standard input/output/error of the main session process.

id *id* Sets the session id (see the `id` file). If the job-id or the proc-id parts are ommited, they are not changed.

Reading from the `ctl` file returns the session ID.

Sending a signal to a session that doesn't have running process will cause the write function to return an error.

2.2.2 `Exec` file

The exec file is kept for backward compatibility. Writing to it creates a file named "exec" in the `fs` directory.

2.3 `Fs` directory

The `fs` directory points to the temporary storage created for the session. The user can create directories and files in that directory. Creation of special files (symbolic links, devices, named pipes) is not allowed.

2.3.1 `Argv` file

Writing to the `argv` file sets the program argument list.

`argv` has the following format:

```
argument-list = 1*(argument (SP | TAB | LF))
argument = ANY
```

If the `argument` contains whitespace characters (SP, TAB or LF) or single quotes, it is enclosed in single quotes and the original single quotes are doubled (i.e. ' becomes '').

Reading from the `argv` file returns the current content.

2.3.2 `Env` file

When the session is created, the content of the `env` file is populated from the global `env` file.

Writing to the `env` file modifies the session environment. Modifications done after the program is executed don't change its environment.

The format of the session `env` file is identical to the global one.

2.3.3 `State` file

`State` is a file that can be used both for reading and writing. It is used by the cluster monitoring framework to mark computational nodes' states. When Xcpu starts the state file contains no information. If a string is written to it this string is returned by subsequent reads.

2.3.4 `Stdin` file

`Stdin` is a write-only file. The data from the write operation is passed to the standard input of the main session process. The write may block if the main process doesn't consume the data.

Closing the stdin file doesn't close the standard input stream of the main process. The file can safely be opened and closed multiple times.

2.3.5 `Stdout` file

`Stdout` is a read-only file. The read operations blocks until the main session process writes some data to its standard output.

If the file is opened more than once, and there are blocked read operations for these files when some data is available from the standard output, xcpufs sends the data to every open file.

Closing the `stdout` file doesn't close the standard output stream of the main process. The file can safely be opened and closed multiple times.

2.3.6 `Stderr` file

`Stderr` is a read-only file. The read operations blocks until the main session process writes some data to its standard error.

If the file is opened more than once, and there are blocked read operations for these files when some data is available from the standard output, xcpufs sends the data to every open file.

Closing the `stderr` file doesn't close the standard output stream of the main process. The file can safely be opened and closed multiple times.

2.3.7 `Stdio` file

`Stdio` file combines `stdin` and `stdout` functions. Reading from `stdio` is equivalent to reading from `stdout`, and writing to it is equivalent to writing to `stdin`.

If the file is opened more than once, and there are blocked read operations for these files when some data is available from the standard output, xcpufs sends the data to every open file.

2.3.8 `Wait` file

`Wait` is a read-only file. Reading from it returns the exit code of the main session process. The read operations will block until the process ends.

2.3.9 `Id` file

`Id` is a read-only file that contains the user-specified id. The format of the id is:

```
id = job-id ‘‘/’’ proc-id
job-id = ANY
proc-id = NUMBER
```

The job-id is a global identifier of the job, the proc-id is an id of the process within the job. Both are set by the user via “id” command.

2.4 Running Xcpufs

Xcpufs accepts the following options:

-d Optional. If set, xcpufs stays in foreground and shows all 9P2000 messages.

-p port Optional. Sets the port number that xcpufs listens on. The default value is 666.

-t tdir Optional. Sets the directory in which xcpufs creates the temporary session directories. The default value is /tmp.

-s Optional. If set, xcpufs runs all programs as the user that started xcpufs.