# Design and Implementation of Support for Pipes in Condor

Master's Thesis

P. van Sebille

Supervision:

Delft University of Technology
Faculty of Technical Mathematics and Informatics
Operating Systems and Distributed Systems Group
P.O. Box 356, 2600 AJ Delft, The Netherlands

prof. dr I.S. Herschberg
ir dr D.H.J. Epema
drs J.W.J. Heijnsdijk
ir J.F.C.M. de Jongh

National Institute for Nuclear Physics and High-Energy Physics
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

dr R. van Dantzig (SMC)

August 1994

# Contents

# Preface

This report is the result of a six-month graduating term for the Operating Systems and Distributed Systems group of the Technical University of Delft. This assignment was performed within the Computer System Group of NIKHEF (National Institute for Nuclear Physics and High-Energy Physics) under the supervision of R. van Dantzig (Spin Muon Collaboration), and D.H.J. Epema and J.F.C.M. de Jongh (Technical University of Delft).

I would like to thank Maarten Litmaath for the interest he showed in my project and also for his help with running a test job at CERN[1].

I would like to thank Ronald Boontje for his help and advise on the design issues of my project, Tom Ploegmakers and Bas Tummers for answering numerous questions on UNIX, and of course all the other members of the CSG for the wonderful time I had during my stay at NIKHEF.

I would like to thank Miron Livny and Mike Litzkow for the discussions we had on Condor, and Mike Litzkow also for always finding the time to answer my questions on the Condor system.

I would like to thank Dick Epema, who provided me with this assignment, together with Jan de Jongh, for their supervision and their many valuable comments. Finally, I would like to thank René van Dantzig for his supervision, guidance and help during my graduation term.

Peter van Sebille, Amsterdam, august 1994.

[1]Centre Européen pour la Recherche Nucléaire.

# Chapter 1

# Introduction

Condor is a distributed batch system that allows for batch processing in a cluster of UNIX workstations. The main goal of Condor is to utilise machines that otherwise would be *idle*. Users may submit jobs which will be queued by Condor and scheduled among available machines. From the moment of submission until the job's completion, Condor will host the job. This is completely transparent to the user. For example, the user does not know, and possibly does not care, where his jobs are run. Condor will pick a machine from a pool of available machines to run the user's job. If necessary, Condor moves a running job from one machine to another. Condor is developed at the University of Wisconsin, Madison, U.S.A., [2],[3],[11],[12],[13],[14],[15].

The type of job that is suitable for Condor is long-running, computing intensive processes that preferably do not much I/O. Although this is not a requirement for running a job with Condor, short-running jobs gain less due to the overhead involved.

Work is in progress in Wisconsin and therefore there are different versions of Condor. An important one is the *official release*, version 4.1.3b, that can be obtained using anonymous ftp to *ftp.cs.wisc.edu*. The type of job that is used in the official version consists of *one* process only. We shall refer to such a job as a *single-process job*. One limitation of this type is very clear: jobs may only consist of a single process. This excludes a variety of jobs that require more than one process, such jobs are referred to as *multiple-process job*. Going from single-process jobs to multiple-process jobs is definitely a step forward. However, whenever Condor is running a job on a different machine than the machine from which the job is submitted, I/O is redirected back to the submitting machine. For a single-process job there can be no other way, the process probably needs input and is likely to produce output, which must be stored in a file. However, for a multiple-process job, the output of one process may be the input of another process in the job. In such a case, we would like to pipe these data from one process to another. What we then have is what we will call a *pipe job*. In Figure 1.1, an

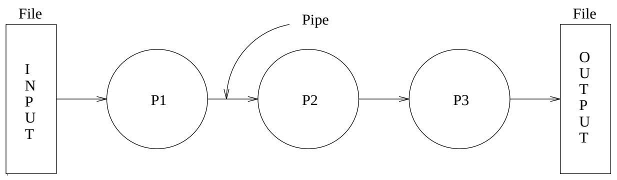

Figure 1.1: Example of a pipe job.

example of a pipe job is shown. The first process reads its input from a file and its output is piped to the second process. The data are piped from the second process to the third and finally, the data are written to a file.

Piping already exists in UNIX shells. The three processes in the previous example could be started in such a shell with the next command:

```
p1 < input | p2 | p3 > output
```

What we see here, in UNIX terminology, is that these three processes are expected to read something from *stdin* and write something to *stdout*. The token < (it implies file redirection) is used to denote that whenever process p1 reads from stdin it actually reads file "input". This is transparent for process p1, all it does is reading from stdin and it does not know that the shell maps a file to it. Similar, token > means that stdout of process p3 is redirected to a file. The token | (the pipe symbol) denotes that stdout of the first process is mapped on stdin of the second process (again transparent for both processes). The data are said to be piped from the first to the second process and from the second to the third.

What we want is to extend Condor with a piping facility. This piping facility should not only support a pipe mechanism described above, that is, piping from stdout to stdin, but also a mechanism that allows for a mapping of files on the ends of pipes[1]. Mapping files on the ends of pipes is a feature not found in any shell programming, but it is a requirement for a large collection of jobs at CERN. These jobs consist of Fortran programs that analyse data, either gathered from experiments or generated by simulation, and these programs make up a pipeline. Typical about these programs is that they use hard-coded filenames (usually something like fort.3 or fort.79). To take full advantage of pipes, we should redirect the data that are written to a file, to the end of pipe. This requires a mechanism that will map the name of a file on the end of a pipe. We will design a piping facility that allows these jobs to run with Condor without rewriting a single line of source code of any of the programs.

We already discussed one version of the Condor system: the official release. Another important version is a test version at Wisconsin; we will refer to it as *wisc* version[2]. The wisc

[1] Note that this not the same as mapping a file name on stdin as in file redirection.
[2] This version is known by the tag: Sparc_Sunos4_1:A.

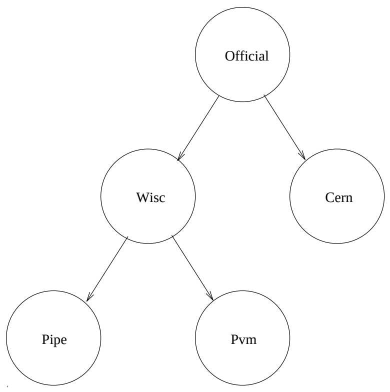

Figure 1.2: Relationship between Condor versions.

version supports more types of jobs than the traditional single process job. It also has quite some support for multiple-process jobs, however, one piece of the software (Shadow) is not yet prepared to handle this support. Also, the designers at Wisconsin already anticipated pipe jobs and provided the basic data structures and operations on them. Derived from this wisc version was a version which we will call the *pvm* (Parallel Virtual Machines) version, because it supports pvm jobs. We shall not discuss pvm here; all we need to know is that a pvm job consists of multiple processes as well as pipe jobs. Finally, there is a version of Condor at CERN, which will be called the *cern* version. This version is derived from (an early) official release of Condor and was the first attempt to implement pipe jobs.

As a basis for my graduation assignment, I used the wisc version. There could have been only doubt on whether to use the wisc or the pvm version. The wisc version is the way to go for future Condor extensions since it is a better implementation than the official release (parts are completely rewritten in C++[3]). The reason to use the wisc version is that I already understood what was missing in it by the time I received the pvm version. So, instead of also understanding the pvm version and removing redundant parts, I decided to extend the wisc version. Just for the record, I'll call my version the *pipe* version. The relationship between all the mentioned Condor version is showed in Figure 1.2.

We have split up the assignment in two parts. First we will extend the wisc version so that it fully supports multiple-process jobs. Next, the pipe functionality will be implemented. It is not our intention to make two versions out of this. The first stage is merely a basis for the actual pipe version. Chapter 2 will give a brief introduction to Condor, in chapter three and four we will discuss the design issues for multiple-process jobs and pipe jobs respectively.

Checkpointing a multiple-process job is merely checkpointing the individual processes. Checkpointing pipe jobs is not as easy due to communication dependencies, these are the pipes. We designed an algorithm for checkpointing a pipe job. Unfortunately, there was no time left to

[3]Originally, Condor was implemented in K&R C.

implement it. Checkpointing pipe jobs will be discussed in chapter 5. The implementation of the pipe version is the subject of chapter 6. In chapter 7 we show some results from testing the pipe version. Finally, in chapter 8 we will present our conclusions.

# Chapter 2

# Introduction to Condor

This chapter gives a brief overview of the Condor system. We will not discuss all details but restrict ourselves to the subjects that are necessary for this report; these are: submitting a job, remote execution and checkpointing.

In a pool of UNIX workstations, Condor monitors the activity of participating machines. When a machine is determined *idle*, it is added to a list of processors. These idle machines are scheduled among available user jobs. The user may submit jobs which are stored in the *JobQueue* on the *submitting* machine. In Figure 2.1 such a pool is depicted.

On each machine run two Condor daemons called the *Schedd* (scheduler daemon) and the *Startd* (starter daemon). One machine in the pool is configured to be the *Central Manager*, on which two extra daemons run. For this paper, the Central Manager is of less interest, so these two extra daemons will be addressed to as the Central Manager. After a configured period of time, a Schedd will negotiate with the Central Manager on behalf of one of the jobs it has in its JobQueue . The Central Manager decides if and on which machine this job may run. The policies for the Schedd to pick one of the user jobs from its JobQueue and for the Central Manager to select an idle machine, are not considered here. All we need to know is that the Central Manager will return the name of the idle machine when a request is granted. The Schedd will request the Startd on this machine (called the *remote host* or the *execution machine*) to start the job. Only if the remote host has enough swap and disk space to run the job, the request is granted by the Startd. Finally, the job is started on the remote machine.

When the job runs remote, it is periodically checkpointed, that is, the state of the job is captured and stored in a *checkpoint file*, which is used to restart the job from on a different machine. There are two reasons Condor needs to checkpoint a job periodically. First, Condor guarantees that a job eventually will complete. For long-running jobs this is essential. Suppose that a machine on which we run a job that takes for about a month of CPU time to complete its computation crashes on the last day. This means that a month of CPU time is wasted

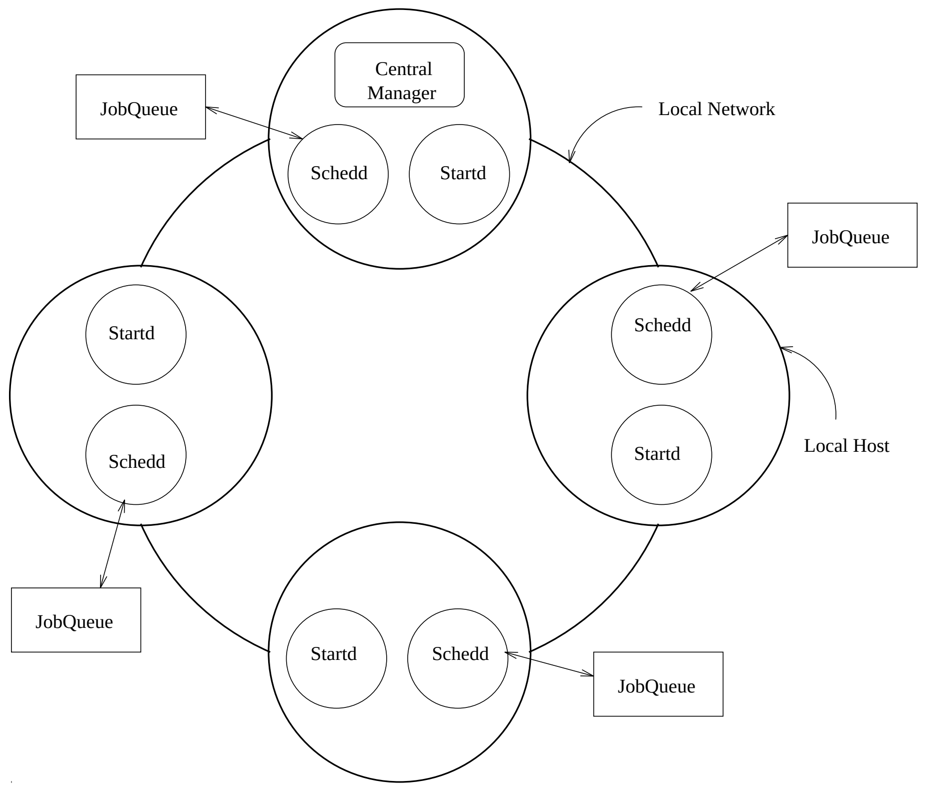

Figure 2.1: Condor pool of workstations.

and the job has to be restarted from scratch. With checkpointing, we could restart the job from its most recent checkpoint on another machine, wasting only the time between the time we last checkpointed the job and the time of the crash. The second reason Condor needs checkpointing is that a user on whose machine Condor is running a remote job, may return to his machine at any time and start working on it. Then, by definition, that machine is not idle anymore and Condor should migrate the job to a new, idle machine. When periodic checkpointing is used, the most recent checkpoint file of the job is used to restart the job from. Again, this prevents a total restart.

## 2.1 Submitting a Job

To submit a job, the user needs to specify the job in a special file, called the *job description file*. This file is passed as an argument to *Condor_submit*, which will parse the job description file and will store the information in the local JobQueue. We will not discuss the syntax of the job description file in full detail but focus on the basics only. Furthermore, we'll only describe the 'traditional' Condor job here: a job that consists of one process only. For this kind of job, the syntax is:

```
<jdf>            ::= <executable> <queue-list>

<queue-list>     ::= <command-list> <queue-command>  |
                     <command-list> <queue-command> <queue-list>

<command-list>   ::= <input-name> <output-name> <error-name>
                     <argument-list> <initial-dir> <root-dir>
                     <environment>

<executable>     ::= "Executable" "=" <exe-name>

<input-name>     ::= "Input" "=" <file-name> | <empty>

<output-name>    ::= "Output" "=" <file-name> | <empty>

<error-name>     ::= "Error" "=" <file-name> | <empty>

<argument-list>  ::= "Argument" "=" <string-list> | <empty>

<initial-dir>    ::= "Initialdir" "=" <dir-name> | <empty>

<root-dir>       ::= "Rootdir" "=" <dir-name> | <empty>

<environment>    ::= "Environment" "=" <string-list> | <empty>
```

```
<queue-command> ::= "Queue"

<string-list>   ::= <string> | <string> <string-list>
<exe-name>      ::= <file-name>

<file-name>     ::= string
<dir-name>      ::= string
```

The following two semantic rules apply with the above syntax:

1. `exe-name` must refer to an existing UNIX executable file
2. both `dir-name` of `initial-dir` and `root-dir` must refer to existing directories

From this syntax it follows that the most simple job description file consists of an *executable*- and a *queue*command.

It is illustrative to look at an example of a job description file:

```
#################################
#
#       Job description file
#
#################################

Executable    = foo

Initialdir    = /home/usr/john_doe
Input         = data
Output        = log
Arguments     = 1 2 3

Queue

################################
```

In this example, we submit a job that consists of a process that has an executable file named "foo". With "Input", "Output" and "Error", the user may specify the names for *stdin*, *stdout* and *stderr*, which allows for a shell-alike file redirection. After condor_submit has parsed the job description file, it copies the executable file and changes this copy so that it looks like a checkpoint file, which is therefore called the *initial checkpoint file*.

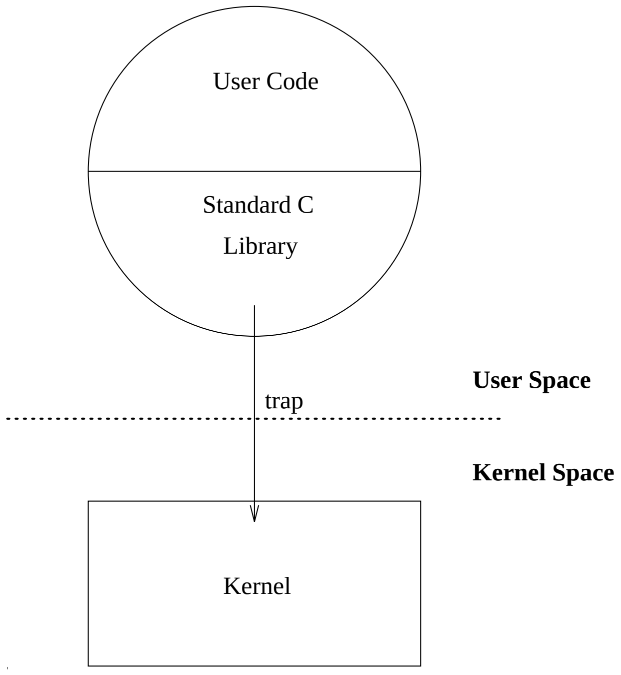

Figure 2.2: Local system calls in UNIX.

## 2.2 Remote Execution

The idea of remote execution is that a job or process is not running on the local but on a remote host. We will call the local host, the machine from which the job was submitted, the *initiating* machine, and the machine the job is running on remotely, the *execution* machine.

The difficulty of remote execution is to give the remote running process the illusion it is still running on the initiating machine. If, for instance, the process opens a file named "foo", it may be that this file resides on a file system mounted only at the initiating machine. Thus the file cannot be accessed from the remote machine. We therefore must have a mechanism that let the remote process "see" the same execution environment on the remote host as on the initiating machine. For this reason that users must relink their programs with the *Condor library*, a replacement for the standard *C library*. Most important about the Condor library is that all system call stubs are rewritten to allow *remote* system calls.

Every system call has a function (called a *stub*) in the C library with the same name as the system call. These stubs merely causes a trap to the kernel, the kernel performs the system call and returns control to the user process. The stub reads the return value of the system call and passes it back to the caller. Figure 2.2 shows the way the C library handles system calls.

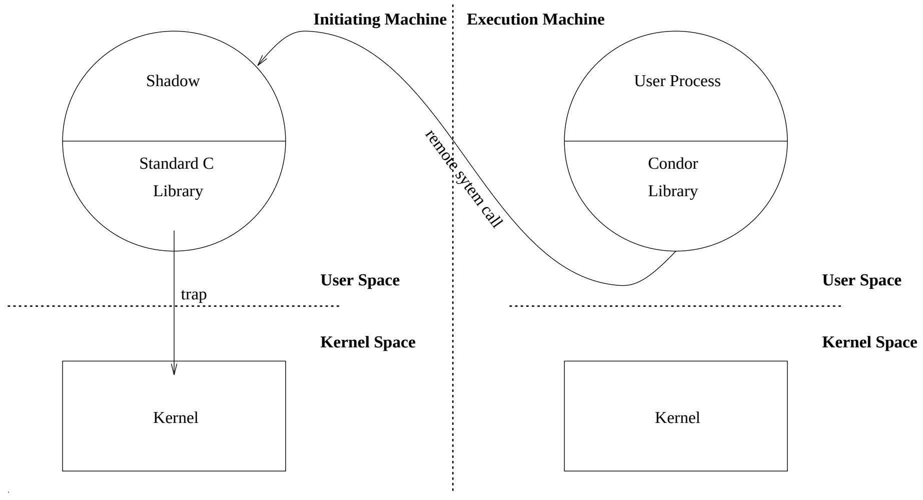

Figure 2.3: Remote system calls in Condor.

In the Condor library, the stubs of all system calls are changed. Most system calls are redirected back to the initiating machine. There, a special process, called the *Shadow*, is running that acts as a server for remote system calls on behalf of the remote running process, as is shown is Figure 2.3. Not all system calls are redirected back to the Shadow by the Condor library. Some system calls must be done locally, for example, *sbrk*() which is used to allocate memory. In Figure 2.4, the Condor library is shown in more detail. The user code only interacts with the standard C library and the Condor system call stubs. The standard C library will also use Condor system call stubs. The Condor routines make normal use of the facilities found in the standard C library.

## 2.3 Checkpointing

A remote running process is periodically checkpointed by Condor. This facility is implemented completely outside the UNIX kernel and is part of the Condor library. Checkpointing a process results in storing the state of the process in a checkpoint file, which is in fact a normal UNIX executable. The difference between the two is that a checkpoint file contains additional information that is used by Condor to restore the state of the process so that it may resume execution from the point where it was checkpointed.

Capturing the state of a running UNIX process is a difficult job. Not only all memory must be saved (stack, data, and text), but also parts of the process' state that is only known to the kernel. Usually, that part is changed as result of a system call, for example, *open*() changes the open file table. Condor changes the stubs of such system calls so that they do not only perform the system call (either locally or remotely) but also administrate the effect of the

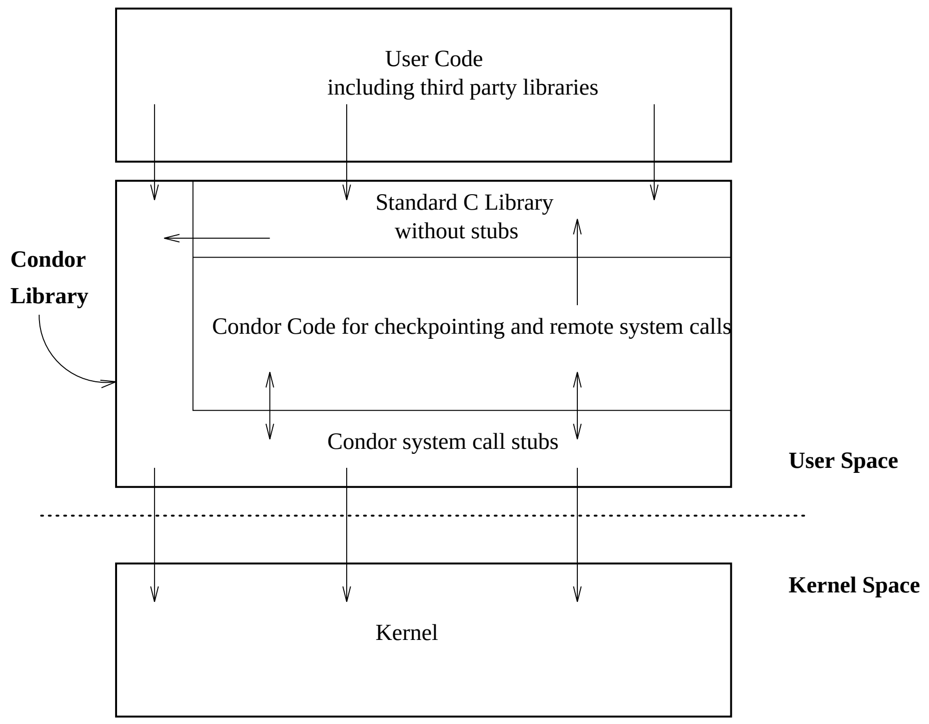

Figure 2.4: Condor library.

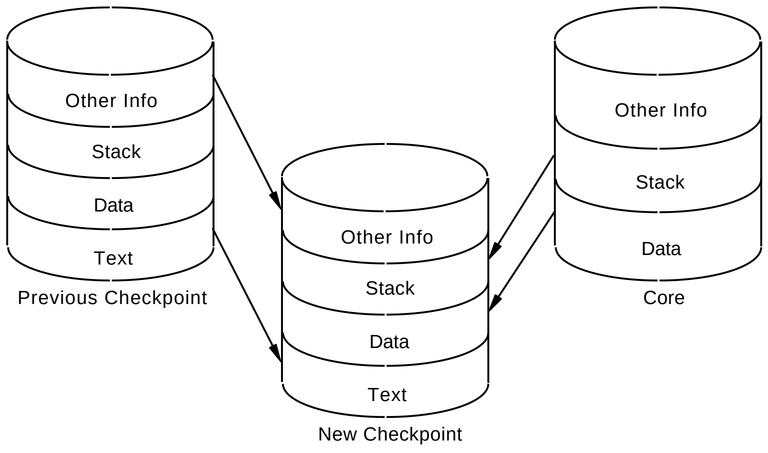

Figure 2.5: Creating a new checkpoint file.

system call. Unfortunately, some system calls have very complex side effects, for example, *fork*() and *exec*(), which may be used to start a child process. Therefore, these system calls are prohibited by Condor; they usually deal with interprocess communication and process control.

As part of checkpointing, the user process exits and dumps core; this is forced by the checkpointing routine in the Condor library. Condor will create a new checkpoint file from the previous one and the core file, as is shown in Figure 2.5. The stack segment (layout of function calls) and data segment (variables etc.) are extracted from the core file as these have changed since the last (or initial) checkpoint. This new checkpoint file is again an executable file. Furthermore, the Condor library includes a function *MAIN*(), which will be the entry point for the process. It is therefore called before the user function *main*(). When run for the first time, MAIN() will do some initialisation and then call the user function main(). The next time the process is started, MAIN() will restore the state of the process. As normal, a UNIX process is born with an empty stack. To let the user process continue where it left off just before checkpointing, MAIN() reads in the stack from the checkpoint file. Before doing this, Condor performs extra initialisations, such as re-opening files that were open before checkpointing. Just before checkpointing, the *stack pointer* and the *program counter* were saved in a buffer with a *setjmp*() call. After reading in the stack, Condor calls *longjmp*() to reset the stack pointer and the program counter. After longjmp(), the process continues from the point it left off just before checkpointing.

# Chapter 3

# The Design of Condor Support for Multiple-Process Jobs

This chapter describes how and to what extent the wisc version supports multiple-process jobs and what is needed to fully support them. A job is administrated in so-called *process structures* ( proc structures). There are two versions of proc structures used in Condor, these are *V2_PROC* structures and *V3_PROC* structures, for single-process jobs and multiple-process jobs respectively. Only a few Condor processes manipulate with proc structures explicitly, these are[1] condor_submit, the Shadow, the Startd and the Starter. First, we will examine the model of multiple-process jobs and how such jobs are specified. Then, the proc structures are discussed and the above mentioned processes are described in more detail. Finally, we summarise what changes are needed for each process.

## 3.1 The Model of Multiple-Process Jobs

For single-process jobs, Figure 3.1 shows the relationship between all relevant processes. When the Schedd gets permission to run a job, it starts the Shadow Parent with the name of the execution machine as argument. The Shadow Parent contacts the Startd on the execution machine and sends it a *start-foreign-job* request. The Startd will create two communication channels, one for remote system calls and one for logging (not depicted) and returns the port numbers that go with these two channels to the Shadow Parent. On the initiating machine, the Shadow Parent forks the Shadow Child, whereas on the execution machine, the Startd starts the Starter.

The Shadow is split up in a Parent and a Child process because the Shadow must run under the

[1] Only the processes of interest are listed.

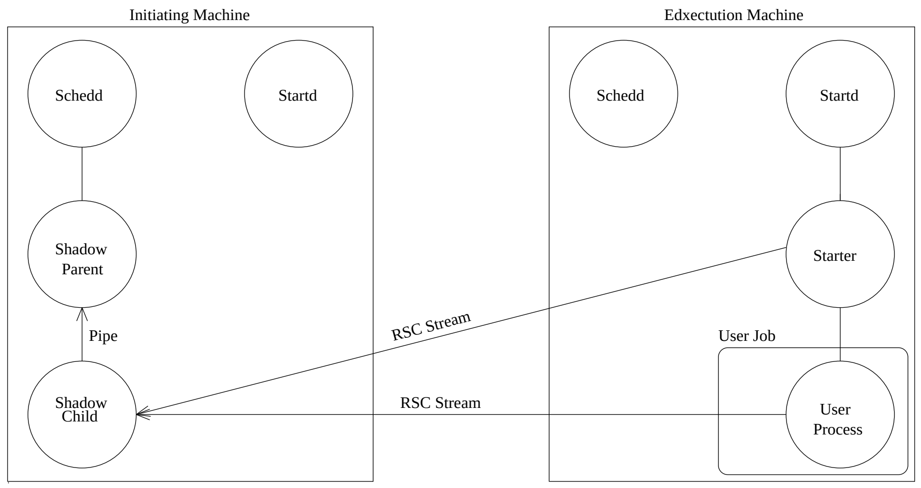

Figure 3.1: The model for single-process jobs.

UID of the user to perform remote system calls but under the UID of Condor and sometimes root to maintain Condor administration. Therefore, the Child runs under the UID of the user and the Parent under the UID of Condor (and sometimes root). The main task of the Shadow Child is to perform system calls on behalf of the user process. When the user process dies, the Shadow Child will receive information regarding the process's death from the Starter. The Shadow Child will send this status information to the Shadow Parent via the pipe.

The main task of the Starter is to control the user job. The Starter must start it, suspends its execution and transfer any checkpoint file back to the initiating machine. Although in Figure 3.1, it is depicted that the Starter and the user process have separate connections with the Shadow, they actually Share the same communication channel. The user process needs it to request the Shadow to perform system calls and the Starter needs it to perform so-called *pseudo system calls*. These pseudo system calls are special requests that are handled by the Shadow Child in the same way as a normal system call request. Since the Starter only needs to communicate through this communication channel when the user process is not running (it is either suspended or not running at all), they can share the channel.

In the wisc version there is no similar model for multiple-process jobs, so we have designed a new model, as depicted in Figure 3.2. All user processes and the Starter have a separate communication channel with the Shadow Child. The Shadow Child will poll each connection with the execution machine and will perform any pending requests.

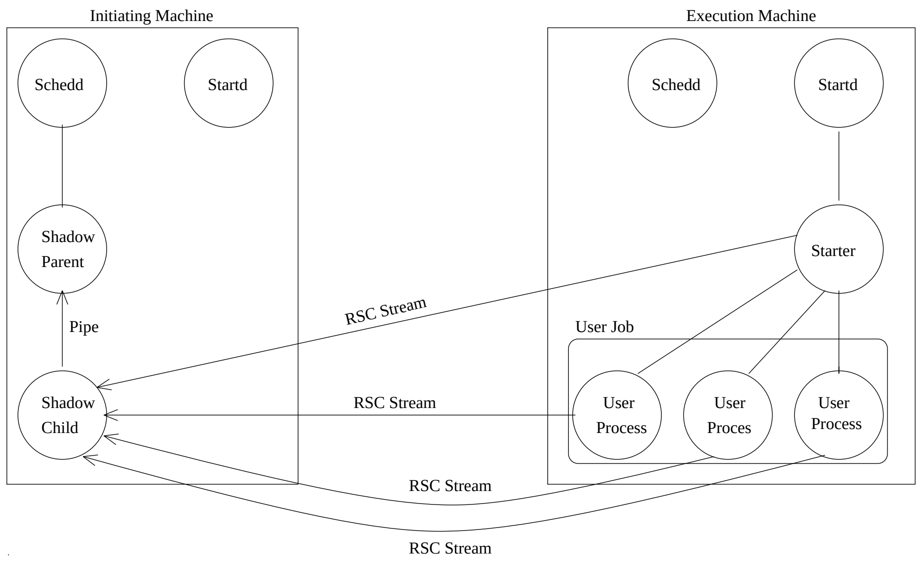

Figure 3.2: The model for multiple-process jobs.

## 3.2 The Specification of Multiple-Process Jobs

The wisc version has a syntax for multiple-process jobs. This will do for the moment so nothing needs to be changed. A difference with single-process jobs is that we now have job-specific information (for example, the environment string) and per-process information. In the syntax of multiple-process jobs, the per-process information consists of the names for stdin, stdout, stderr and the argument string. A formal description[2] is given below.

```
<jdf>           ::= <executables> <queue-list>

<queue-list>    ::= <command-list> <queue-command>  |
                    <command-list> <queue-command> <queue-list>

<command-list>  ::= <job-specific> <proc-specific>

<job-specific>  ::= <initial-dir> <root-dir> <environment>

<proc-specific> ::= <per-proc> | <per-proc> <proc-specific>

<per-proc>      ::= <input-name>  <output-name> <error-name>
                    <argument-list>

<executables>   ::= "Executable" "=" <exe-list>
```

[2]A few details are omitted.

```
<input-name>    ::= <exe-name> "." "Input" "=" <file-name> | <empty>

<output-name>   ::= <exe-name> "." "Output" "=" <file-name> | <empty>

<error-name>    ::= <exe-name> "." "Error" "=" <file-name> | <empty>

<argument-list> ::= <exe-name> "." "Argument" "=" <string-list> | <empty>

<initial-dir>   ::= "Initialdir" "=" <dir-name> | <empty>

<root-dir>      ::= "Rootdir" "=" <dir-name> | <empty>

<environment>   ::= "Environment" "=" <string-list> | <empty>

<queue-command> ::= "Queue"

<exe-list>      ::  <exe-name> <exe-list> | <empty>

<string-list>   ::= <string> | <string> <string-list>
<exe-name>      ::= <file-name>

<file-name>     ::= string
<dir-name>      ::= string
<string>        ::= char+ | <empty>
```

The semantic rules are similar to those for single-process jobs:

1. All `exe-name`s must refer to existing UNIX executable files, must be unique and must be listed in the `exe-list`.
2. At least one `exe-name` must be specified.
3. Both `dir-name`s of `initial-dir` and `root-dir` must refer to existing directories.

Again, a short example of a job description file is given. It shows a job of two processes, foo and bar. For process bar, stdin and stdout is specified; for process foo only stdin is specified; for both, arguments are specified. Note that the initial working directory is job-specific information.

```
   #################################
   #
   #       Job description file
   #
   #################################
```

```
Executable    = foo bar

Initialdir    = /home/usr/john_doe

foo.Input     = inp
foo.Output    = data
foo.Arguments = one two three

bar.Input     = samples
bar.Output    = results
bar.Arguments = 1 2 3

Queue

################################
```

## 3.3 Process Structures

A proc structure contains all the information that is needed to administrate a job. Since Condor was designed for single-process jobs, the original proc structure (see Figure 3.3) only contains information about one process. The term proc structure is misleading: It seems if a proc structure contains information about one process only; this is true for single-process jobs (there is only one process) but not for multiple-process job. A proc structure holds the information for *all* processes in a job. Proc structures should have been called *job structures*, but they are not; we will still use the term proc structure. When this proc structure (now called a *V2_PROC* structure) is used, **version_num** must be set to 2, otherwise erroneous events may occur.

For multiple-process jobs, a *V3_PROC* structure is used, see Figure 3.4 . **Version_num** of a *V3_PROC* structure must be set to 3. The first part of this proc structure contains job-specific data which are almost the same as for a *V2_PROC* structure. New, among other things, is **universe**, which specifies the type of job: Currently the following values are defined:

- **STANDARD** : Original - single-process jobs, one per machine
- **PIPE** : Pipes - all processes on a single machine
- **LINDA** : Parallel applications via Linda
- **PVM** : Parallel applications via Parallel Virtual Machine
- **VANILLA** : Non-Condor linked jobs

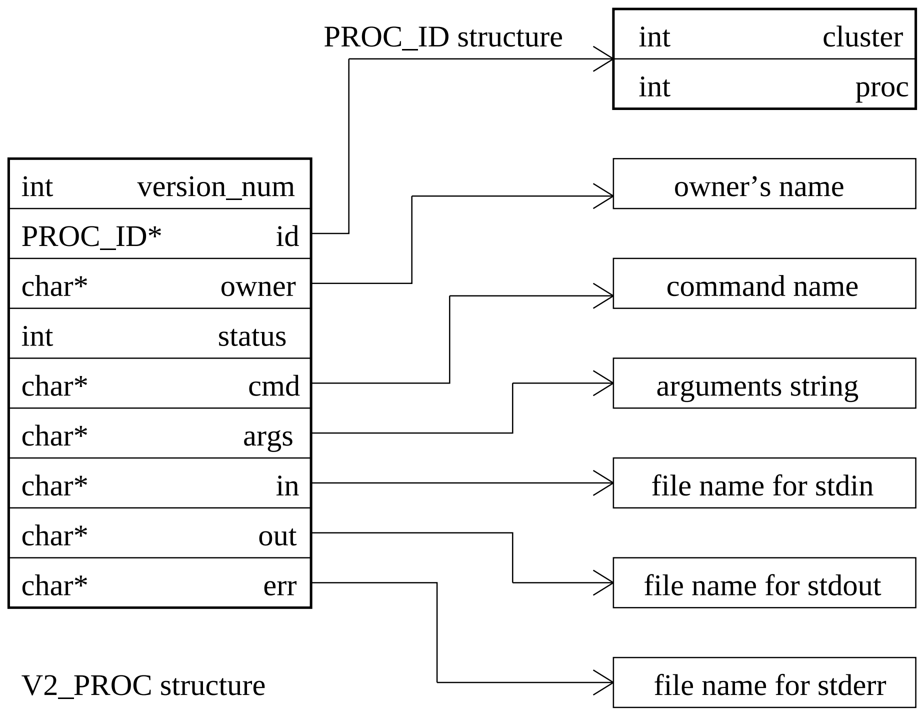

Figure 3.3: The original proc structure (*V2_PROC* structure).

- **PVMD** : Explicit, PVM daemon process

Types **linda**, **pvm**, **vanilla** and **pvmd** are not considered here; only single-process jobs (type **standard**) and multiple-process jobs (type **pipe**) are of concern. Since a multiple-process job is basically the same as a pipe job without pipes, they both are of type **pipe**.

The second part specifies per process the name of the executable, the arguments and the file names for **stdin**, **stdout** and **stderr**.

The next part contains the number of pipes and, per pipe, a pipe description P_DESC which is listed below.

```
typedef struct {
     int     writer; /* index into cmd array */
     int     reader; /* index into cmd array */
     char    *name;  /* for named pipes, NULL otherwise */
} P_DESC;
```

The **writer** and **reader** are indices in the **cmd**-array and therefore specify the two processes which share a pipe. Although the wisc version does not run pipe jobs, the support to administrate such a job is present.

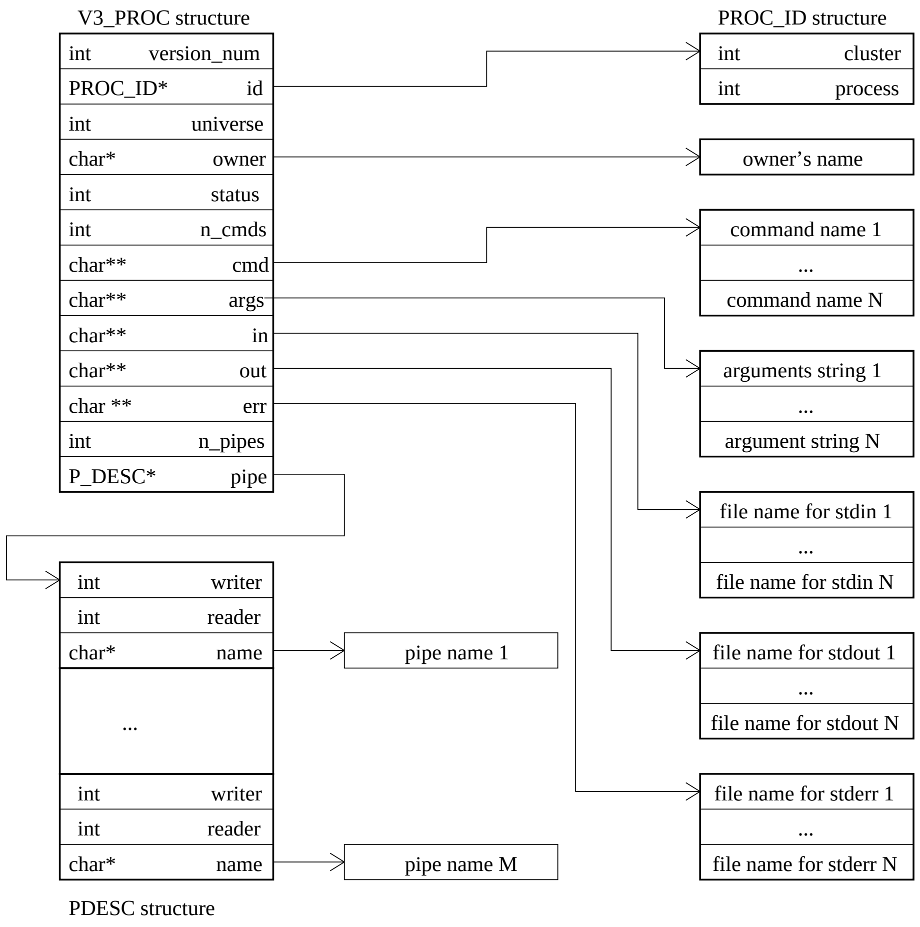

Figure 3.4: The new proc structure (*V3_PROC* structure).

The *V3_PROC* structure is sufficient for multiple-process jobs so nothing needs to be changed.

## 3.4 The Shadow

It is the Shadow which has no support for multiple-process jobs. It can handle requests from one client connection only; for this, we will change the Shadow as described below. We need not change the protocol with either the Startd or the Starter.

After the Schedd got permission to run a job, it starts the Shadow (Parent). The Shadow is invoked with the name of the remote host and the job's cluster and proc id. as arguments. The Shadow starts a child process, the Shadow Child, which will handle all remote systems calls. The actions that are performed by the Shadow (Parent and Child) are listed below.

**Shadow Parent**

```
 1) read job from job queue
 2) contact STARTD on the remote host
 3) send STARTD a START_FOREIGN_JOB request
 4) send STARTD the job's CONTEXT
 5) read two port numbers sent by the STARTD
 6) create  two stream socket connections using these port numbers
 7) create a pipe for communication with the Child
 8) start the SHADOW Child
 9) while (NOT received all status from SHADOW Child)
10)    read status on pipe
11) mail user
```

**Shadow Child**

```
1) while (NOT remote job has finished)
2)    poll all connections for requests
3)    for all connections
4)        if (has a pending request)
5)            handle request
6)            return result
7)    if a user process died
8)        send status to SHADOW Parent via pipe
```

For the Shadow Parent, we introduced step 9 and 10, so that it will now read status for all user processes instead of one. The Shadow Child is completely changed, except for step (5) and (8). These two steps are only slightly changed so that they accept a parameter that specifies for which user process a request should be handled (5) or should be sent status (8).

The Shadow Parent is responsible for making the connections with the Startd on the remote host. This connection is established via publically known port numbers. Using this connection, the Shadow Parent requests the Startd to start a job by means of a **start_frgn_job** command. With this request, it sends the **context** of the job, which contains the architecture and operating system for which this job is compiled. If the Startd is running on a machine with the same specifications, the Startd creates two communication channels and returns the port numbers that goes with them. The Shadow Parent uses the two port numbers to connect with the created channels; one channel will be used for handling remote system calls, the other for logging. Furthermore, the Shadow Parent creates a pipe which is used for communication with the Child. Then, the Shadow Child is forked and the Shadow Parent will wait until it receives information from the Shadow Child.

The Shadow Child should be viewed as a server for remote (pseudo) system calls. All user processes and the Starter should be viewed as clients. Therefore, the Shadow Child will wait until a request is made on one of the communication channels. It polls all channels for pending requests; if it finds one, it performs the request and sends back the result. If there are no pending requests, the Shadow Child is put to sleep. After a user process has terminated, and the Starter has sent information about its death, the Shadow Child examines whether this user process has terminated normally or abnormally. In both cases, the Shadow Child sends status information to the Shadow Parent through the pipe. Since we now have more than one process, before we send information, we send a number identifying the process. The information is:

- the reason the user process exited
- the image size of the user process
- the resource usage of the user process
- the directory in which the user process has created a core dump

In case of an abnormal termination (e.g., the user process made a segmentation fault), the whole job is terminated[3] and marked *completed*; it will never run again. After all user processes have terminated normally, the Shadow Child will terminate too. For multiple-process jobs, the individual processes are not related by any communication channel; therefore, when one user process terminates abnormally, other user processes may continue as normal. However, in pipe jobs, processes depend on one another, therefore, they cannot be run individually, only as a whole. We anticipate this behavior and do not allow other processes to continue when a user process terminates abnormally.

After the Shadow Parent has received all information about either all normally terminated user processes or *one* abnormally terminated user process, it will wait for the Shadow Child

[3] By sending a *kill foreign job* command to the Startd.

to die. Depending on the status information it has received, it updates the proc structure for this job and stores the structure in the JobQueue. If the job has terminated, either normally or abnormally, the Shadow Parent will mail the user about the completion of this job.

## 3.5 The Startd

Although we haven't changed the Startd (as it does not explicitly manipulate a job), it is described anyway. However, we will only focus on the interaction with the Shadow and the Starter; all other functionality of the Startd is omitted. One of the functions of the Startd, is to start the Starter which, in his turn, will start the user job.

When the Startd is started, it creates a socket using a publically known port number. The Startd is polling this socket to see if someone makes a connection on it. If so, it reads and performs the request: The accepted requests are:

- **start_frgn_job**: this request is made by a Shadow to start a *user job* on this machine
- **ckpt_frgn_job**: request to checkpoint the *user job*
- **kill_frgn_job**: request to kill the *user job*
- **req_new_proc**: request to start a new *user process*
- **starter_x**: request to explicitly use alternate Starter_**x** ( $0 \leq$ **x** $\leq 9$ )

**start_frgn_job** is the traditional request made by a foreign Shadow to start a job. The wisc version also allows a job to be started by means of a **starter_x** request[4]. In the file *condor_config*, which holds the global configuration of the Condor pool, one can specify alternate Starters by means of a **alternate_starter_x** = **executable name** statement. The traditional request will use, as normal, *condor_starter* in the Condor *bin* directory. In either case, the Starter will read the **context** of the job sent by the foreign Shadow. If this **context** matches the architecture and operating system of the host, the Startd returns the port numbers of two created communication channels[5]. As a last action concerning the start of foreign jobs, the Starter is started. Since the Startd does not need the communication channels with the Shadow, it closes them after starting the Starter.

The **ckpt_frgn_job** and **kill_frgn_job** commands may be requested by some Condor service program. Since such programs do not know how to contact the Starter (or the user process)

[4] Used for testing new Starters.
[5] As described earlier.

directly, they connect with the Startd by means of its publically known port and make the request. As a result, The Startd will signal the Starter.

Normally, the Startd starts the Starter which will request the Shadow for a proc structure, then, the Starter starts the user job. The **req_new_proc** request may be used to dynamically start new user processes on the remote host whenever the Starter is already hosting a remote job. After receiving this request, the Startd will signal the Starter. The Starter will react by requesting a new proc structure at the Shadow . Using this proc structure, the Starter will start one or more processes it finds in it. This special feature of Condor is of no interest for multiple-process jobs, as it is used for PVM; it is mentioned here for completeness only.

## 3.6 The Starter

The Starter is responsible for starting and controling the foreign user job. It is invoked with the name of the submitting machine as argument. The Starter is implemented as a finite state machine, which means that *control* is separated from *action*. Therefore, the Starter can be described with a state transition diagram. This diagram shows the states the Starter can be in and the events which cause a transition to another state. The original diagram that is included with the *wisc* Condor version is shown in appendix B. To describe the Starter, a simplified state transition diagram is shown here (Figure 3.5); for clarity, some asynchronous events, which may occur at any time (such as *die* requests), are omitted.

The names for events, states and actions in this picture match the names that are used in the code. Function names are in italics and asynchronous events are depicted with bold arrows. An event causes a transition from one state to another and with each event goes an action[6]. An event may be either synchronous or asynchronous. The asynchronous events are signals, usually sent by the Startd or by the Starter itself; these are listed in Table 3.1. Synchronous events occur depending on the state the Starter is in.

There are only a few changes needed for the Starter. Because the Shadow cannot handle multiple-process jobs the Starter will request the Shadow about one process only. We only need to change this behavior so that the Starter will fetch all the checkpoint files from the initiating machine instead of only one. Furthermore, the Starter can handle the failure when it couldn't get the information about a process or when it couldn't transfer a checkpoint file. In that case, the missing process could be added later. Again, for multiple-process jobs this behavior would be enough, but for pipe jobs it would not. When the Starter could not get the information about all user processes or could not get all checkpoint files, it will terminate, assuming an error.

[6]Actions for events are not depicted here, see appendix B.

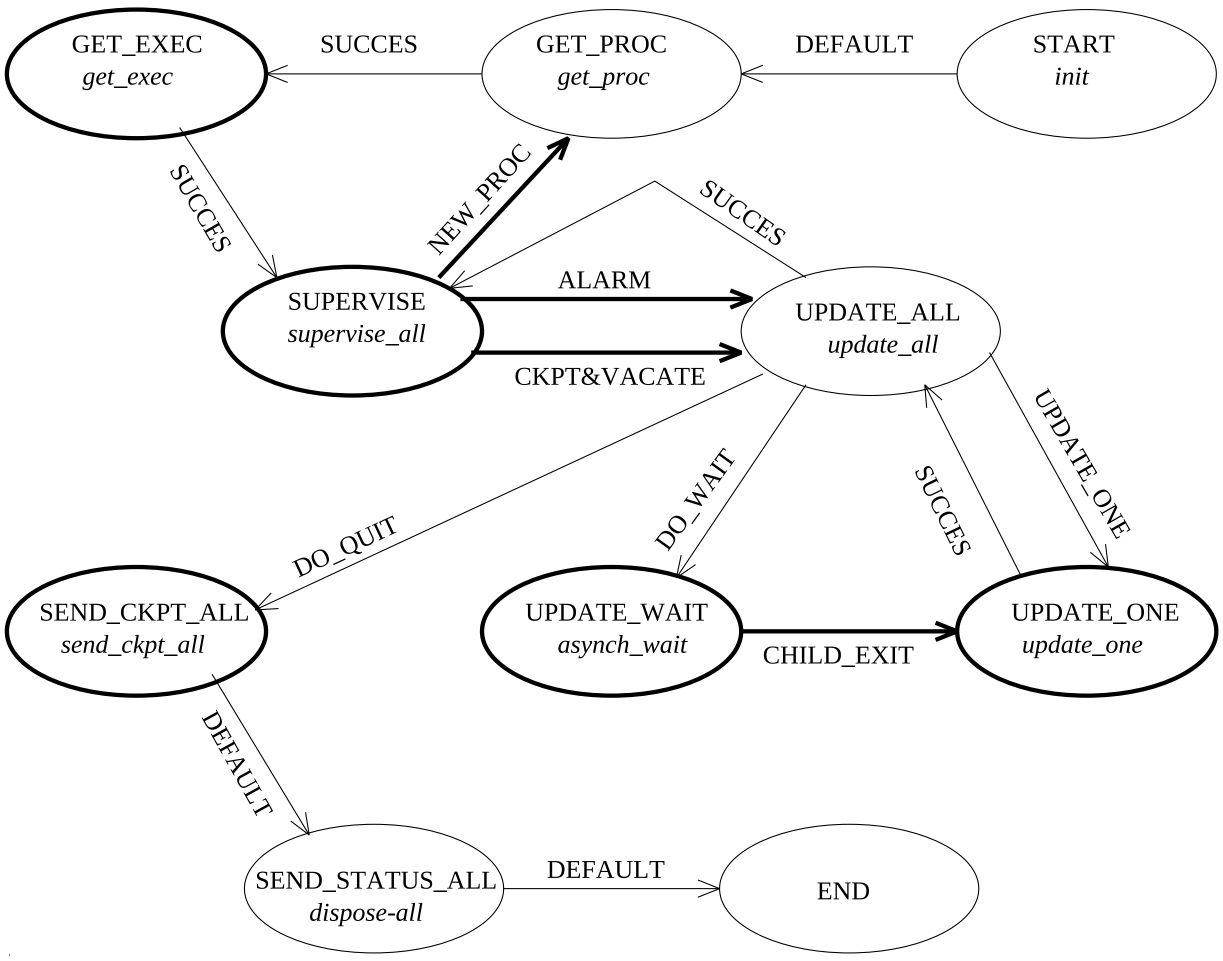

Figure 3.5: State Transition Diagram of the Starter.

| Event | Signal | Description |
|---|---|---|
| **get_proc** | **sighup** | start a new user process |
| **suspend** | **sigusr1** | suspend execution of all running user processes |
| **continue** | **sigcont** | resume execution of suspended user processes |
| **vacate** | **sigtstp** | user processes should migrate |
| **alarm** | **sigalarm** | alarm set for periodic checkpointing |
| **die** | **sigint** | kill all user processes |
| **child_exit** | **sigchild** | notification that a child process exited |
| **ckpt_and_vacate** | **sigusr2** | all user processes should checkpoint and migrate |

Table 3.1: Asynchronous events.

### 3.6.1 Starting a Job

We will now examine the state transitions that will occur when a multiple-process job is started. The Shadow has contacted the Startd and the Startd has started the Starter with the name of the initiating machine as argument. The actions performed at each state and the state transitions are listed below:

- **start**: *init* will perform initialisation actions such as: moving to the *execute* directory, setting the resource limits and closing unused file descriptors.
- **default** : does nothing
- **get_proc**: Function *get_proc* should send the Shadow as many **get_proc** requests as there are user processes[7]. The Shadow will respond by sending the proc structure for each process. Furthermore, it sends the name of the checkpoint file and the name of the file in which a new checkpoint should be stored. Finally, it sends the signal number which can be used for a *soft kill*, which only means that the Starter should set up a signal handler for this signal; it may be used to kill a user process immediately but in a clean way.
- **success** : does nothing
- **get_exec**: In *get_proc*, the Starter will fetch the checkpoint files for all user processes. It will make a symbolic link to a checkpoint file if it can be accessed directly. If not, the Starter will fetch it either via NFS or via the communication channel wit the Shadow.
- **success**: Function *spawn_all* will fork and exec all user processes.
- **supervise**: *supervise_all* will start the periodic checkpoint timer for all user processes and then waits for some asynchronous event.

[7]This is what need to be changed; originally, the Starter would make one request.

Each process requires a separate communication channel for remote system calls with the Shadow. Therefore, the Starter makes a **pseudo_new_connection** request with the Shadow for each user process. The Shadow opens a new communication channel and sends back the port number that goes with it. The Starter uses this port number to connect with the newly created channel. Now, when the Starter forks and execs a user process, this user process is born a communication channel with the Shadow.

### 3.6.2 Checkpointing a Job

Checkpointing a multiple-process job is done by checkpointing the individual processes. The Starter in the wisc version is capable of checkpointing each user process. We have not tested this, however, nor have we changed anything about the mechanism. Below, we will describe the state transitions that occur when the Starter checkpoints a multiple-process job.

Normally, the Starter is in state **supervise**. Checkpointing takes place whenever the checkpoint alarm expires or the Starter receives a **checkpoint_and_vacate** signal from the Startd, which probably means that the user reclaimed his machine and the job should move.

In both cases, the Starter moves to state **update_all** and suspends the execution of all user processes. Then, it sends the first user process a checkpoint signal. After this, it moves to state **update_wait** where it waits for the user process to exit. When this happens, the Starter moves to state **update_one** where it updates the checkpoint file for the user process that just has exited; from the old checkpoint file and the core dumped by the user process, a new checkpoint file is assembled. This new checkpoint file is immediately committed, that is, the old one is discarded. Now, the Starter moves back to state **update_all** and sends the next user process a checkpoint signal and performs the same loop. In this way, all user processes are checkpointed.

Finally, when all user processes have made a checkpoint, the Starter will move to either state **supervise** or to state **send_ckpt_all**. In the first case, all user processes are restarted. In the second case, the job should vacate and therefore, in state **send_ckpt_all**, the Starter sends all checkpoint files back to the Shadow. Then, it moves to state **send_status_all** where it sends for all user processes status information to the Shadow. Finally, it moves to state **end** which will cause the Starter to exit.

### 3.6.3 Termination of a User Process

Again, it must be noted that Figure 3.5 is a simplification of the original State Diagram; all kinds of asynchronous events are left out. The most important task of the Starter is to detect and investigate the termination of a user process, which is one of those asynchronous events.

The reason a user process terminates can be one of the following[8]:

- **normal termination**: This means that the user process has finished its computation and exits.
- **abnormal termination with a core dump**: The user process either violated the system or dumped core because of checkpointing. In the first case, this user process is not allowed to be restarted again; in fact, the whole job must be terminated and may never be restarted.
- **abnormal termination without a core dump**: Again, the user process violated the system.

Normally, the Starter sends status information about the termination of a user process immediately after such a user process has died, with the exception of a user process that terminated with a core dump because of checkpointing. In case of an abnormal termination with a core dump, the Starter will also transfer the core file back to the initiating machine.

## 3.7 Summary of Changes

We shortly list the changes that are applied to the Starter and the Shadow of the wisc version to fully support multiple-process jobs.

1. **Shadow Parent**:
   - read status for all user processes instead of one
2. **Shadow Child**:
   - maintain multiple connections instead of one
   - listen on multiple connections instead of one
   - wait for all user processes to die instead of one
3. **Starter**:
   - fetch a checkpoint file for each user process
   - start all user processes instead of one

[8]A *normal* or *abnormal* termination must be seen from a UNIX point of view. Condor regards a termination of a user process which dumped core for checkpointing as a *normal* termination.

Chapter 4

# The Design of Condor support for Pipe Jobs

When designing pipe jobs we must meet two requirements:

1. allow for a piping mechanism like the one in the shell
2. allow for the mapping of file names to the ends of a pipe

The first requirement is intuitive, as it is a very common and powerful mechanism in UNIX. The second requirement is more practical and allows for SMC jobs at CERN to run without changing a single line of source code. Note that these two requirements have different semantics. In the first case we redirect stdout to stdin via a pipe. In the second case, we allow piping of data between two processes which were programmed to read from and write to files. At run time, when the process tries to open a file which is to be piped to another process. We do not open the file but map it on a pipe end, we will call this a file-to-pipe mapping.

Furthermore, looking more closely at the jobs that run at CERN, we not only found out that these programs were written with hard-coded file names, but also that these names may be the same for different programs. This has the following consequences:

- When we map filenames to the end of a pipe, the names of the read and write end need not be the same. Typically, the name for an input file is *fort.10* and for and outputfile *fort.11*. Our syntax must allow to specify both names for each pipe.
- The processes in a job cannot run in the same directory, because the names of input and output files may be the same. If we want these files to be different for each process, we must run them in different directories. Any *chdir* system call issued by a user process

is performed (remotely) by the Shadow. As a consequence, all user processes share the same working directory. For the sake of the SMC jobs, the Shadow need to service each user process in its own *local* working directory. Furthermore, we must allow the user to specify the *initial* working directory for each user process.

For the pipe jobs we needed to change the Shadow, the Starter and the Condor library. The Shadow has all the information about a remotely running job. So, when on the remote machine (Starter or user process), information is needed about the pipes, a request to the Shadow should be made to obtain the information. This introduces new *pseudo* system calls to the system.

## 4.1 The Model and Assumptions for Pipe Jobs

In our model, we distinguish two kinds of processes: user processes and control processes. User processes are running user programs that were linked with the Condor library and control processes are part of the Condor Software (all Condor daemons, the Starter and the Shadow). Pipe jobs consist of one or more user processes, which all have a communication channel with the Shadow, and these user processes may be connected through pipes. Furthermore, unless explicitly stated otherwise, a pipe job runs as a whole on one machine.

Because we want to *checkpoint* pipe jobs we will briefly list the standard behavior of pipes in UNIX:

- Each pipe has a fixed-sized buffer of size PIPE_BUF[1], which varies on different machines. A Posix compliant should set PIPE_BUF to at least _POSIX_PIPE_BUF.
- There may be more than one reader and more than one writer of a pipe; if so, the pipe is shared between all its readers and writers.
- Writing data of size less than or equal to _PIPE_BUF to a pipe, is guaranteed to be atomic. An attempt to write more than this amount will result in writing as much as possible and putting the writer to sleep until the pipe gets drained by a reader.
- A read call on a *non-empty* pipe will result in reading the requested amount of data or, if fewer data are available, all the available data; in both cases the call returns the number of bytes read. A read call on an *empty* pipe will put the process to sleep until some other process writes data to it; whenever there is no process that has the pipe open for writing, the read call does not block and returns 0 to indicate end-of-file.

[1]This is a UNIX constant.

It is also possible to open a pipe (for reading or writing) with a *nonblocking* option, which means that a read or write call will never block.

The pipes which will be used between the user processes *must* be blocking. Remember that we allowed mapping of files on the ends of a pipe. Therefore, we must preserve the semantics of read and write calls on files when applied to pipes. For files, a read call will always be successful and will return the number of bytes read; however, when fewer data are returned than requested, end-of-file is reached. For pipes, whenever there are data available, a read call immediately returns (for both blocking and non-blocking pipes) with at most the amount of requested data; if fewer data are returned than requested, then no more data were available at the moment and one or more additional read calls should be performed to obtain all the requested data. Therefore, we should change the read call stub of a pipe so that it mimics the read call on a file. We must do this with blocking read calls so that the process is put to sleep when no data are yet available. Doing this with non-blocking read calls would actually mean that we are polling the pipe for more data; this is an unnecessary performance decrease.

Furthermore, we impose only two restrictions on pipe jobs, these are:

1. When a process has more than one file-to-pipe mapping, the names of the files must be unique. If they are not, we should translate a *write* call to the pipe into duplicated write calls for each pipe. Consider the example shown in Figure 4.1 in which process 1 writes to file *data* and process 2 and 3 read from it. All three processes have a separate pointer to the file indicating where the next read/write operation will take place. Since process 2 and 3 expect to read from a *file*, we must create separate pipe connections when we map file *data* to a pipe. Otherwise, the data written by process 1 is shared by process 2 and 3 according to the policy: "first come, first served", see Figure 4.2. As is shown, process 2 and 3 share the same read pointer. For the sake of simplicity, we do not allow such multiple file-to-pipe mappings. The same argument holds for piping stdout of one process to another; this may also happen only once.

2. No process may pipe data to itself; also for the sake of simplicity

There are no restrictions on the number of pipes per process, except for the total number of open objects per process, a standard UNIX restriction. Also, circular pipe topologies are allowed. Note that even though there is the danger for deadlock when allowing circular pipe connections, Condor cannot prevent the user from writing poorly designed programs.

## 4.2 The Specification of Pipe Jobs

Our syntax adds an extra command to the syntax of multiple-process jobs: `pipe-desc`. Furthermore, the initial working directory may be specified per process. A formal description

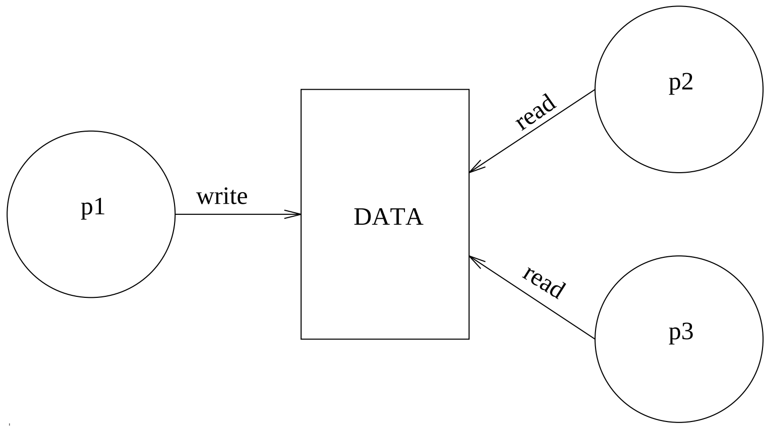

Figure 4.1: Three processes sharing a file.

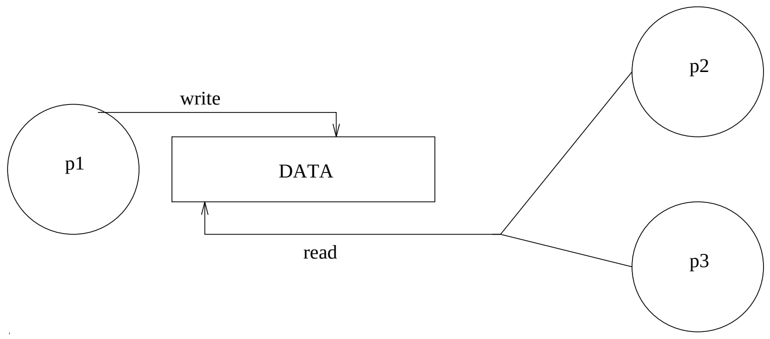

Figure 4.2: Three processes sharing a pipe.

is given below:

```
<jdf>            ::= <executables> <queue-list>

<queue-list>     ::= <command-list> <queue-command>  |
                     <command-list> <queue-command> <queue-list>

<command-list>   ::= <job-specific> <proc-specific>

<job-specific>    ::= <root-dir> <environment>

<proc-specific>   ::= <per-proc> | <per-proc> <proc-specific>

<per-proc>       ::= <input-name>  <output-name> <error-name>
                     <argument-list> <initial-dir>

<executables>    ::= "Executable" "=" <exe-list>

<pipe-desc>      ::= "Pipe" "=" <pipe-list>

<input-name>     ::= <exe-name> "." "Input" "=" <file-name> | <empty>

<output-name>    ::= <exe-name> "." "Output" "=" <file-name> | <empty>

<error-name>     ::= <exe-name> "." "Error" "=" <file-name> | <empty>

<argument-list> ::= <exe-name> "." "Argument" "=" <string-list> | <empty>

<initial-dir>    ::= "Initialdir" "=" <dir-name> | <empty>

<root-dir>       ::= "Rootdir" "=" <dir-name> | <empty>

<environment>    ::= "Environment" "=" <string-list> | <empty>

<queue-command> ::= "Queue"

<exe-list>       ::= <exe-name> <exe-list> | <empty>

<pipe-list>      ::= <simple-pipe> | <advanced-pipe> | <pipe-list> | <empty>

<simple-pipe>    ::= <exe-name> "|" <exe-name>

<advanced-pipe> ::= <exe-name> ">" <file-name> <file-name> ">" <exe-name>

<string-list>    ::= <string> | <string> <string-list>

<exe-name>       ::= <file-name>
```

```
<file-name>     ::= string
<dir-name>      ::= string
<string>        ::= char+ | empty
```

The same semantic rules for the specification of multiple-process jobs hold for pipe jobs. In addition the following semantic rules apply to the above syntax:

1. When a process has more than one pipe, all its `pipe-name`s must be unique.
2. The two `exe-name`s in a `pipe-desc` must be different.
3. When stdout of one process is piped to stdin of another process, no names for stdout of the first and stdin of the second process may be specified.

The first and the second rules are needed because of the two restrictions we imposed on pipe jobs. The third rule deals with the fact that both the | token and the `input-name` and `output-name` try to specify what to do with stdin and stdin. We cannot allow the user to specify both a piping command and a file redirection command for either stdin or stdout, as it is ambiguous.

Below are two example description files to show the use of both piping mechanisms.

```
#################################
#
#       Job description file
#
#################################

Executable        = foo bar
Pipe              = foo | bar

foo.Input         = inp
foo.Arguments     = one two three

bar.Initialdir    = /home/usr/john_doe/special
bar.Output        = results
bar.Arguments     = 1 2 3

Queue

################################


#################################
```

```
#
#       Job description file
#
#################################

Executable       = foo bar
Pipe             = foo > fort.11 fort.10 > bar

foo.Initialdir   = /home/usr/john_doe/directory_one
foo.Input        = inp
foo.Output       = data
foo.Arguments    = one two three

bar.Initialdir   = /home/usr/john_doe/directory_two
bar.Input        = samples
bar.Output       = results
bar.Arguments    = 1 2

Queue

################################
```

In the first example, we pipe `foo`'s stdout to `bar`'s stdin. Note that we explicitly specified an initial working directory for `bar`; since we did not specify one for `foo`, its initial working directory will be the directory from which the job is submitted.

The second example, shows the use of mapping file names to the ends of a pipe. When `foo` or `bar` tries to open file `fort.11` or `fort.10` respectively, we map them on the correct ends of the pipe. Note that we may specify names for stdout for `foo` and stdin for `bar`.

## 4.3 Process Structures

For pipe jobs, we need to extend the  proc structure so that it also contains the needed information about pipes. The wisc version already anticipated this with a pipe table in the *V3_PROC* structure. However, it only associated one name with each pipe; for our purposes, we need two names and so we extended the definition of the *V3_PROC* structure accordingly. The pipe table has as many entries as there are pipes and entry contains:

1. the name of the file that is mapped on the write end of the pipe
2. the name of the file that is mapped on the read end of the pipe
3. the index in the executable list of the process that reads from the pipe

4. the index in the executable list of the process that writes to the pipe

Furthermore, instead of *one* initial working directory, a *table* of initial working directories is included in the *V3_PROC* structure.

## 4.4 Setting up the Pipes

When using pipes on the remote machine, we can make use of either named or unnamed pipes. We shall use unnamed pipes and postpone the discussion about this choice until we discuss checkpointing pipe jobs in the next chapter.

The Starter will create the pipe connections for the user processes and when starting a child, these pipe connections are inherited. To create the pipe connections, the Starter needs to know the number of pipes and for each pipe:

- which pipe end belongs to which process
- what file descriptor should go with each end

Note that the Starter need not know what file names are mapped on what pipe ends; this is handled by the *open* stub of the Condor library.

In the philosophy of Condor, the Shadow should be the one and only process that has all the information about the job. Therefore, the Starter should make requests, by means of pseudo system calls, to obtain the information it needs. Unfortunately, the implementation of the wisc version is inconsistent with this philosophy. The Starter in this version will simply request the proc structure that goes with the job and thus has access to all information about the job, even to information it will never use. In our design of pipe jobs we will introduce a different approach that conforms to the philosophy. Ideally, the Starter should use the following protocol with the Shadow for starting a job:

1. Request job-type-independent information. Typically this may be: *type*, *id*, *number of processes*, *owner* etc.
2. Request job-type-dependent information. For a pipe job, for instance, the earlier mentioned information may be requested.
3. Request per-process information. This may be: *name of the checkpoint file*, *initial working directory*, etc.

It is not our intention to completely redesign the Starter, but we deal with step 1 and 2. We introduce two new pseudo system calls which will be described below. **PSEUDO_get_job** is used to get the following job-type-independent information (step 1):

1. The type of the job;
2. The id of the job;
3. The number of processes.

From the type of the job the Starter may tell what job-type-dependent information it should request the Shadow. For pipe jobs only, we introduce **PSEUDO_pipe_info** to obtain the following job-type-dependent information (step 2):

1. The number of pipes;
2. A table that holds for each pipe, the processes that read from and write to it (that is, the offsets in the executable list);
3. A table that holds for each pipe the file descriptor for the read and write end.

We will not change the behavior of step 3, the per-process information; we still request for the *V3_PROC* structure. However, we will request the same *V3_PROC* structure for each user process, pretending that it only contains per-process information.

Before a user process is started, the Starter will create the pipes and will use the *dup2* system call to duplicate the pipe ends to the file descriptor that is specified in the table it received from the Shadow.

## 4.5 The Runtime Model of Pipe Jobs

The runtime model of pipe jobs affects all system calls that manipulate either file names or file descriptors. Most important are : *open*, *read* and *write*. *open* is a generic function for opening objects which are associated with a UNIX path name, such as named pipes, files and devices. The open call returns the file descriptor (a small positive integer) that goes with the opened object. This file descriptor is used for further manipulation of the object. Opening an object and administrating the information is necessary for both checkpointing and remote system calls. Therefore we split up the discussion about the affected system calls in open, read and write operations, and other system calls. Also, part of the runtime model is the treatment of the standard files: stdin, stdout and stderr.

### 4.5.1 Standard Files

When a user process is started, the pipe connections are already open. As part of the initialisation, Condor (that is function MAIN in the Condor library) needs to known what to do with stdin, stdout and stderr. In the wisc version, the names of the files to which these standard files should be redirected were passed as an argument to the user process. So the first argument is the file name for stdin, the second for stdout and the third for stderr. In MAIN, these files are opened and duplicated to file descriptor 0, 1 or 2 for stdin, stdout or stderr respectively. Again, this implementation is in conflict with the design philosophy of Condor. It requires the user process to interpret the argument list for these files names and for the Starter to set up this argument list when it starts the user process (and thus need to know the names as well). So, the information about the names for stdin, stdout and stderr, travel from the Shadow to the Starter to the user process.

Instead, MAIN should request the Shadow what to do with the standard files. Therefore, we introduce yet another pseudo system call: **PSEUDO_open_std_file**, which is called by MAIN for each standard file separately. It can be thought of as a request to the Shadow: "What should be done with standard file x" (x=0,1 or 2, for stdin, stdout and stderr, respectively). Because of pipe jobs, the Shadow could respond with either "It is a file and has the next name ..." or "It is already open at the following file descriptor ...". In the first case, the standard file is redirected to or from a file. The second case tells us that the Starter already created a communication channel for the standard file. In case of pipe jobs, the communication channel can only be a pipe, but we like to keep things general; in the future, we could use sockets to redirect data to a process on another machine. That is why we refer to an anonymous object that is already open instead of a specific object, such as a pipe. Conforming to the design philosophy of Condor, the Shadow should determine what to do with stdin, stdout and stderr, after that, it sends back the result (either a file name or a file descriptor). The semantics of **PSEUDO_open_std_file** can be found in the next Chapter.

### 4.5.2 Open System Call

Before we discuss the new design of the open system call stub, we first look at some details.

Originally, Condor redirects the open system call back to the Shadow. The Shadow opens the file and sends back the file descriptor; the object is opened *remotely*. As an optimisation, Condor will see if it can access the object via the *Network File System* (NFS). We will not discuss how this is done, all we need to know is that if the object can be accessed via NFS, the object is opened *locally*. Which way the object is opened, is transparent to the user process. As usual, *open* returns the file descriptor which is used by the user process for further manipulations on the object.

Without any precautions, this may lead to file descriptor clashes, as is described next. Suppose

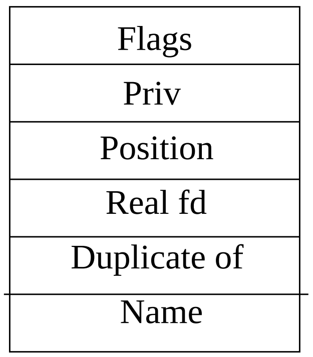

Figure 4.3: Entry in the VUFDT.

the user process opens a file named "foo" and assume that the file can be accessed via NFS. Then file "foo" is opened locally at, let's say, file descriptor 5. Then, the user process opens a file named "bar" and we now assume that this file cannot be accessed via NFS. Then the Shadow is requested to open the file and let's assume that also file descriptor 5 is returned. Then there are two objects referred to with the same file descriptor, which leads to erroneous events. Note that this example is not some rare case, it may happen all the time. File descriptors are actually offsets in a table of all open objects, the user file descriptor table. When a new object is opened, the kernel returns the offset of the first slot that is not taken. So it is very likely that when, for example, some files are opened locally (NFS) and some are opened remote (Shadow) that there will be overlapping file descriptors. This is an unwanted situation of course, and Condor's solution is the use of a so-called *virtual user file descriptor table*[2] (VUFDT). This table is used for both the mapping of *virtual* file descriptors to *real* file descriptors[3] and for administrating of all open objects for the need of checkpointing. When an object is opened (either locally or remotely), Condor does not return the *real* file descriptor, instead it returns a *virtual* file descriptor, which is the first non-empty entry in the VUFDT (note the analogy with the normal user file descriptor table, hence the name). For each system call that takes a file descriptor as argument, Condor first maps the virtual file descriptor to the real file descriptor before the system call is performed; the mapping takes place in the stubs of these system calls. The format of an entry in the VUFDT is showed in Figure 4.3.

In such an entry `Flags` is a bitfield that specifies the type of object. The wisc version uses the following bitmask:

- FI_OPEN: this object is open
- FI_DUP: this object is a duplicate of `Duplicate of`

[2]In neither literature nor sources has this table been given a name, so we will refer to it like this.
[3]The terms *virtual* and*real* file descriptors are also ours.

- FI_PREOPEN: this object is opened previously
- FI_NFS: this object was opened via NFS
- FI_RSC: this object was opened via remote system calls
- FI_WELL_KNOWN: well known socket connection with the Shadow

Important to note is that FI_NFS and FI_RSC are used to distinguish whether the real file descriptor refers to a *local* file descriptor or a *remote* file descriptor. `Priv` is also a bitfield that holds information on whether this object is opened for reading, writing or reading *and* writing; this is used for checkpointing purposes only. If the object is a file, `Position` stores the offset of the read/write pointer, also for checkpointing purposes only. `Real fd` holds the real file descriptor, either local or remote. Duplicating a file descriptor is handled by duplicating the virtual file descriptor instead of the real file descriptor. In that case, the FI-DUP bit is set in the `Flags` field and `Duplicate of` refers to the virtual file descriptor. Finally, if the object is a file, `Name` points to a string containing the absolute path name. To provide for the same semantics of files when mapping them to pipes, we need to alter some system calls. Also, we need to tell apart the file descriptors that refer to pipes and those that don't. So we introduce a new bitmask called FI_PIPE which will be used to mask the `Flags` field.

We now have all the information we need to discuss the open stub. The wisc version will first see whether or not the object can be accessed via NFS. If this is not the case, it will be performed by the Shadow. For our pipe jobs, the open system-call stub needs to ask the Shadow first whether the name of the object is a file that needs to be mapped on a pipe. For this request we introduce the **PSEUDO_file_2_pipe** call. The Shadow checks whether or not this is a file that needs to be mapped on a pipe end and sends back either true or false. When true, the Shadow also sends back the file descriptor at which the pipe end is open at; next we create a new virtual file descriptor for it, which is the return value of the open stub. When the pseudo system call returns false, the normal procedure is followed; first NFS, then the Shadow. In the wisc version, creating a virtual file descriptor is done by means of a function *MarkFileOpen*, which takes as argument the name of the object and the real file descriptor. This function will search for an empty slot in the VUFDT, store the appropriate information in it and return the offset of the slot as the virtual file descriptor. This mechanism is not altered.

### 4.5.3 Read and Write System Calls

To preserve the semantics of files, we need to alter the read and write system-call stubs. Reading and writing to a file are atomic actions; if they succeed, they succeed completely. This is different for pipes, for which, at least, the semantics of the read call allows that it returns with fewer data read than requested; the programmer should make additional read calls to read all the requested data.

The file descriptor that is passed as an argument to the read and write system call-stubs is virtual. The stubs should first test whether or not this virtual file descriptor refers to a pipe. If not, the stub will continue as usual, that is, when the virtual file descriptor refers to a local object, perform the local system call, otherwise perform the remote system call. If the virtual file descriptor does refer to a pipe, we should make as many local system calls as it takes to either read or write all data. Furthermore, each write call on the pipe is chopped into pieces of at most PIPE_BUF bytes because these are guaranteed to be atomic for pipes. The following pseudo C code will demonstrate how the read call is changed:

```
int
read(int v_fd, char* buf, int nbytes)
{
int r_fd,rval,count;
char *pc;

    r_fd=map_fd(v_fd);

                /* Start changed for pipe jobs */
    if (is_pipe(v_fd))
    {
        count=0;
        pc=buf;
        do
        {
            rval=local_sys_call(r_fd,pc,nbytes-count);
            count+=rval;
            pc+=rval;
        } while ( rval>0 && count<nbytes);
        rval=count;
    }
    else
    {
                /* End changed for pipe jobs */
        if (is_local(v_fd))
        {
            rval=local_syscall_read(r_fd,buf,nbytes);
        }
        else
        {
            rval=remote_syscall_read(r_fd,buf,nbytes);
        }
    }

    return rval;
}
```

Note that `local_syscall_read` and `remote_syscall_read` are called with the real file descriptor and testing on the type of object is done with the virtual file descriptor.

### 4.5.4 Other System Calls

Because we allowed file-to-pipe mapping, we have a problem with the differences in semantics between files and pipes. For example, files are randomly accessible objects, whereas pipes are not. So, not all operations that may be performed on files may be performed on a file that is mapped to the end of a pipe.

We shall give an overview of all system calls that manipulate with files and pipes. We shall make a distinction between system calls that take a *pathname* as an argument (this may possibly be a file) and system calls that take a file descriptor as an argument (this may possibly be a file that is mapped on a pipe). This distinction is necessary because from its name only, we do not know what kind of object we have. That's why system calls on pathnames are performed remotely by the Shadow. Table 4.1 shows for all the affected system calls on whether we should *change* (C) , *not change* (NC) or *not allow* (NA) them.

*fcntl* is marked for both "change" and "not allowed" because this call performs a variety of operations on a file descriptor; of these operations, some are allowed and the others are not. What to do with the system calls on file descriptors is quite obvious, as the object already exists (we have a file descriptor that identifies it). For the other category, this decision is not that simple due to the file-to-pipe mappings. For example, suppose the user specified that the (input) file "foo" should be mapped on a pipe, then file "foo" does not exist; when the process reads file "foo" it actually reads a pipe. Now what should happen if the program that opens file "foo" first uses *access* to test for its existence? Should we change the call so that it returns "it exists" or should we not change it and therefore it will return "it does not exist" . For this reason, except for *truncate*, we mark these system calls as "change", which should be interpreted as: "they *possibly* need to be changed[4]". Only *truncate* is not allowed as it is a common operation on files but is not allowed on pipes. The above list describes what should be done with these system calls, consult the next chapter to see what we did with them.

When an attempt is made to perform a system call that is not allowed according the above list, the user process which issued it should be terminated and therefore the whole job. Note that there is a difference in handling the termination of a user process after detecting an illegal system call. Illegal system calls on file descriptors are detected by the Condor library, whereas the detection of system calls on pathnames are detected by the Shadow.

[4]It is clear that *open* must be changed.

| System Call | Description | NC | C | NA |
|---|---|---|---|---|
| **File Desc.** | | | | |
| dup/dup2 | duplicate file desc. | X | | |
| fchmod | change permissions mode | X | | |
| fchown | change the owner | X | | |
| fstat | obtain statistics | X | | |
| fsync | move modified data to disk | X | | |
| ioctl | perform special func. on device | X | | |
| close | close an object | | X | |
| read/readv | read / multiple read | | X | |
| write/writev | write / multiple write | | X | |
| flock | apply or remove advisory lock | | | X |
| fcntl | perform special function | | X | X |
| ftruncate | set file to a specified length | | | X |
| lseek | move read/write pointer | | | X |
| **Pathname** | | | | |
| creat | create new file (it calls open) | X | | |
| access | determine accessibility | | X | |
| open | open an object given its name | | X | |
| lstat/stat | obtain statistics | | X | |
| chmod | change permissions mode | | X | |
| chown | change the owner | | X | |
| unlink | remove directory entry | | X | |
| truncate | set file to a specific length | | | X |

**Table 4.1** Table of affected system calls.

## 4.6 Serving Remote System Calls for Pipe Jobs

We already discussed the introduction of a few new pseudo system calls. These calls do not introduce any difficulty for the Shadow; there will just be some extra entries in the table of functions for remote system calls.

What still needs to be discussed is the issue of each user process needing its own working directory. For this, the Shadow keeps a table of directories, one for each user process, which will be initialised with the user process's initial working directories. For a single-process job, when the user process is started for the first time, function MAIN issues a *getwd* to request the initial working directory, immediately followed by a *chdir* to change to that directory. This may seem very strange, but remember that the Shadow is very likely not running in the initial working directory specified by the user. Since *chdir* is performed remotely by the Shadow, the Shadow will actually change to this directory. Also note that the Shadow Child (which is performing the remote system calls) runs under the UID of the owner, preserving normal UNIX file-access permissions.

For pipe jobs, this behavior will be the same for each user process. Furthermore, any time the Shadow performs a remote system call, it first changes to the working directory of the process who made the call. This has the overhead of changing directory for each remote system call.

## 4.7 Summary of Changes

Below, we briefly (only the major issues are mentioned) describe the changes for the Shadow, the Starter and the Condor library which are needed for pipe jobs:

1. **Shadow**:
   - serve each user process in its own working directory
   - support the new pseudo calls: **PSEUDO_get_job**, **PSEUDO_pipe_info**, **file_2_pipe** and **open_std_file**
   - maintain the open pipe table
2. **Starter**:
   - use the **PSEUDO_get_job** and **PSEUDO_pipe_info** to obtain information from the Shadow
   - set up the pipe connection for the user processes
   - do not start user processes with the names of their standard files as arguments
3. **Condor Library**:
   - do not interpret the arguments as the names of the standard file, but use **PSEUDO_open_std_file** instead
   - change open/read/write system-call stubs so that they may be used on pipes
   - change some system-call stubs (such as lseek) so that they terminate the process

# Chapter 5

# The Design of Checkpointing Pipe Jobs

In this chapter we will discuss checkpointing pipe jobs in general, furthermore, we will present the design of a checkpoint/restart algorithm for pipe jobs that run as a whole on one machine.

Pipe jobs require a different checkpointing technique than multiple-process jobs due to the communication links. A summary of checkpointing communicating processes can be found in [16].

Checkpointing is, in general, used for two purposes:

- **Process recovery**: In case of a machine crash, a process need not be restarted from the very beginning, but it is restarted from a previous checkpoint. For process recovery, the process needs to be periodically checkpointed.
- **Process migration**: Due to some policy (e.g., load balancing), it may be necessary to migrate a process from one machine to another. For this, we only need to checkpoint the process at migration time.

Although the checkpointing mechanism can be the same for both process recovery and process migration, the recovery (restart) mechanism is not. For process migration, a possible checkpoint/restart mechanism can be the following. First, the execution of the process is suspended, next, the process is checkpointed and moved to a new machine. In the mean while, we must deal with incoming messages addressed to the checkpointed process. On the new machine, before the process is restarted from its checkpoint, the communication links with other processes must be set up again and all pending messages must be redirected to the new machine. Now, the process may resume execution. For process recovery, the checkpointing mechanism is closely related to the recovery (restart) mechanism. Processes are periodically checkpointed and, after a failure, the failed process is *rolled back* to a previous checkpoint. Due to this rolling back, *missing* and *orphan* messages may occur. The main issue in process recovery mechanisms is to prevent these *missing* and *orphan* messages.

It is important to note the difference in at what moment a process is checkpointed. For process migration, this is at the time we decide to migrate a process. In process recovery we cannot predict when a failure will occur, therefore, periodic checkpointing is used . This difference is important for the restart mechanism; since in process recovery rolling back is required.

Now, let's examine the need for checkpointing in Condor. Two main goals of Condor are:

1. Condor only uses *idle* time: This means that whenever a user returns to his machine, that machine is, by definition, not *idle* anymore. Therefore, Condor should migrate a Condor job to another idle machine.

2. Jobs are guaranteed to complete: Normally, whenever a machine crashes, a user should restart his programs from the very beginning. For jobs that run a long time, this is unacceptable. Therefore, Condor periodically checkpoints a process and uses this checkpoint to roll the process back to in case of a failure.

In current Condor versions, for both process migration and process recovery, periodic checkpointing is used. This means that for process migration, a checkpoint/roll back mechanism is used. It takes too long to checkpoint a process at migration time, instead, an older checkpoint is used to restart. Until now, this was not a big problem because there was no inter-process communication. However, inter-process communication is present in pipe jobs. In the (near) future, Condor will be capable of taking a checkpoint at migration time, due to a new checkpoint mechanism. We must keep this mind when designing checkpoint/restart and checkpoint/rollback mechanisms.

## 5.1 The Centralised Approach

If we run a pipe job as a whole on one machine, see Figure 5.1, (i.e., centralised), we can checkpoint the job as a whole, that is, checkpoint each process. For both process migration and process recovery, it is required that all process are checkpointed. Using the checkpoints of the individual processes we can restart the job on another machine. There are, of course, additional actions to be taken in checkpointing the pipes. However, this is discussed in section 5.3; for now, it is enough to know that we need to checkpoint all processes and not a subset of them.

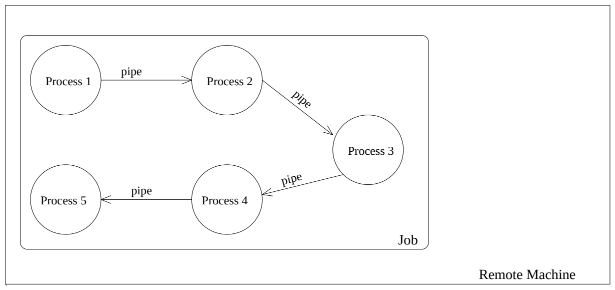

Figure 5.1: Running a pipe job on one machine.

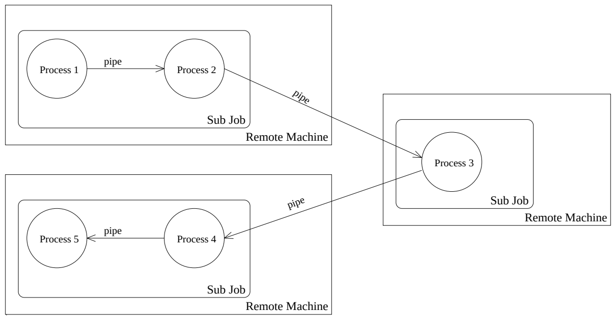

Figure 5.2: Running a pipe job distributed on three machines.

## 5.2 The Distributed Approach

If we run a pipe job distributed among several machines, we have split up the job in sub jobs. For example, in Figure 5.2 we run the same job as in the previous section, distributed on three machine and thus there are three sub jobs; two sub jobs consist of two processes and the third consists of only one process. Here, we can distinguish two kinds of inter-process communication, that is, *inter-machine communication* and *intra-machine communication*. For both process migration and process recovery, we do not need to make a distinction between a sub job and its individual processes. For process migration we should migrate an entire sub job, that is, all processes on the machine. For process recovery, we should recover all processes that crashed on a machine, thus the sub job. Therefore, we only need to view a distributed running job, as a job that consists of sub jobs with intra-machine communication. These sub jobs may be connected through inter-machine communication with other other sub jobs. For both process migration and process recovery we should checkpoint a sub job. This checkpointing technique is the same as for a centralised running job. However, the distributed approach has an additional problem, that is, how to deal with the inter-machine communication. Dealing with the inter-machine communication is much easier for process migration than for process recovery.

There is usually a tradeoff between checkpoint and roll back mechanisms. If one of them is cheap, the other will be expensive; this depends on:

1. amount of messages sent
2. number of processes involved
3. amount of disk space needed for storage
4. impact on the network
5. time needed for execution

The choice for a specific combination depends on the frequency of the invocation of the checkpoint and the roll back mechanism. In a highly reliable environment where errors do not occur often (and thus the invocation of the roll back mechanism), one would choose a cheap checkpoint mechanism and thus an (probably) expansive roll back mechanism. Therefore, it makes a difference whether we implement process migration with or without a roll back mechanism. The number of invocations of the roll back mechanism will be much higher if we implement process migration with a checkpoint/rollback mechanism. This affects the choice of a specific combination of checkpoint/rollback mechanism.

## 5.3 A Checkpoint/Restart Mechanism for Pipe Jobs: the Centralised Approach

The essential issue of checkpointing a pipe job is how to checkpoint the pipe connections between the user processes. Condor has already a mechanism to checkpoint a running UNIX process; we need not change this mechanism. However, before an individual process may be checkpointed, all pipe connections it has with other user processes have to be checkpointed as well. Therefore, we need to store the state of a pipe. We do this by emptying the pipe and storing its data in a buffer at the *reader* site of the pipe. When a user process reads the pipe and there are data in the buffer, we should first return these data; only when the buffer is empty, we will actually read from the pipe. After all pipes are emptied, the process is checkpointed in the usual way.

In the literature, the *initiator* is the control or user process that initiates the checkpoint algorithm. In our model, a user process will not act as the initiator; typically the initiator will be the Starter.

### 5.3.1 The Checkpoint Algorithm

Our checkpoint algorithm for pipe jobs is based on checkpoint coordination, which means that all user processes coordinate in establishing a *certain* state before making an individual checkpoint. This *certain* state is, in our case, the state in which all user processes know that there will be no more *writing* on any pipe. When this state is reached, the processes are said to be *synchronised*. Let us explain first how this synchronisation is done. For this purpose we will use yet another pipe, which we will call the *SYNC-pipe* and which is setup between the Starter and all user processes. The Starter opens this pipe for reading (when it starts up), while the user processes should open it for writing (at every re-start); thus the SYNC-pipe has *one* reader and as many writers as there are user processes. This pipe will never be used to transfer any data, only to synchronise.

The outline of the algorithm is as follows:

1. The Starter sends *all* processes a **prepare_for_checkpoint** signal; next, it will perform a read call of *one* byte on the SYNC-pipe. Since this pipe is empty, the call will block until someone writes to it *or* until all writers have closed their pipe ends. Upon the receipt of a **prepare_for_checkpoint** signal, a user process suspends its execution and closes its end of the SYNC-pipe. When *all* user processes closed their ends of the SYNC-pipe, there are no more writers and therefore the read call of the Starter will return with zero bytes read. Now, all user processes are synchronised.

2. After the Starter returns from its read call, it checkpoints all user processes one by one (in any order) by means of sending a **checkpoint** signal. Upon the receipt of a **checkpoint** signal, a user process first empties all pipes it has open for *reading*. For each pipe, these data are stored in a separate buffer. It may now continue in taken a checkpoint in the usual way, which will result in the user process sending itself a signal to produce a core file.

3. After the Starter sends a user process a **checkpoint** signal it waits until the user process dies. In UNIX, a parent process is sent a *SIGCHLD* signal whenever one of his child processes has died. Since the Starter is is the parent processes of all the processes in a pipe job, it will be notified when one of these child processes dies. The Starter will then perform a *wait* system call to examine which child process died and the reason it had died. If the child that died was sent the **checkpoint** signal and the child died because of checkpointing, the Starter updates the user process' checkpoint file, using the old checkpoint file and the core file. Now the Starter sends the next user process a **checkpoint** signal.

4. The newly created checkpoint files are *tentative* until all processes have checkpointed, then they will be *permanent*. After all tentative checkpoint files are turned into permanent ones, the checkpoint (of the pipe job) is said to be *committed*. Whenever an error

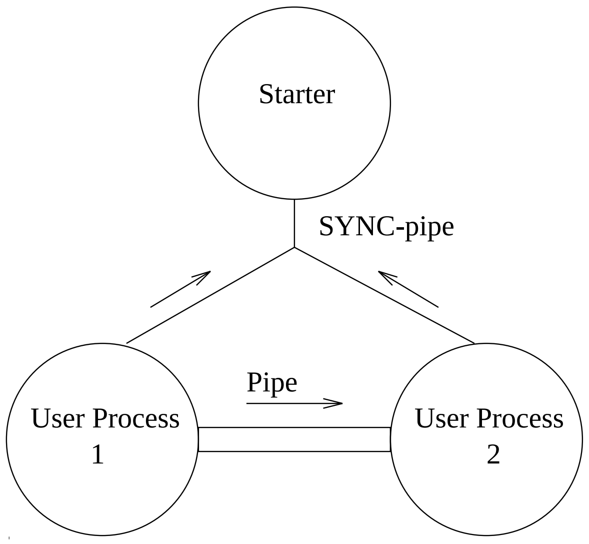

Figure 5.3: Situation just before checkpointing.

occurs before the checkpoint is committed, all tentative checkpoint files are discard and the initiator must invoke the roll back algorithm.

It is illustrative to demonstrate this mechanism with the next example. There are two user processes that share a single pipe; process 1 writes to it whereas process 2 reads from it. The situation just before the invokation of the checkpoint algorithm is shown in Figure 5.3; the arrows show the read/write direction on the pipes.

The Starter first signals both process 1 and 2 a **prepare_for_checkpoint** signal and performs the read call on the SYNC-pipe, see Figure 5.4 . Upon the receipt of this signal, process 1 and 2 both suspend their execution and close their ends of the SYNC-pipe. Since there are no writers on the SYNC-pipe anymore, the Starter returns from its read call and synchronisation is established, see Figure 5.5. Now, the Starter may checkpoint both processes as usual, so first process 1 is sent a **checkpoint** signal, Figure 5.6. Since process 1 does not have any pipes open for reading, it does not need to perform any special actions but can checkpoint immediately, resulting in a core dump. The Starter will notice that process 1 has dumped core and create a new checkpoint file for the first process from its old checkpoint file and the core file. Then, the Starter sends process 2 a **checkpoint** signal, Figure 5.7. Since process 2 has a pipe open for reading, it must empty the pipe before making a checkpoint. It therefore reads the pipe until no more data are available and stores this data in a buffer, Figure 5.8. Now, process 2 will make a checkpoint in the usual way. The Starter will notice that process 2 dumped core and will create a new checkpoint file for this process as well. The last action will be to commit the checkpoint by making all tentative checkpoint files permanent.

Before we go on with the discussion, two things need to be said. The first is how a pipe is

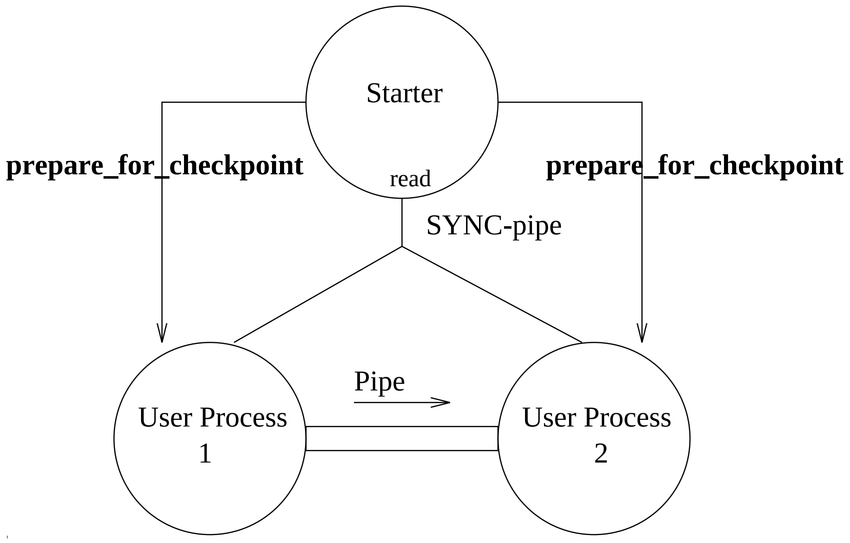

Figure 5.4: Starter sends **prepare_for_checkpoint** signals.

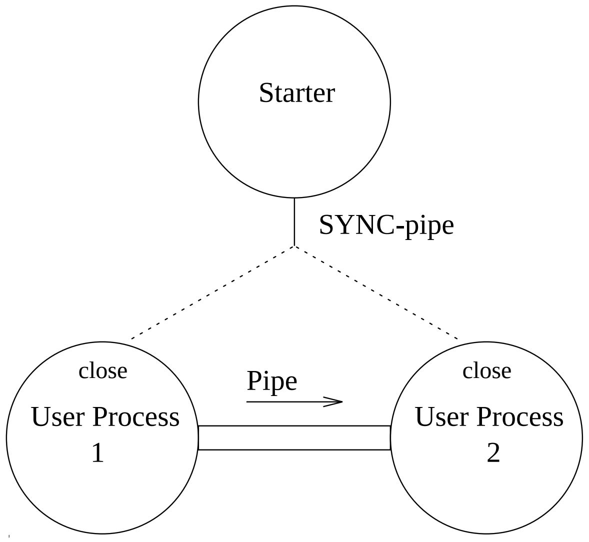

Figure 5.5: Synchronisation is established.

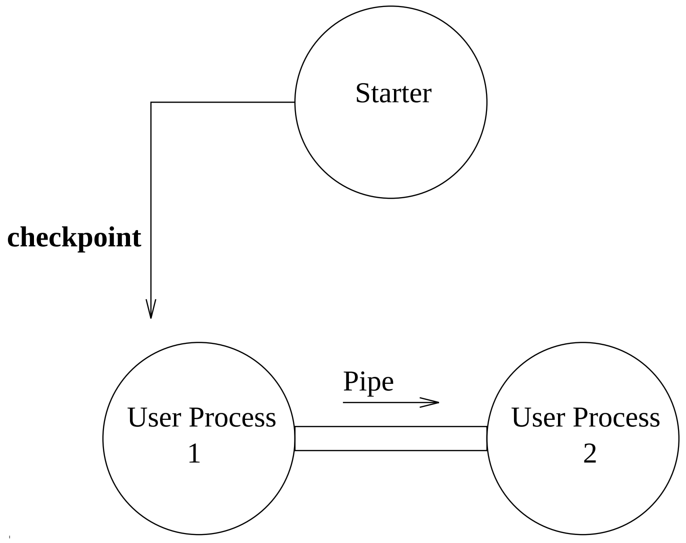

Figure 5.6: Process 1 is sent the **checkpoint** signal.

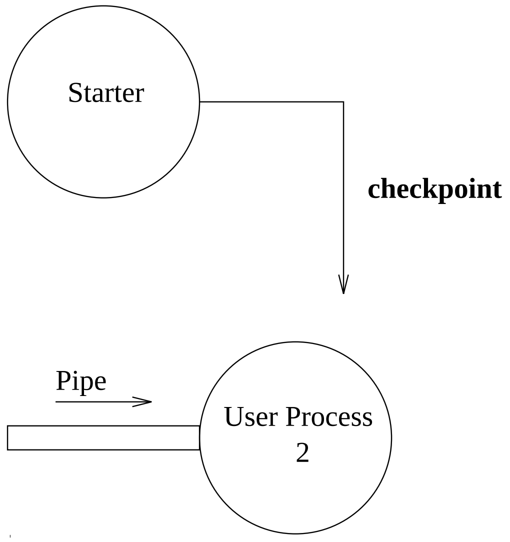

Figure 5.7: Process 2 is sent the **checkpoint** signal.

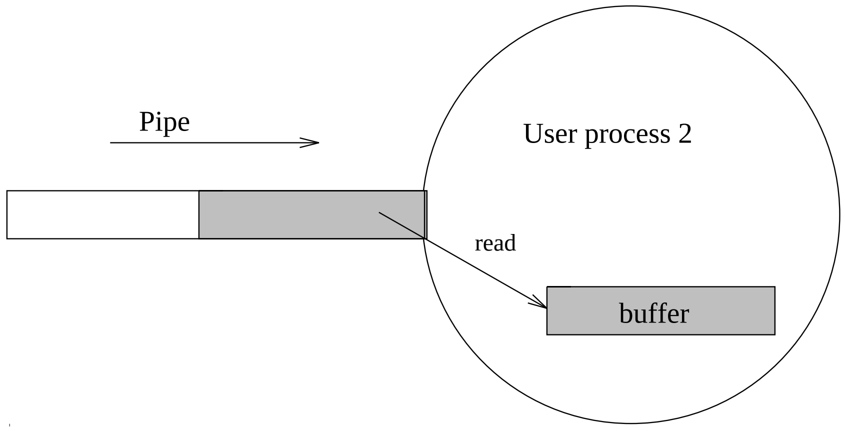

Figure 5.8: Process 2 empties its pipe.

emptied and the second on how to commit all checkpoint files. When a user process receives a **checkpoint** signal, it should empty all pipes it has open for reading. It could empty a pipe by just performing one or more read calls on the file descriptor of this already opened pipe. However, since this pipe is open for blocking I/O (we already explained why), such a read call might block if the pipe is already empty and there is some other process that has this open for writing. This leads to a deadlock, that is, this user process is blocked, waiting for someone to write or close this pipe and all other user processes are waiting for this user process to finish checkpointing. We must therefore either close all ends of the pipes that are open for writing (in all user processes) or perform an non-blocking read call on this pipe. Let's discuss both alternatives:

1. Close all write ends: This can only be done by an extra step (no synchronisation needed) in which the Starter sends all user processes a **close_write_ends** signal; this signal should be sent *after* synchronisation and *before* any process is sent a **checkpoint** signal. Signals are guaranteed to be delivered and therefore, a read call on a pipe, in order to empty it, will eventually return because in finite time all write ends of the pipe will be closed. It is important to note that closing the write ends of a pipe for this purpose may only be done *after* synchronisation. Suppose we let a user process close all pipe ends it has open for writing after it closes the SYNC-pipe, that is, after receiving the **prepare_for_checkpoint** signal but before the synchronisation is accomplished. It now may happen that another user process which not yet received the signal is woke up from a blocking read call on one of the pipes because the first process just closed (the only) write end of the pipe. This leads to the situation that the user process returns from a read call with zero bytes read and therefore assumes that the writer has finished its computation.

2. Non-blocking read call: Since all pipe ends are created with the *blocking* option, we must temporarily set the pipe end in a *non-blocking* state. Then, we can read blocks

of data upon the pipe until the pipe is empty, that is, a read call returned zero bytes. Due to the synchronisation, we know for sure that no more data will be written on the pipe after we emptied it. After the pipe is empty, the pipe is reset to the *blocking* state. Currently, the process will immediately die for checkpointing, in the future, a process may continue after checkpointing. We reset the pipe's state to anticipate for the new checkpoint mechanism.

We will use the non-blocking read call to empty a pipe, this saves the sending of an extra signal to all user processes.

Committing the checkpoint files is the last action of our checkpointing algorithm. If we copy or move each new checkpoint file to the old checkpoint file, we destroy old current checkpoint files before we actually committed the checkpoint. A crash during the commitment would result in a wrong set of checkpoint files, therefore, after a crash we must check to see whether we crashed during a commitment. Commitment takes place twice; once by the Shadow on the initiating machine, and once by the Starter on the execution machine. A commitment by the Starter is obvious since the Starter is the initiator of the checkpoint algorithm. A commitment by the Shadow is needed because on the initiating machine a recent and consistent set of checkpoint files must be stored. Whenever a remote running job should vacate it must store its most recent set of checkpoint files on the initiating machine, then, after the job is assigned a new machine to run on, this set of checkpoint files is transferred to the new machine. Therefore, the Shadow must check whether or not a set of checkpoint files is consistent on the *initiating* machine. Whenever a job is restarted, the Shadow must first check for an incomplete commitment; if so, it must finish this commitment before transferring any checkpoint file. It has no use to check whether a crash on the *execution* machine occurred during a commitment and thus, the job is rolled back from the set of checkpoint file that are stored on the initiating machine. A check on the execution machine could be done by the Starter or perhaps even the Startd, but suppose, in the worst case, the execution machine was down for days and the job is already restarted from the set of checkpoint files stored on initiating machine, then it may be possible that the job has already finished its computation. Therefore, such a check would be useless.

### 5.3.2 The Rollback Algorithm

The algorithm used to roll back is quite simple. The latest set of checkpoint files of all user processes is always consistent. Therefore rolling back means that every user process is restarted from its latest checkpoint file. As usual, the Condor code of a user process performs the needed actions to continue the user process from where it left of just before checkpointing; only one extra action is needed: open the SYNC-pipe for writing in order to be able to synchronise for checkpointing again.

In the previous chapter we stated that we would use unnamed pipes for our connections between the user processes, now it's time explain why.

If we use named pipes, we have some difficulties in restarting a pipe job. When a user process is restarted it should reopen all pipes that were opened before checkpointing. Now suppose we are restarting a process that had a pipe open for *reading* before it was checkpointed, then this pipe has to be reopened. If we do that, the process is blocked until a *writer* opens the pipe as well. Now, suppose that the other process, that *writes* to the pipe, has finished its computation just before checkpointing, then the pipe is never opened for writing and thus we have a deadlock. We could also open the pipe with a non-blocking option, so that the deadlock will not occur, but then there is the possibility that the process starts reading from the pipe before the write process opened the pipe. As a consequence, the read process will read return from its read call with zero bytes read, indicating EOF. These problems can be solved in two ways. The first solution would be to add an extra synchronisation step: all user processes are started and open their pipes with a *non-blocking* option, then they should wait until all processes reached this state, then they may continue[1]. The other solution would be to let the user process request information (at the Shadow) about how to open the pipe: with either a blocking or a non-blocking option. Because the Shadow knows whether or not the other process that shares the pipe has already finished it will respond accordingly: if the other process has finished, with the *non-blocking* option; if the process has not finished, with the *blocking* option.

When using unnamed pipes, both ends of a pipe are created before a single user process is started. The above mentioned problems will not occur so no extra actions are needed.

[1] After they reset their pipes in a *blocking* state.

# Chapter 6

# Implementation

This chapter describes how the pipe version is implemented. The implementation of both multiple-process jobs and pipe jobs, required changes all over the source code of the wisc version. We tried to keep the modifications and extensions as simple and clean as possible. For the Shadow, this is was not an easy task as it was hard-coded to act as a server for one process only. We would have liked to re-implement the Shadow but there was no time to do this properly; instead we modified it extensively.

Unless explicitly stated otherwise, everything that is described in this chapter refer to the things we changed in the wisc version.

## 6.1 Multiple-Process Jobs

The changes for multiple-process jobs primarily affects the Shadow and the Starter. The Shadow is implemented in old-style C (Kernighan and Richie), whereas the Starter is implemented in C++.

### 6.1.1 The Shadow

In order for the Shadow to maintain connections with all user processes and with the Starter (these are the clients), it keeps a table of communication channels, which is an array of the following structure:

```
    struct
    {
        XDR xdr_stream;
        int sockfd;
        int port;
    } connection_t
```

Condor makes use of the XDR (eXternal Data Representation) library to send and receive data. For each client, the port number, the file descriptor of the socket connection and an XDR variable is kept. The relationship between the three is that the socket connection was made using the port number and the XDR stream uses the socket connection to send to and receive data from.

The table is dynamically allocated when the Shadow Parent knows the number of clients; thus after reading in the  proc structure for the job. The number of client connections is equal to the number of user processes plus one (the Starter has a separate connection, which is stored at offset 0 in this table).

Only the Starter in the wisc version had support for the **PSEUDO_new_connection** call to request a new connection on behalf of the user processes. We have changed the Shadow in the following way to support the call as well: When the Shadow Child starts up, it has only one client, the Starter. The Starter will request for each user process a new socket connection, by means of the **PSEUDO_new_connection** call. The Shadow Child will create a new socket connection and sends back to the Starter the port number that goes with it. The Shadow Child will then wait for the Starter to make a connection, by issuing a *listen* system call on the socket. The Starter will issue a *connect* system call on the port number it received from the Shadow Child and the connection is established. By connecting, the Starter created a socket, which will be duplicated a fixed place, just before it starts the user process for which this connection was made. When the user process starts, MAIN in the Condor library will create an XDR stream with the socket it inherited from the Starter. When the Starter connects on the Shadow Child's port, the Shadow Child will return from its *listen* call and is return a file descriptor for the established socket connection. With this socket, the Shadow also creates an XDR stream and administrates this new client connection in the table of connections. The socket connection that was used to listen for the Starter to connect is closed.

The Basic outline of our implementation of Shadow Child can be given with the next two functions, written in pseudo C code:

```
Handle_System_Calls()
{
    while( 1 )
    {
        number_pending=wait_for_requests();

        for(i=0 ; i<number_of_client_connections ; i++)
        {
            if (has_a_pending_req( i ))
            {
                rval=do_REMOTE_syscall( i );
                if ( rval == -1 )
                {
                    handle_proc_termination( dead_user_proc );
                }
            }
        }
    }
}
```

What we see, is that the Shadow Child performs an infinite loop in which it waits for requests. `wait_for_requests` will perform a *select* system call on all the sockets of client connections that are open. *Select* will block until, on at least one socket, new data arrive (the number of the pending requests are returned for logging purposes only). Next, we loop over all clients to see if it has a pending request. If so, the (pseudo) system call is performed by `do_REMOTE_syscall`. `Rval` is not the return value of the performed system but a value specifying whether or not the client has finished its computation. Below, `do_REMOTE_syscall` is described with pseudo C code. This function already existed but we changed it and gave it a parameter to identify the client to perform the call for. All new pseudo system calls we introduced are implemented in this function.

```
do_REMOTE_syscall( int i )
{

    syscall_num=xdr_int( i );

    switch(syscall_num)
    {

        case SYSCALL_open:
            .....
            .....
            rval=1;
            break;

        case PSEUDO_new_connection:
            .....
            .....
            rval=1;
            break;

        .....
        .....
        case SYSCALL_exit:
            .....
            .....
            rval= 1;
            break;

        .....
        .....

        case PSEUDO_subproc_status:
            .....
            .....
            rval= 1;
            break;

         .....
         .....

         default:
             /* ERROR, unknown system call number */

    }
    return rval;
}
```

As is shown, first the number of the system call is read from the XDR stream. Normal system calls have positive numbers, the pseudo system call have negative numbers starting from -1. Except for the **PSEUDO_subproc_status** call, `Do_REMOTE_syscall` will return 1. When a user process finishes its execution, the last system call that it performs is *exit*, and so when receiving a request for this call, the Shadow Child closes the connection with this client. The Shadow Child will handle the termination of the user process when the Starter issues a **PSEUDO_subproc_status** call with as argument, the id of the process that dies. A client does not necessarily have to terminate via an *exit* call. When it terminates abnormally, *exit* will very likely not be performed. The Shadow does notice, however, that it lost the connection with such a client; in that case the Shadow marks the connection as being closed. Eventually, the Starter will update the Shadow Child with information about the cause of death.

### 6.1.2 The Starter

The changes for the Starter were minimal. Most important is that it needed to fetch the information for all processes, instead of only one. All other support to host a multiple-process job was already present.

## 6.2 Pipe Jobs

In order to support pipe jobs, we changed the proc structure; as a consequence, we needed to change all functions that referred to either the pipe table (we introduced a second name for each pipe) or the initial working directory (we introduced per-process initial working directories). Because the proc structure moves around the system via XDR, the implementation of `xdr_proc`() that either reads a proc structure from or write a proc structure to an XDR stream is also changed.

### 6.2.1 The Shadow

For pipe jobs, only the Shadow Child is changed. These changes are listed below:

1. support the new pseudo system calls: These pseudo calls we introduced are: **PSEUDO_get_job** and **PSEUDO_pipe_info** (requests made by the Starter), **PSEUDO_file_2_pipe** and **open_std_file** (requests made by the Condor library). For the first and the last pseudo call, the information is obtained from the proc structure, for the second and the third call, we introduce a new table for the pipes, which is described in the next item.

2. new data structure to administrate pipes: Note that the proc structure also contains a table with information about the pipes. However, we need to store additional information, so instead of spreading the information among two tables, we will create a new one. We will refer to this table as the *open pipe table* (OPT); the format of an entry from this table is shown below:

   ```
   struct
   {
       int writer;
       int reader;
       char *w_name;
       char *r_name;
       int w_fd;
       int r_fd;
   }
   ```

   The new information, are the file descriptors at which both pipe ends are expected to be open.

3. operations on the new data structure: The open pipe table is initialised when the Starter issues the **PSEUDO_pipe_info** call. The value of the two file descriptors in each entry of the open pipe table are calculated as follows: if a process has n pipe ends, then they

are assigned a value: base, base+1, ..., base+n, where *base* is the lowest file descriptor at which a pipe end may be open. After all file descriptors are assigned a value, the OPT (without the names of the pipes) is sent to the Starter.

The Shadow Child will consult the open pipe table for the **PSEUDO_file_2_pipe** call. The format of this call is:

```
int rval=file_2_pipe(char *name, int *fd);

rval=0:    object is a pipe, open at fd (*fd)
otherwise: object is not a pipe (*fd=-1)
```

The open stub will call this pseudo system call with the name of the object to be opened. The Shadow Child will lookup in the OPT to see if the user process has a pipe end with the specified name. If so, it sets `fd` to the according value and returns 0, if not, it sets `fd` to -1 and returns 1.

4. serving each client in its working directory: To serve each process in its own working directory, the Shadow Child changes directory just before performing a system call. Function `do_REMOTE_syscall` is changed in the following way:

```
do_REMOTE_syscall( int i )
{

               /* start changed */
               /* WorkDir: table of directories */
    if (i>0)
    {
         chdir(WorkDir[i-1])
    }
               /* end changed */

    syscall_num=xdr_int( i );

    switch(syscall_num)
    {

        .....
        .....

    }

    return rval;
}
```

Note that client 0, is the Starter and we need not change directory for it; we don't let the Starter use relative pathnames.

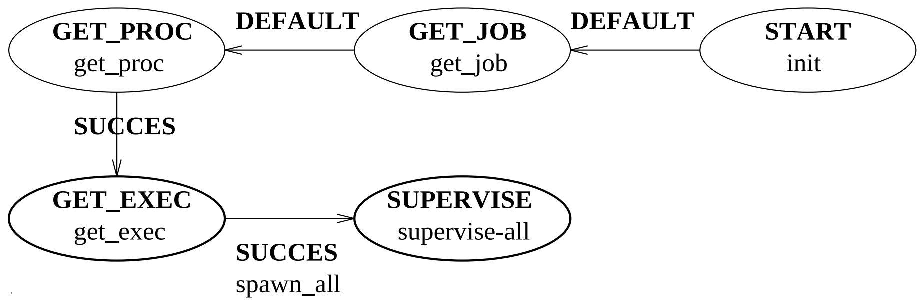

Figure 6.1: Starting the job.

### 6.2.2 The Starter

For the Starter, we took advantage of the fact that it is written in C++ and is implemented as a finite state machine . The new mechanism we introduced as part of starting a new job, is implemented as *states*. When the Starter is started this will result in the state transitions as are shown in Figure 6.1. In the appropriate states, the according pseudo system calls are performed to obtain the information from the Shadow.

The wisc version administrates information about user processes in objects instantiated from class *UserProc*. This class has a constructor which takes a pointer to a proc structure as argument, which is used to copy information about this user process to local data members. Class *UserProc* has methods for all necessary manipulations on the user process. For example, if *objUser* is an object of this class, then *objUser->execute*() is used to start the user process; the implementation of *UserProc::execute* will perform a combination of *fork* and *exec* system calls. The basic idea of this administration is, of course, great. However, the extensions that were made, are not. As described in section 3.3, the wisc version supports different types of jobs. Each type of job will use objects instantiated from class *UserProc* to store the information about the individual user processes. As a consequence, the implementation of the methods of class *UserProc* contains numerous *if (type==...) then-else* statements to possibly do something different for each type. This is definitely not the object-oriented (OO) way of design and implementation. We did not implement pipe jobs in the manner as described above. Instead, we have initiated the OO approach in implementing different types of jobs.

Formally, class inheritance in the OO approach is only allowed when an *is-a-special-kind-of* relationship exists between two classes. That is exactly the case with the user processes of different types of jobs. A process of a pipe job *is-a-special-kind-off* a normal process. That is why we inherit class *PipeProc* from class *UserProc* and override the methods of class *Userproc* to perform the extra actions which are needed for processes that belong to a pipe job (such as setting up the pipe connections). And that is why *other* types of jobs should be implemented in the same way! (we will not list the definitions of either of the classes, the interested reader is referred to the source code) Most important in the manipulation of a user process, is starting it; i.e., calling UserProc::execute(). In the wisc version, this is typically a method that contained a lot of *if(type=....)* ... It is illustrative to look at the new method *execute* (simplified).

```
void
UserProc::execute
{
    char    *argv[ 2048 ];
    char    **argp;
    char    **envp;
    int     user_syscall_fd;

    create_arglist(argv);           (*)
    envp=env_obj.get_vector();

    initialise_child();             (*)

    if ( (pid=fork())==0)
    {
        manipulate_fdt();           (*)

        install_signals();          (*)

        do_exec(argv[0],argv,envp); (*)
    }

    cleanup_after_child();          (*)

    state=EXECUTING;
}
```

The marked functions (*), introduced by us, are in fact virtual methods of class UserProc. This allows for inherited classes, such as our class PipeProc, to override the behavior of the base class. For example, method *initialise_child*, which is called just before *fork* will do nothing for class UserProc and return immediately, whereas for our class PipeProc, it will setup the pipe connections for this user process.

We also introduced a new class: *Job*, which contains the information received from **PSEUDO_get_job** call. This is also for a more intuitive OO approach when dealing with job manipulations.

Due to the pipe jobs, the individual processes are artificially synchronised by the pipe connections. As a side effect, these processes terminate close after one another. The wisc version's detection mechanism for a child's termination was not able to keep up with pipe jobs. In the wisc version, signal SIGCHLD, which is sent by the kernel to a process whenever one of the

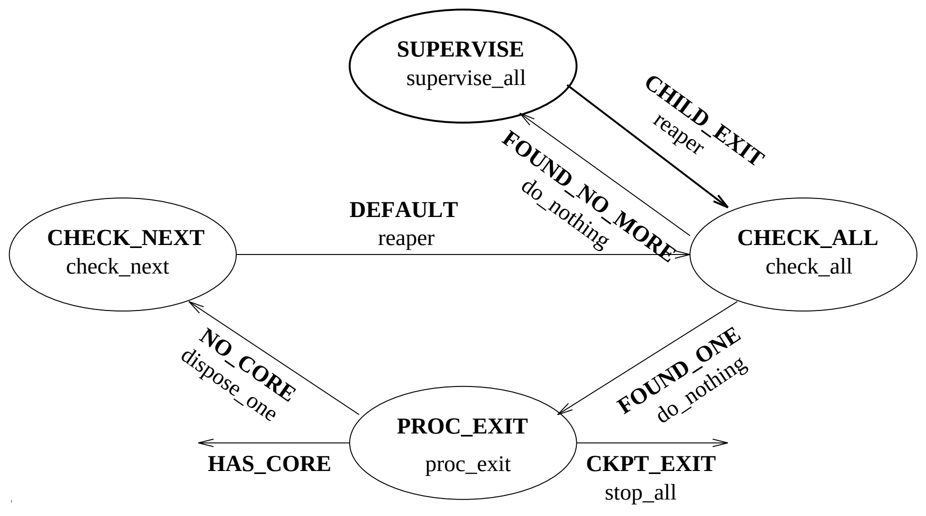

Figure 6.2: New state transitions for reaping children.

process' children died, is associated with the asynchronous event **child_exit**. Normally, when the Starter has started all user processes, it remains in state **supervise** waiting for some asynchronous event to occur. Whenever a user process dies, the state machine driver catches signal SIGCHLD and generates event **child_exit**. This causes a state transition to state **proc_exit** and then to either state **send_core** or back to **supervise** depending on whether the child dumped core or not[1]. The functions that are called along the way, transfer information back to the Shadow, explaining the cause of death. From the time the SIGCHLD is catched until the time the Starter gets back to state **supervise**, signal SIGCHLD is blocked. In UNIX, the kernel associates *one* bit with each signal. When a process is sent a signal, the kernel sets the appropriate bit. As soon as the user process handles this signal, by either catching or ignoring it, the bit is cleared by the kernel. When we are handling the termination of a child and signal SIGCHLD is blocked, all SIGCHLD signals that are sent by the kernel because other child processes died, will be masked as if only one user process died.

In the wisc version, function *reaper*, which is called whenever event **child_exit** occurs, will use the *wait* system call to obtain the information about which child died, why it died, and how many resources it consumed. Simply changing the *reaper* is not enough, as we need to follow the above described path to handle the termination of a child. Therefore, we have added states **check_all** and **check_next** and events **FOUND_ONE** and **FOUND_NO_MORE**. The death of a child that occurred during state **supervise**, causes a state transition to **check_all** next to **proc_exit** back to **check_all**. Figure 6.2 shows where these new states fit in.

Also, the *reaper* is changed so that it will perform a non-blocking *wait* system call to see whether a child has died or not. If another child did die when we handled the termination

[1]Checkpointing is not considered.

of the first, we will handle its termination as well, after that we have looped back to state **check_next** again. We will continue to loop this way until the reaper finds no more dead child processes. Then *check_all* generates event **found_no_more**, which moves back to state **supervise**. With these extra states and events, it doesn't matter how soon after one another the children die; eventually, they will be reaped.

Finally, the wisc version passed each user process the names of the standard files as arguments. As proposed in the previous chapter, user processes in a pipe job are started without these names as argument.

### 6.2.3 The Condor Library

The changes to the library were straight forward and no additional data structures were necessary. In MAIN, we will call for each standard file the pseudo system call **PSEUDO_open_std_file** which has the next syntax:

```
int rval=open_std_file(int which, char *pathname, int *fd)

which     : 0->stdin, 1->stdout, 2->stderr
pathname  : name of the file
fd        : file descriptor of inherited object (pipe/socket)

rval>0    : std file is already open at file descriptor (*fd)
otherwise : std file is a file, open it with name "pathname"
```

We shall not discuss here the system calls we changed, as there are some that were changed at the Shadow side. So we shall discuss them all together in the next section.

The only thing that needs to be explained is how we force the termination of a process that issues an illegal system call on a pipe (for instance, *lseek*). Note that "normal" illegal system calls (such as, *fork* and *exec*) are detected by the Shadow Child. These system calls are performed remote, as usual, and the Shadow has them marked as illegal. The system calls we have introduced as being illegal, are not illegal in all circumstances; only when they are applied to a pipe. Therefore, we cannot transfer the system call back to the Shadow, as the Shadow cannot see whether or not a pipe is involved. Therefore, we will handle such illegal calls in the Condor Library by sending ourselves a SIGILL signal. By default, this signal causes the termination of the process and results in a core dump. The Starter will detect that this is an abnormal death, reports it to the Shadow which will terminate the job. Furthermore, the Starter moves the core file back to the submitting machine. With the use of a debugger, the user may examine the core file and looking at the stack frame will tell which illegal system call was attempting to manipulate which pipe.

### 6.2.4 System Calls

We shall discuss here what happened with the list of affected system calls.

The system calls on *file descriptors* are adjusted as proposed. For *fcntl*, we only allow the *F_DUPFD* operation to duplicate the file descriptor, all other operations are illegal.

For the system calls on *pathnames*, we changed *open* and *truncate*. The reason we did not touched the other calls is that these calls are not very likely to be performed on normal in- and output files. We prefer simplicity over generality.

## 6.3 Miscellaneous

There are a few details about the implementation of the pipe version that are worth mentioning. They are not specific for pipe jobs, but are a side effect of the project.

### 6.3.1 JobQueue

The JobQueue on each machine is implemented with the use of the *dbm* (DataBase Manager) or *ndbm* (which administrates multiple databases) Dbm and ndbm provide for storage of variable sized *records* in a file and for basic database manipulations (such as fetch, store, delete, etc.) on these records using a unique *key*. The maximum size of a record that can be handled by (n)dbm is 1008 bytes. Condor uses it to store the proc structure of jobs. Since the database file may be accessed by more processes at the same time (for example, the Shadow and condor_submit), Condor uses the *flock* system call to accomplish exclusive file access. When one process opens the JobQueue, the database file is locked, so that another attempt to open it will block until the first process closes it. For single-process jobs the limitation of 1008 bytes was not a problem, but for multiple-process jobs the information that needs to be stored in a proc structure will grow as the number of processes increases. Soon enough we found out that a job of 4 processes with a couple of pipes could not be stored because the size of the proc structure exceeded the limit.

After looking around we found that *gdbm* (GNU DataBase Manager) could be used as a substitute for (n)dbm. It does not impose a limit on the size of the records and most importantly, it is compatible with n(dbm), so only replacing the libraries *should* be enough. Of course, things are never as easy as people would like you to believe, so we had to change the following:

- gdbm: remove the LOCK_NB options from the WRITE/READLOCK macros in systems.h (otherwise gdbm will not block when a file is already exclusive open)

- condor:in *job_queue.c*, change (3x) "Q->pagf" in "dbm_pagfno(Q)" (actually, this is redundant because gdbm will lock the file for you)
- condor: change in all Makefiles "DBM= -ldbm" in "DBM= -lgdbm" and add to the LD_FLAGS an additional -L option for the location of gdbm

### 6.3.2 Condor_submit

Except for the changes necessary for the syntax of pipe jobs, we added a new option on the command line. Condor_submit may now be called with:

```
condor_submit [-c] [-q] job-description file
```

The -q (quit) option already existed and prevents condor_submit to print a summary of the job description after submission. We included the [-c] (check only) option to perform only a check on the syntax and not actually submit the job. This extension is useful because:

1. Condor_submit will create initial checkpoint files for each of the executable files, right after interpreting the executable list. For jobs with large executable files this may take some time. Unfortunately, this time is wasted when later a syntax error is detected. The user should then re-submit the job.
2. Condor_submit does not actually parse the job description file to see whether its contents conforms to the syntax. Instead, it will scan for each command for a matching line, for example, pipe specifications will be scanned by looking for a line that contains the token "pipe". This means that "pipe = ...." will be regarded as a match, whereas "ppipe = ..." will not, however, the latter is perfectly legal (it is simply looked over). So, in the last case, instead of giving a warning, condor_submits just concludes that no pipe specifications are given! By running condor_submit with -c, the user may first check the listed summary on any mistakes.

Also, the names of the executable files may now contain a path specification (either relative or absolute). Normally, Condor_submit would search only in the current directory for the executable files.

### 6.3.3 Notifying the User

The user is sent a mail (by the Shadow Parent) whenever his job has terminated, explaining the cause of the termination. We would like to inform the user at what time the job was submitted, at what time it was completed, what the resource usage was etc. Also, when the process dies, we would like to know why it died.

Since we have more processes in the job, a new layout of this mail is required and an example of it is shown below:

```
Your condor job 12.0
    0    phnx
    1    geom
    2    snomux

exited normal.

Submitted at:        Wed Aug  3 17:01:55 1994
Completed at:        Wed Aug  3 20:09:00 1994
Real Time:             0 03:07:05

 --------------------------------------------------------------------
| Process | Remote User Time | Remote System Time | Total Remote Time |
|---------|------------------|--------------------|-------------------|
|       0 |       0 02:13:56 |         0 00:01:05 |        0 02:15:01 |
|       1 |       0 00:35:22 |         0 00:00:31 |        0 00:35:53 |
|       2 |       0 00:02:17 |         0 00:00:30 |        0 00:02:47 |
 --------------------------------------------------------------------

Local User Time:       0 00:00:32
Local System Time:     0 00:01:45
Total Local Time:      0 00:02:17

Leveraging Factor:   76.1
Virtual Image Size:  25104 Kilobytes
```

The first line shows us that the mail is about the job with id 12.0. Next the names of the processes are listed (phnx, geom and snomux), followed by the cause of termination (in this case the job exited normal). If process 1 died because of a segmentation fault, it would have said: "Your condor job....., process 1 died because of signal 11".

Next, the time of submission and completion are listed. The subtraction of these two is the real time (also called *turn around time*). Then, for each process, the remote user time, the remote system time and the total of these two are listed. Because the Shadow has contributed to the jobs execution, its user and system time and the total of these two are printed as well. Finally, the leveraging factor, that is the ratio of remote CPU time to the local CPU time, and the virtual image size (all three processes together) are listed.

# Chapter 7

# Tests and Results

To test the implementation of our pipe version of Condor, we have run three different types of jobs: jobs that consist of programs we wrote ourselves, jobs that consist of programs that are used by the SMC group and jobs that consist of the program GREP. These jobs are described in more detail in section 7.1.

We are of course interested in the *performance* of our pipe version of Condor. We shall compare pipe jobs that run with and without Condor (this is the traditional way for the SMC jobs to run). We also run the individual processes sequentially, that is, without pipes. Because we need a job that will run for quite some time, we used an SMC job. The results are presented in section 7.2.

In section 7.3 we give some numbers on Condor's overhead in hosting a job.

Furthermore, we are interested in the *workload* of pipe jobs, defined as: the number of processes in the run queue[1] averaged over 1 minute. For normal single-process jobs, Condor will suspend the job's execution when it detects that the workload exceeds some number in the configuration file; this probably means that the user has returned to his machine. If after a period of time (may also be configured), the workload is still larger than the limit, Condor will vacate the machine by killing the job, otherwise the job may continue. Due to multiple-process jobs and pipe jobs, the detection of runnable non-Condor processes (the owner's processes) is not that easy. The workload of the Condor job may fluctuate depending on the phase the job is in. In section 7.4 we therefore examine the workload of the SMC job.

[1]This is the run queue of the UNIX kernel.

## 7.1 Jobs tested under the Pipe Version of Condor

The following list shows the types of pipe jobs that have run successfully:

1. Jobs with complex pipe topologies. For this purpose we wrote a simple `file copy` program with the following syntax:

   ```
   filecopy n inputfile_1 inputfile_2 ... inputfile_n outputfile_1
   outputfile_2 ... outputfile_n
   ```

   This program copies `input_file_x` to `output_file_x`. Figures 7.1, 7.2 and 7.3 show some of the pipe topologies tested. Below, the relevant parts of job description files of these jobs are listed:

   ```
   ###########################
   #  Job description file: figure 7.1

   Executable   = p1 p2 p3
   Pipe         = p1 > output1 input1 > p2, p2 > output1 input1 > p3,
                  p1 > output2 input2 > p2, p2 > output2 input2 > p3,
                  p1 > output3 input3 > p2, p2 > output3 input3 > p3
   ###########################

   ###########################
   #  Job description file: figure 7.2

   Executable   = p1 p2 p3 p4 p5
   Pipe         = p1 > output1 input1 > p2, p1 > output2 input2 > p2,
                  p1 > output3 input1 > p3, p1 > output4 input2 > p3,
                  p1 > output5 input1 > p4, p1 > output6 input2 > p4,
                  p2 > output1 input1 > p5, p2 > output2 input2 > p5,
                  p3 > output1 input3 > p5, p3 > output2 input4 > p5,
                  p4 > output1 input5 > p5, p4 > output2 input6 > p5
   ###########################

   ###########################
   #  Job description file: figure 7.3

   Executable   = p1 p2 p3
   Pipe         = p1 > output1 input1 > p2, p2 > output1 input1 > p3,
                  p1 > output2 input2 > p2, p2 > output2 input2 > p3,
                  p3 > output1 input2 > p1
   ###########################
   ```

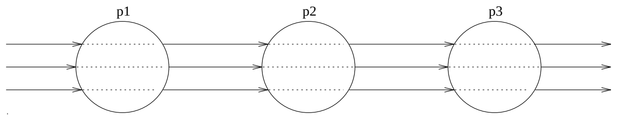

Figure 7.1: Job with filecopy processes.

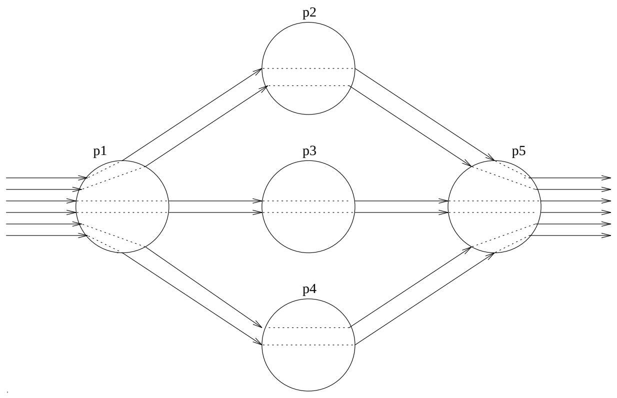

Figure 7.2: Job with filecopy processes.

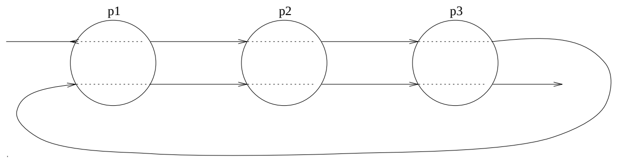

Figure 7.3: Job with filecopy processes.

Note that Condor requires that the names of the executable files are unique, therefore `p1`, `p2`, `p3`, `p4` and `p5` in the above examples are all symbolic links to the executable file `filecopy`.

In the first two jobs, the first process reads from files and the last process writes to files. The input and output files of the processes inbetween are mapped to pipes. The last job has a circular pipe connection. Process p1 only reads one file, pipes the data to process p2 which pipes them to process p3 which pipes them back to process 1 (because of the pipe command: `p3 > output1 input2 > p1`). From there, data are piped to process p2, next to process p3 which writes it to a file. Note that this job may deadlock if the input file is too large. What then happens, is that before process p1 starts reading from its pipe, process 3 is blocked because the pipe is full. For a small file, process p1 is ready copying file 1 and starts reading the second before the pipe with process p3 gets full.

2. To test real-life jobs, we have linked the programs in an SMC job with the Condor library. This job consists of three processes, these are `phoenix`, `geometry` and `snomux`, which run in a pipeline. `Phoenix` will start reading data from a file or a tape, next, its output is piped to `geometry`, next to `snomux`, which will create an output file. The relevant part of job description file is listed below:

```
###########################
#  Job description file: SMC job

Executable   = phnx geom snomux
Pipe         = phnx > fort.10 fort.11 > geom,
               geom > fort.10 fort.11 > snomux

###########################
```

3. Because the use of filters is a common and powerful tool in pipe jobs, we have built GNU's[2] GREP utility to run with Condor. Such programs are usually run as the first or last stage of a pipe job. For example, GREP may be used to filter relevant data from the input before processing it. Also, compression programs may be used as a last stage to compress (large) output files. The steps that were needed to install GREP for Condor are listed in Appendix C.

## 7.2 Performance of Pipe Jobs

To test the performance of our pipe version of Condor, we run the SMC job in the following way:

[2]GNU stands for GNU is Not UNIX.

1. as a pipe job *with* Condor
2. as a pipe job *without* Condor (using named pipes)
3. as two jobs, one pipe job with two processes and one job with the third process; all running with Condor
4. as three jobs, one for each process, all run sequentially with Condor

Comparing the first and the second situation, will show Condor's overhead in running pipes. The first and the third/fourth situation will show the advantage of pipe jobs compared to running the individual processes sequentially.

Before presenting the results, we will describe the SMC job and the test environment in more detail. `Phoenix` (p1) will read from a magnetic tape that is physically mounted on the same machine where the job will run (a Sun SparcStation 10) and produces about 130MB of output. `Geometry` (p2) will read in this output and will produce output of about 160MB and finally, `Snomux` (p3) will will read in `Geometry`'s output and will create an outputfile of about 10MB. Only when these processes are run sequentially, output is written to files, when they are run in a pipe job only the last output file is created, the other output is redirected to the pipes.

Because of the size of the output files, we had to use a file system that was mounted at our test machine, so the data are not sent over the network. Also, because the average time of running one job was more than three hours, we have run each job only twice. We have run these jobs at weekends only when nobody was using the machine. Because it is not our intention to test the performance of remote system calls, we have run the job on the initiating machine, as a consequence, the Shadow runs on the same machine as the pipe job. Finally, because the logging facility of Condor decreases the performance, we turned off all logging.

Table 7.1 shows the results of the execution time of the different jobs. We use the piping symbol (|) to indicate that processes are connected via a pipe, whereas a comma (,) is used to denote that the process(es) run sequentially. For each job, the CPU time for each process is listed. Because the Shadow is part of a Condor job, its CPU time is listed also. Next, the *Total* time is listed, which is the summation of the CPU time of the individual processes including the Shadow, followed by the turn-around time (*Turn*). *Util.* shows the system utilisation (percentage); it is obtained by dividing *Total* by Turn, multiplied by 100. Note that there were in fact 6 only different condor jobs: 1 with three processes (p1 | p2 | p3), 2 with two processes ( p1 | p2 and p2 | p3) and 3 with one process (p1, p2 and p3 individually). We listed *all* possible combination of Condor jobs and calculated the times of the Shadow, Total and Turn from the individual jobs.

From this table it is clear that there is only little performance decrease in running the job with Condor in comparison to running the job without Condor: 173:35 CPU time against 171:07 CPU time. Also, when comparing a process that runs in a pipe job to a process that

| Type | P1 | P2 | P3 | Shadow | Total | Turn | Util. (%) |
|---|---|---|---|---|---|---|---|
| Condor: 3 processes | | | | | | | |
| p1 \| p2 \| p3 | 134:42 | 35:49 | 2:47 | 2:17 | 173:35 | 185:40 | 93 |
| p1 \| p2, p3 | 134:11 | 36:30 | 3:43 | 7:32 | 182:06 | 191:53 | 95 |
| p1, p2 \| p3 | 135:13 | 36:37 | 2:45 | 6:30 | 181:05 | 193:10 | 94 |
| p1, p2, p3 | 135:13 | 37:27 | 3:43 | 11:37 | 188:00 | 201:17 | 93 |
| Condor: 2 processes | | | | | | | |
| p1 \| p2 | 134:11 | 36:30 | - | 4:15 | 176:50 | 183:36 | 96 |
| p2 \| p3 | - | 36:37 | 2:45 | 2:45 | 42:07 | 45:20 | 93 |
| p1, p2 | 135:13 | 37:27 | - | 8:29 | 181:09 | 193:00 | 93 |
| p2, p3 | - | 37:27 | 3:43 | 7:52 | 49:02 | 53:27 | 92 |
| Condor: 1 process | | | | | | | |
| p1 | 135:13 | - | - | 3:44 | 138:57 | 147:50 | 94 |
| p2 | - | 37:27 | - | 4:46 | 42:03 | 45:10 | 93 |
| p3 | - | - | 3:43 | 3:07 | 6:50 | 8:17 | 82 |
| Without Condor | | | | | | | |
| p1 \| p2 \| p3 | 133:15 | 35:27 | 2:25 | - | 171:07 | 178:12 | 96 |

**Table 7.1** Execution times of jobs with and without Condor (in minutes:seconds).

runs stand-alone, we see that the stand-alone job needs more time to finish. Finally, note that the Shadow consumes more CPU time when processes run separately.

## 7.3 Overhead of Condor

To get a better idea of Condor's overhead in starting a job and handling its termination, we have run jobs that consist of *empty* processes, which do nothing and terminate immediately. The steps taken by Condor that are included in the overhead are:

- parsing the job description file
- creating initial checkpoint files from the original executable files
- storing the job's proc structure in the JobQueue
- negotiating with the Central Manager by the Schedd
- starting the Shadow and the Starter
- reading in the proc structure from the JobQueue

- sending the proc structure and the checkpoint files to the Starter
- starting the user processes
- detecting the death of a child and sending back information to the Shadow
- deleting the proc structure from the JobQueue after all processes have terminated

Condor's overhead is in fact the turn-around time of these jobs. The turn-around time is measured as the difference of two time stamps: the first is made by condor_submit immediately when it starts up, the second is made by the Shadow Parent, just before it mails the user.

The only parameter that may influence this time is the size of the executable file. Therefore, we ran 100 jobs: the first 10 consist of one empty process, the next 10 of two, up until jobs of ten empty processes. Since the turn-around times may largely differ between jobs with the same number of processes, we list not only the average time, but also the best and worst times. The results are listed in Table 7.2; in Figure 7.4 the relationship between the number of processes and the turn-around time is shown. The exact image size of the *empty* program is 557056 bytes.

| # of processes | best | worst | | mean |
|---|---|---|---|---|
| 1 | 0:17 | 0:44 | | 0:29 |
| 2 | 0:32 | 0:45 | | 0:34 |
| 3 | 0:27 | 0:41 | | 0:35 |
| 4 | 0:39 | 1:12 | | 0:47 |
| 5 | 0:44 | 1:29 | | 1:00 |
| 6 | 0:49 | 1:22 | | 1:11 |
| 7 | 0:59 | 1:34 | | 1:19 |
| 8 | 1:04 | 2:21 | | 1:26 |
| 9 | 1:17 | 2:28 | | 1:36 |
| 10 | 1:47 | 2:05 | | 1:54 |

**Table 7.2** Turn-around time of jobs with empty processes (minutes:seconds).

## 7.4 Workload of Pipe Jobs

Condor uses the workload for the detection of the presence of non-Condor processes. For single-process jobs, when Condor detects an higher workload than a certain limit, it concludes that there are non-Condor processes running and thus migrates a Condor job.

For multiple-process jobs, we cannot use the same technique for the detection of non-Condor processes. For a single-process job, we just assume that the workload of such a job will never exceed a certain number. We cannot impose such a limit on multiple-process jobs, as the

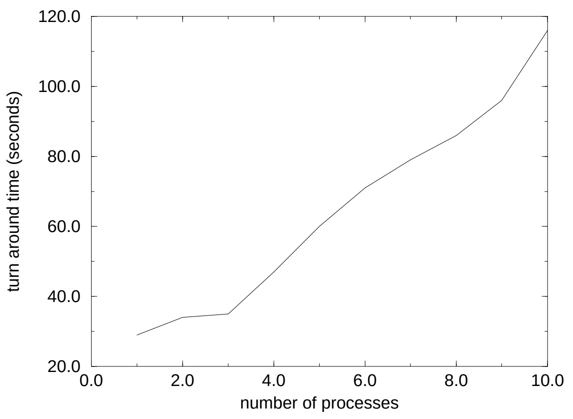

Figure 7.4: Relationship between number of processes and the turn-around time (seconds).

workload may fluctuate during the run and depends on the number of processes. Therefore, from the workload only, Condor cannot distinct the following two items:

1. Nearly all Condor processes are sleeping and there are running non-Condor processes;
2. all Condor processes are running and there are no non-Condor processes running.

In the first case, the Condor job should probably be migrated if these non-Condor processes will continue running. In the second case, there are no non-Condor processes, so nothing needs to be done.

To obtain information about the workload, we run a small script in the background when running a pipe job, which will run every minute the UNIX program `w`. This program will list, among other things, the workload. Figure 7.5 shows the workload of the SMC job with all processes connected with pipes (p1 | p2 | p3). Just like the other pipe jobs, we have run the SMC job on the submitting machine.

The outlier of the workload at the end of the job is caused by the Starter and the Shadow Parent. Normally, these two processes are sleeping, but when a user process dies, they wake up. Also, the Shadow starts the program `mail` to send the user mail when the job has terminated.

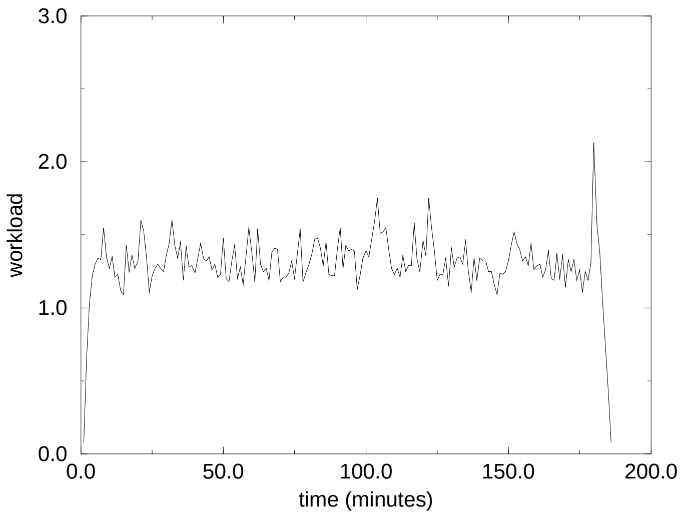

Figure 7.5: Workload of an SMC job running with Condor.

# Chapter 8

# Conclusions

We now present the conclusions related to our design and implementation of pipes in Condor and to our design of a checkpointing mechanism for pipe jobs.

## Running Pipe Jobs

We successfully implemented a piping facility in the Condor wisc version. We provided for both a shell-alike piping mechanism and for a file-to-pipe mapping, which allows jobs in practical use to run without changing a single line of source code. We imposed only a few restrictions on the pipe topology. We successfully built the GNU's GREP utility which may now run with Condor. A whole range of filters, compression programs and other utilities, now becomes available for running in pipe jobs with Condor.

From the tests, we conclude that there is no significant decrease in performance when running pipe jobs with Condor. Running the job as a whole with pipes does result in a slightly shorter turn-around time than running the processes sequentially. This does not show for long-running jobs as they spend much more time consuming CPU power than performing I/O operations; this even holds for jobs with large input and output files as was shown with the SMC job. However, a clear advantage of pipe jobs is that there is no need to store intermediate files, which keeps the file system from filling up. We did not consider the impact on the network, but it seems clear that piping data from one process to another in the job instead of redirecting these data back to the submitting machine, results in less network traffic.

The issue of detecting the presence of non-Condor processes still needs to be solved. The fluctuation of the workload of multiple-process jobs and pipe jobs makes it very difficult to impose a limit on the workload for which such jobs may run. Therefore, Condor needs a new mechanism to detect the presence of running non-Condor processes. To do this, it probably needs to read kernel information about running processes. Unfortunately, this is platform dependent.

### Checkpointing Pipe Jobs

We have presented the design for a checkpoint/rollback mechanism for pipe jobs which run as a whole on one machine. Our checkpoint mechanism does not rely on any order of checkpointing user processes. We anticipated the new checkpoint mechanism, which is in the process of being implemented in Wisconsin, for which a user process need not die for checkpointing. We expect the mechanism that we designed will work fine with this new checkpointing mechanism.

In the future, a checkpoint/rollback mechanism for pipe jobs which are distributed among several machines is designed. We suggest that a distinction is made between checkpointing for *process migration* and checkpointing for *process recovery*. For the first, only the processes that need to migrate need to be checkpointed and these do not have to roll back (only restarted). For the latter, probably more than one process needs to be checkpointed depending on the message exchanges and processes may have to roll back. The reason for this distinction is that rolling back is more expsensive, in both real time and CPU time, than restarting a job. Also, process migration occurs more often than process recovery. It is therefore better to use a checkpoint/restart mechanism for process migration and a checkpoint/rollback mechanism for process recovery.

Note that before we can checkpoint a distributed pipe job, there should be a mechanism in Condor to distribute a job. Currently, a Schedd is returned a name of *one* machine to run the job on. Distributing a job requires a different policy for scheduling jobs among available machine. Also, a mechanism should be present for the Shadow to host such a job. First of all, however, we should study what we gain from distributing a pipe job. It is not that obvious that running a pipe job of say three processes will run three times as fast when running it distributed on three different machine as to running it centralised on one machine.

### Possible Future Extensions to Condor

Finally, we have some hints on future extensions to the Condor system in general:

- Condor_submit should use a different parsing technique to detect badly formatted job description files. This could be implemented with the use of Lex (LEXical analyser) and Yacc (Yet Another Compiler Compiler).
- The syntax itself could be changed to provide for a shell-alike syntax. The user may then specify something like: `Executable= p1 | p2 | p3 > output`. This is much clearer for simple jobs that consist of a single pipeline.

- The Starter should be changed to conform to the philosophy of Condor, that is, the Starter should not request a proc structure for the per-process information but should only request the relevant data. This could be done by means of one or more new pseudo calls.
- The Starter should also be changed to take full advantage of C++: other types of jobs should be implemented with new classes derived from class UserProc.
- Other utilities, such as compression programs, could be built for Condor to provide for more powerful jobs.

# Bibliography

[1] M.J. Bach, *The design of the* UNIX *operating system*, Prentice Hall, 1986.

[2] A. Bricker, M.J. Litzkow and M. Livny, "Condor technical summary," Version 4.1b, University of Wisconsin - Madison, 1991.

[3] A. Bricker and M.J. Litzkow, UNIX manual pages: condor_intro(1), condor(1), condor_q(1), condor_rm(1), condor_status(1), condor_summary(1), condor_config(5), condor_control(8), condor_master(8), Version 4.1b, University of Wisconsin - Madison, 1991.

[4] X. Evers, *A literature study on scheduling in distributed systems*, TU-Delft, October 1992.

[5] A. Goscinski, *Distributed Operating Systems - The logical design*, Addison-Wesley publishing company, 1991.

[6] C. Hunt, *TCP/IP network administration*, Nutshell Series, O'Reilly & Associates, Inc., 1992.

[7] B.W. Kernighan and D.M. Ritchie, *The C programming language*, second edition, Prentice Hall, 1988.

[8] L. Lamport, LaTeX*: A document preparation system*, Addison-Wesley publishing company, 1986.

[9] L. Lamport, LaTeX*local guide for Nikhef*, revised by Marcel Prins and Gertjan Stil, 1987.

[10] S. Leffler, M. McKusick, M. Karels and J. Quarterman, *The design and implementation of the BSD4.3* UNIX *operating system*, Addison-Wesley, 1989.

[11] M.J. Litzkow, "Remote UNIX, turning idle workstations into cycle servers," in *Proceedings of the 1987 Summer Usenix Conference*, Phoenix, Arizona, 1987.

[12] M.J. Litzkow, M. Livny and M.W. Mutka, "Condor - A hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California, 1988, pp. 104—111.

[13] M.J. Litzkow and M. Livny, "Experience with the CONDOR distributed batch system," *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, 1990.

[14] M.J. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel," *Usenix Winter Conference*, San Francisco, California, 1992.

[15] M.W. Mutka and M. Livny, "Profiling workstations' available capacity for remote execution," in *Proceedings of Performance '87, The 12th IFIP W.G. 7.3 International Symposium on Computer Performance Modeling, Measurement and Evaluation*, Brussels, Belgium, 1987, pp. 529—544.

[16] P. van Sebille, *Checkpointing in distributed systems*, TU-Delft, Februari 1994.

[17] S. Talbott, *Managing projects with make*, Nutshell Series, O'Reilly & Associates, Inc., 1986.

# Appendix A

# Process Structures

```
typedef struct {
                /* job specific data */
     int             version_num;     /* version of this structure */
     PROC_ID         id;              /* job id (cluster and proc) */
     char            *owner;          /* login of person submitting job */
     int             q_date;          /* UNIX time job was submitted */
     int             completion_date; /* UNIX time job was completed */
     int             status;          /* Running, unexpanded, completed, .. */
     int             prio;            /* Job priority */
     int             notification;    /* Notification options */
     int             image_size;      /* Size of the virtual image in K */
     char            *env;            /* environment */
     char            *rootdir;        /* Root directory for chroot() */
     char            *iwd;            /* Initial working directory   */
     char            *requirements;   /* job requirements */
     char            *preferences;    /* job preferences */
     struct rusage   local_usage;     /* accumulated usage by shadows */

                /* process specific data */
     char            *cmd;            /* a.out file */
     char            *args;           /* command line args */
     char            *in;             /* file for stdin */
     char            *out;            /* file for stdout */
     char            *err;            /* file for stderr */
     struct rusage   remote_usage;    /* accumulated usage on remote hosts */
} V2_PROC;


typedef struct {

                /* job specific data */
```

```
        int             version_num;     /* version of this structure */
        PROC_ID         id;              /* job id */
        int             universe;        /* STANDARD, PIPE, LINDA, PVM, etc */
        int             checkpoint;      /* Whether checkpointing is wanted */
        int             remote_syscalls; /* Whether to provide remote syscalls */
        char            *owner;          /* login of person submitting job */
        int             q_date;          /* UNIX time job was submitted */
        int             completion_date; /* UNIX time job was completed */
        int             status;          /* Running, unexpanded, completed, .. */
        int             prio;            /* Job priority */
        int             notification;    /* Notification options */
        int             image_size;      /* Size of the virtual image in K */
        char            *env;            /* environment */
        char            *rootdir;        /* Root directory for chroot() */
        char            *iwd;            /* Initial working directory   */
        char            *requirements;   /* job requirements */
        char            *preferences;    /* job preferences */
        struct rusage   local_usage;     /* accumulated usage by shadows */

                   /* process specific data */
        int             n_cmds;          /* Number of executable files */
        char            **cmd;           /* Names of executable files */
        char            **args;          /* command line args */
        char            **in;            /* file for stdin */
        char            **out;           /* file for stdout */
        char            **err;           /* file for stderr */
        struct rusage   *remote_usage;   /* accumulated usage on remote hosts */
        int             *exit_status;    /* final exit status */

                   /* pipe specific data */
        int             n_pipes;         /* Number of pipes */
        P_DESC          *pipe;           /* Descriptions of pipes */

                   /* PVM specific data */
        int             min_needed;      /* for PVM jobs */
        int             max_needed;      /* for PVM jobs */

        char            pad[50];         /* make at least as big as V2 proc */
   } V3_PROC;
```

# Appendix B

# State Transition Diagram of the Starter

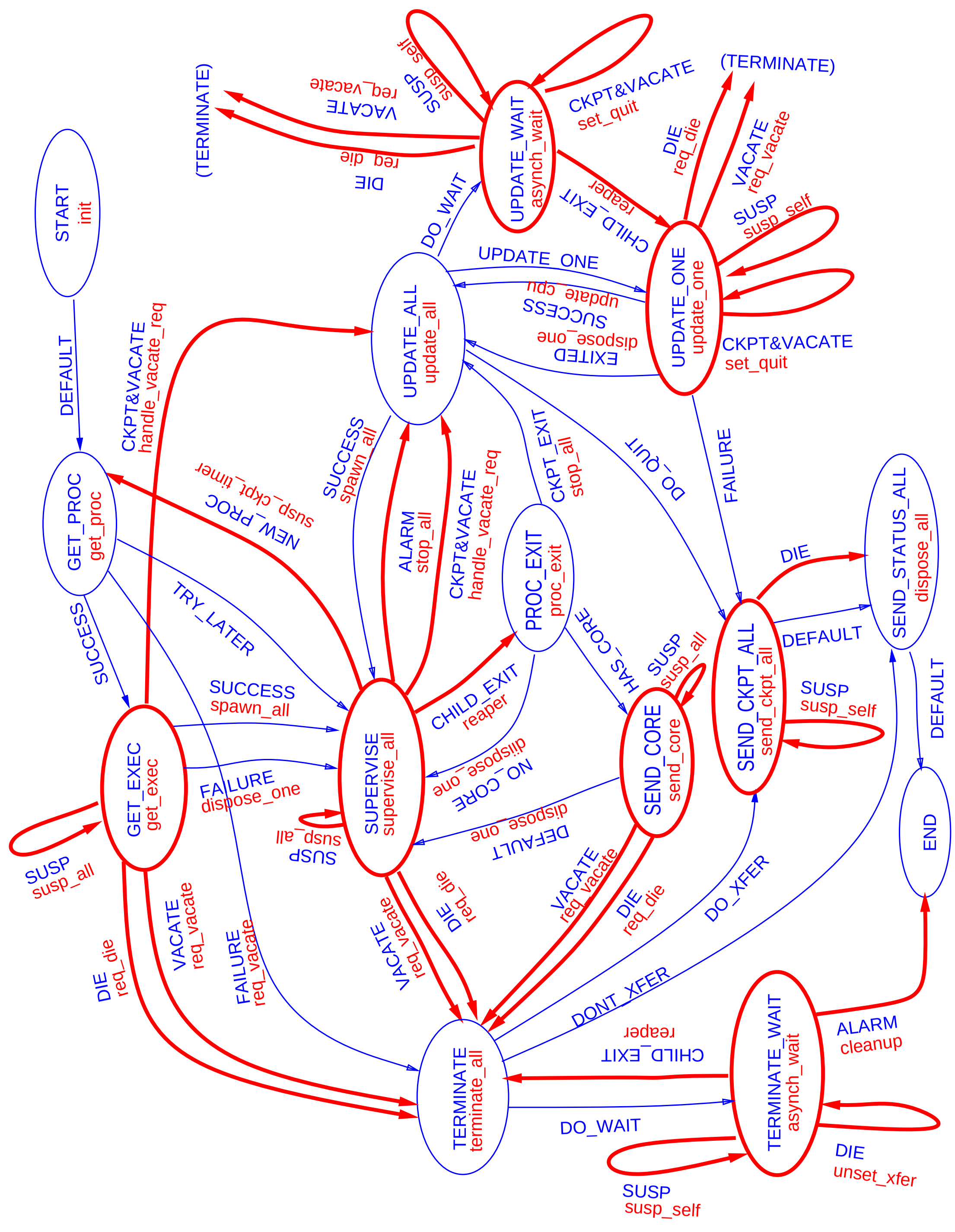

Figure B.1: The original State Transition Diagram of the Starter.

KEY

| | |
|---|---|
| ◯ | State not requiring handling of asynchronous events |
| ⭘ | State requiring handling of asynchronous events |
| → | Synchronous Event |
| ➡ | Asynchronous Event |
| (name) | Name of state |
| (name) | Name of action routine |

ASYNCHRONOUS EVENTS

| | |
|---|---|
| VACATE | If user jobs running, terminate them. Then if have checkpoint files, send them back. |
| CKPT&VACATE | If user jobs running, checkpoint them. Then if have checkpoint files, send them back. |
| DIE | If user jobs running, terminate them. Then don't send back any checkpoint files. |
| SUSPEND | If user jobs running, suspend them. Then wait for a CONTINUE signal. |
| CONTINUE | Resume normal operations after a suspension. |
| ALARM | In SUPERVISE state, tell user job to checkpoint. In VACATE state, kill user jobs forcibly (−9). |
| CHILD_EXIT | User process exited. |

FLAGS

| | |
|---|---|
| QUIT | Leave after completing and transferring checkpoints, initially false |
| XFER | Transfer checkpoint files before terminating, initially true |

Figure B.2: Notes on the State Transition Diagram.

# Appendix C

# Running GNU's GREP Utility

Building an application suitable for running with Condor is not always an easy task, so it is illustrative to show what needs to be done for building GNU's GREP utility. This is a good example, because it uses the *mmap* system call, which is not supported in any Condor version. Furthermore, I compiled it with GNU's gcc which gives some trouble when linking with the Condor library, which is compiled with Sun's cc. I will list all steps that were necessary to install GREP for Condor; this may help you with installing your own applications.

First get the GREP distribution from your favorite ftp-site and install it as normal. Since the Condor library is compiled with Sun's cc, it is best to compile other applications with cc as well. I couldn't compile GREP with our (older) version of cc, so I used GNU's gcc (2.5.7) instead. Installing GREP for normal use is no big deal, just run "configure", then "make".

To link with the Condor library, we should use a command like this[1]:

```
ld -o a.out -dc -e start -Bstatic /your_path/condor\rt0.o a.o b.o
/your_path/libcondor.a
```

Normally, this will do for most of your applications. However, since we compiled GREP's source with gcc, we get the next message form the linker.

```
/bin/ld   -dc -dp -e start -Bstatic  -o grep.condor condor_rt0.o
grep.o getopt.o regex.o dfa.o kwset.o obstack.o search.o libcondor.a

ld: Undefined symbol
___main
___eprintf
```

[1]Consult the man pages on condor_submit.

The undefined symbols are a consequence of compiling with gcc and linking with the Condor library which is compiled with cc. What we should do is extract these symbols from GNU's standard C library[2]. Instead of making one link command, we shall use *incremental linking*. We extract these symbols with the following command:

```
/bin/ld  -r -Bstatic -o undefined.o -u ___main -u ___eprintf
/your_path/libgcc.a
```

Note that we did not specify any object files to link. We only `undefined` both `___main` and `___eprintf`; as a result these two symbols are extracted from library `libgcc.a`. Now, we may extend our original link command with this extra object file. The following script shows the complete actions involved:

```
CONDOR=/global/xwindow/condor_b/lib
GCCLIB=/global/xwindow/gcc-2.5.7/lib/gcc-lib/sparc-sun-sunos4.1/2.5.7/libgcc.a
UNDEFINES="-u ___main -u ___eprintf"

/bin/ld  -r -Bstatic -o undefined.o $UNDEFINES $GCCLIB

/bin/ld   -dc -dp -e start -Bstatic  -o grep.condor $CONDOR/condor_rt0.o
grep.o getopt.o regex.o dfa.o kwset.o obstack.o search.o undefined.o
$CONDOR/libcondor.a
```

We have now an executable that is, in principle, suitable for running with Condor. Unfortunately, the program will dump core once in a while. We are mailed by Condor that the program was killed by signal 11 (SIGSEGV). Examining the stack frame stored in the core file shows the following[3]:

```
Program terminated with signal 11, Segmentation fault.
#0  0x15b8c in _doprnt ()
(gdb) bt
#0  0x15b8c in _doprnt ()
#1  0x124a0 in fprintf ()
#2  0x2718 in fillbuf ()
#3  0x2d48 in grep ()
#4  0x3864 in main ()
#5  0x25abc in MAIN (argc=3, argv=0xf7ffffa8, envp=0xf7ffffb8)
    at ckpt_main.c:260
```

It shows that we died in `_doprntf`, a function of the standard C library. The program died because of a segmentation fault, which normally indicates that the program is buggy.

[2] I admitt that this is tricky but it seems to work!

[3] Fragment taken from gdb (GNU debugger).

However, this is very unlikely and therefore we will use the log files of Condor to figure out what went wrong. We learn from the Shadow's log that *mmap* is used by GREP, which is not supported by Condor. It is not illegal however, Condor just ignores it. Looking at GREP's Makefile shows us that GREP can be built without using *mmap*; just remove the `-DHAVE_WORKING_MMAP=1` option from the Makefile. Build GREP again and re-link it with the Condor library. GREP will work fine now.