Condor Porting Guide

Miron Livny, Jim Pruyne, Dhaval N. Shah & Jim Basney
condor-admin@cs.wisc.edu

June 6, 1996

1 Introduction

The aim of this guide is to aid the user who is porting Condor to a new platform. The reader is encouraged to go through the Condor Technical Summary before perusing this guide. The items contained in this document address issues that we, as the Condor team members, have come across a number of times while porting Condor to various platforms. Certainly all possible issues will not be covered here simply because we haven't encountered also possible environments, and because an exhaustive list of every possible source of trouble is impossible.

The porting philosophy of Condor has been to keep the code as compliant to POSIX.1 as possible. It appears that most vendors support this standard. In some cases POSIX.1 is not sufficient because it does not provide a complete programming interface. In these cases, it has been our goal to provide the necessary functionality in a Condor utility library which provides the proper implementation for each platform.

2 Condor Components

The Condor source distribution consists of the following components :

1. A config directory for Imake
2. Utility libraries and include directories
3. The individual daemons.
4. Checkpointing library
5. Scripts

It is recommended that a port to a new platform proceed in the order given above.

3 Three Phase Condor Porting

Condor supports a number of features such as remote system calls and checkpointing which are not needed in all environments, and in any event are often among the most difficult to port. For this reason, Condor can be ported incrementally with some features left out until they are desired. The three modes in which Condor can operate are:

Clipped Wing Condor : Clipped Condor supports remote execution of jobs, but does not support either checkpointing or remote system calls. Only "vanilla" type Condor jobs may be run in a clipped environment. Clipped Condor will be generated if the platform.cf file in the `config` directory has the macros `DOES_CHECKPOINTING` and `DOES_REMOTE_SYSCALLS` set to `NO`. The files in the `config` directory are described in more detail in the next section.

One Wing Condor : One Wing Condor adds the ability to perform remote system calls. This is not generally a useful state for running the system, but can be useful during porting and debugging. To build a One Wing system, set `DOES_REMOTE_SYSCALLS` to `YES`.

Full Condor : This is the final stage of Condor supporting both checkpointing and remote system calls. It may not be suprising to learn that you get to this state by setting `DOES_CHECKPOINTING` to `YES`.

4 Using Imake

In an effort to encapsulate differences such as compiler names, options, and system particulars, condor uses `imake`. Imake essentially runs the C preprocessor, with a standard set of rules, to generate a `Makefile` from an `Imakefile`. Condor assumes that the preprocessor which will be executed conforms to the ANSI standard. You can override the name of the C preprocessor to be run by setting the environment variable `IMAKECPP`. Imake must also be given the name of the directory from which to take its standard rules. For Condor these rules reside in the `config` directory which is described in the next section. To tell imake where to find the rules directory, use the command line switch `-I` similar to you would with most C compilers. For example, you would run "`imake -I../../config`" to generate a `Makefile` in a directory which is two levels below the `config` directory. It will always be necessary to run `imake` prior to running `make` in any of the Condor source directories.

5 Configuration files

At the top level of the Condor source distribution are the `src` and `config` directories. Before proceeding to the source, it is necessary to specify the configuration of your platform. The first step is to provide `imake` a way to identify the platform you are running on. This is done in the `Platform.select` file. In this file, conditions are written which try to detect what platform is being used based on symbols which are pre-defined by the C preprocessor. To determine what symbols are pre-defined on your platform, consult your C compiler manual.

In modifying the `Platform.select` it will be necessary to decide upon a platform name. This is commonly of the form `CPU_OS.cf` (e.g. `ALPHA_OSF1.cf`). Next, it is necessary to write a platform specific configuration file. The best method is simply to start by copying an existing configuration file, hopefully one which is as similar to the target platform as possible. The configuration settings are mostly evident from their names. If nothing else is set, make sure that the name of your C and C++ compilers are given. These may be specific to your installation rather than your platform.

6 The Header and Utility Library Directories

Condor contains two directories dedicated to header files, and three utility libraries. Ports of each of the utility libraries must be completed prior to proceeding to any of the other Condor code because all other components rely on these libraries. The goal of these header files and libraries is to abstract away all platform dependencies so that the rest of the Condor system sees a consistent programming environment.

6.1 Headers

The header files for Condor reside in the `h` and `condor_includes` directories. The goal of these headers is to provide definitions of Condor's internal data structures as well as to hide platform specific header file differences. Examples of the differences between platforms are dependencies on the order of including system header files, or which functions and constants are defined in which header. As a rule, most of your porting activity should take place in these directories. If you ever find yourself doing some platform checking in a source file, stop and ask yourself if you could have accomplished the same thing in an existing header file. In general, by putting platform specific changes in the headers, you avoid replicating these changes in numerous source files. Pay particular attention to the files in the `condor_includes` directory which contain the word `fix` in their name. These were specifically intended to account for differences in a specific system header file. In may be worth going through these files at least once before trying to compile any of the source. In all likelihood, you'll end up making a number of changes to these files.

6.2 Utility Libraries

The Condor utility libraries reside in the directories `condor_util`, `condor_c++_util` and `condor_xdr_lib`. With a proper set of header files, most of these libraries port relatively easily, particularly the C++ and XDR libraries.

The `condor_util` contains the most platform specific functionality. In some cases, functions will be included or not included in the generated library based on values given in your specific `Platform.cf` discussed earlier. If you find conflicts between functions in the utility library and the system library, it is probably a wise idea to modify your Platform.cf to avoid inclusion in the Condor library so that the vendor provided function is used.

Some of the functions in the Condor utility library retrieve values from the kernel or run other utility programs to get information about the machine Condor is running on. These will nearly always require special attention on each platform. Source files which include these calls are: `phys_mem.c`, `virt_mem.c`, `free_fs_blocks.c` and getmnt.c.

7 Daemons

7.1 Overview

For the most part, porting the Condor daemons should be a matter of fixing up function prototypes and header files. This section will point out some particular trouble spots, however, and try to give suggestions.

7.2 StartD

One particularly non-portable aspect of the startd is the load average computation. There are currently two approaches to getting the current load average: (1) using `nlist` to examine the appropriate kernel variable and (2) using `popen` to get the output from `uptime`. It is preferrable that you port both of these approaches so the startd can run efficiently under the root uid (and use the first approach) or under a non-root uid (and use the second approach). The source files of interest all begin with `load_avg`.

The startd is also responsible for tracking tty activity to determine if a machine is idle. You will need to add code in `event.c` to handle this tracking on your system.

7.3 Starter

The starter performs analysis on Condor executables and core files to make sure they appear complete and correct. For Condor executables, the executable should have the correct magic number for the architecture and should be linked with the Condor libraries. Core files should also have the correct magic number. You will need to write your own `analyze_exec.<arch>.C` and `analyze_core.<arch>.C` based on current versions, with the correct magic numbers from your system's header files.

The starter also sets resource limits for Condor jobs (stack size, core size, etc.). You will need to write your own `resource_limits.<arch>.C`, based on current versions.

7.4 SchedD

Porting of the schedd should be straightforward.

7.5 Shadow

The shadow relies on the condor_syscall_lib.a for much of its functionality. Otherwise, porting the shadow code should be straightforward.

7.6 Master

Porting of the master should be straightforward.

7.7 Central Manager

Porting of the central manager should be straightforward.

7.8 KbdD

Porting of the kbdd should be straightforward.

8 Utility Programs

The following are small utility programs that you will need to build:

condor_checkpoint	condor_customize	condor_globalq	condor_history
condor_jobqueue	condor_off	condor_on	condor_preen
condor_prio	condor_q	condor_reconfig_schedd	condor_reschedule
condor_rm	condor_status	condor_summary	condor_update_prio
condor_vacate			

These programs should not require much (if any) porting, but you should build and test them at this point in your Condor port because you will need them to test the daemons and examine the state of your pool.

At this point, you have built all of the daemons and the utility programs. You are ready to get a small test pool running. Make sure the collector is getting good information from the daemons in your pool. Submit vanilla jobs to your pool, and verify that they run on idle machines, that they vacate properly, etc.

9 Checkpointing

9.1 Overview

Porting the checkpointing library (`libckpt.a` in the `condor_ckpt` directory) is the last step in a Condor port, after successfully porting Clipped Condor. This library contains the functionality to checkpoint and restart Condor jobs. There are two main issues in porting the checkpointing library: (1) whether to port the support for dynamic linking and (2) how to find the bounds of your process's address space.

9.2 Linking on your System

As a rule, we do not plan to port dynamic linking support to platforms for which it is not absolutely necessary. Your first task is to determine whether this will be necessary on your system. Some questions to ask are:

- Can you statically link trivial programs on your system?
- Can you statically link programs which use sockets?
- Given an arbitrary collection of object files, can you link them statically (i.e., there is no special object format for static vs. dynamic linking)?

If you answered no to any of these questions, then you must port the dynamic linking support on your system. Otherwise, you should require Condor jobs to be statically linked and avoid the trouble of dynamic linking support.

You will also want to know how C, C++, and Fortran programs are routinely linked on your system. `gcc -v` is particularly helpful for C and C++ programs; `f77 -v` might give the same information for Fortran programs. When linking Condor jobs, you will need to replace `crt0.o` (or `crt1.o` on some systems) with condor_rt0.o, either by invoking gcc with the -nostartfiles option or by invoking ld directly. Note especially which objects are removed from the link line with gcc -nostartfiles. You only want to replace crt0.o; all other objects should remain where they were originally on the link line (and so might now need to be specified explicitly).

9.3 Static Linking

If you are only porting static linking support, then you should only need to work with the machdep.<arch>.C file. In this file, you will need to define functions which find the data segment start and end address, find the stack segment start and end address, determine whether the stack grows down or up, and return the index into the jmp_buf (used by setjmp and longjmp) where the stack pointer is stored.

You will need to experiment to find how to compute these values on your system. Look at other machdep.<arch>.C files for examples. The Makefile in this directory will build a machdep_test program which you can use to test your port.

9.4 Dynamic Linking

If you are porting dynamic linking support, you will need to support a larger range of calls in the machdep.<arch>.C file. Look at the machdep.Solaris.C file for an example. This file uses ioctl and the /proc file system to find the bounds of segments in the process's address space. If you do not have the /proc file system on your architecture, then you should contact the condor development team for advice on alternatives.

image.C is the file which contains most of the actual checkpointing functionality. By default, this file is configured for static checkpointing. You will need to add your architecture's flag to ifdef's in this file so that the dynamic checkpointing code is included instead. Look for ifdef's which contain the keyword Solaris.

9.5 Standalone Mode

You should first test your port of the checkpointing library in standalone mode. In this mode, you can checkpoint and restart executables without submitting them to your Condor pool. In the condor_test_suite directories, you should build the executables which end in .standalone. Some of these test programs call the ckpt() function, which causes an explicit checkpoint; others require you to send the process a SIGTSTP or control-Z to cause the checkpoint. To restart from a checkpoint file, run the executable with the command line arguments -_condor_restart <ckpt_file>.

9.6 Remote Mode

Once standalone mode checkpointing looks good, you can start testing in remote mode by submiting jobs to your pool. The remote mode executables in the test suites end in .remote and their submit files end in .cmd. Bugs found during remote mode testing are usually due to problems with remote system calls rather than with the checkpointing library.

10 Final Notes

That's it! Congratulations ! Thank you for porting Condor. If you have any questions or suggestions, feel free to contact us via email at condor-admin@cs.wisc.edu.