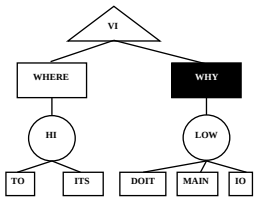
Guide to Porting IPS

IPS Version 5.0 (October 7, 1992)

Bruce Irvin

University of Wisconsin-Madison



© Copyright 1992 All Rights Reserved.

--

--

1. Introduction

IPS has been ported to DECstation, Sun, Vax, Sequent Symmetry (Dynix and Dynix/ptx), and Cray Y-MP computers. To take advantage of our past experiences, and to make future ports of IPS easier, we provide this porting guide. This guide describes the areas of IPS that are machine dependent, shows how the machine dependent features are implemented for the currently supported architectures, and gives instructions for implementing the machine dependent features on a new architecture.

The IPS instrumentation library contains most of the machine dependent code. Sections 2-7 discuss porting issues for various parts of the instrumentation library. The IPS slave is discussed in section 8, the master is discussed in section 9, and the EDCU library is discussed in section 10. The guide concludes with a description of how to test a new IPS implementation to make sure that it functions properly.

We assume that you have installed the IPS sources in a directory and have assigned the shell variable *$ips* to be that directory. Whenever we refer to *$ips* we expect you to either use the shell variable or substitute the full path name of the location of the IPS sources. We also assume that you have the IPS sources for at least one other supported architecture.

This is a preliminary version of this guide. We are currently working on a more expanded version and will appreciate any comments. Send them to rbi@cs.wisc.edu

2. Getting Started

The first thing that you should do is create separate subdirectories in *$ips/lib* for your new architecture. These directories should be of the form instlib.<archname>, crt0.<archname>, and syslib.<archname>, where <archname> refers to the new architecture (eg. MIPS). The second thing you should do is to examine the sources for one of the existing ports of the system. Try to study the sources of the system that most resembles yours. For example, if your target system uses big endian byte ordering, study the SPARC sources.

3. System Calls

IPS automatically traces several system calls including _exit(), accept(), bind(), close(), connect(), dup(), dup2(), execve(), fork(), open(), pipe(), read(), readv(), recv(), recvfrom(), recvmsg(), send(), sendto(), shutdown(), socket(), socketpair(), wait(), wait3(), write(), and writev(). To trace a system call, the instrumentation library provides a dummy call for the system call. The dummy call calls the real system call, writes one or more trace records, and returns to the caller. To make this transparent to the user, we

give the dummy calls the same names as the real system calls, and change the names of the real system calls. We refer to such calls as *instrumented system calls*.

Here is an example of what happens when a user's program calls the instrumented system call open():

```
/* This is a simple user program */
 main()
 {
 int fdescriptor;

     fdescriptor = open("/tmp/myfile",
                        O_CREAT | O_RDWR, 0644);
 }
```

Above, open() refers to the following dummy system call:

```
/*
 *  IPS instrumented open() -- Call the real system call,
 *      and write a trace record for the open.
 */

   open(path, flags, mode)
   char *path;
   int flags, mode;
   {
    int ret;

      /* Call the real open() syscall */
       ret = _open(path, flags, mode);

      /* Write a trace record for the open */
       <Code for tracing>

      /* Return to the user */
       return (ret);
   }
```

Notice that the actual system call is now known as *"_open()"* . Our standard convention for changing a system call's name in the instrumentation library is to prepend an underscore character. For example, the system call _exit() becomes __exit().

To change the names of the instrumented system calls, you will need the system call sources for your machine. Copy the code from the real system calls to your new syslib directory. Prepend an underscore to each of the source code filenames (e.g. accept.c becomes _accept.c). Next, edit each file and prepend an underscore character to the

declaration of each system call (e.g. change accept() to _accept()). After you have added the underscores, copy a Makefile from one of the other syslib directories to your syslib, and execute *make* . If the system calls for your architecture are written in assembly then you may need to change the Makefile to use the assembler instead of the C compiler.

The directory *$ips/lib/instlib* contains the sources for the (machine independent) dummy system calls. To compile the dummy system calls, copy the Makefile from another architecture's instlib.<archname> directory to your architecture's instlib.<archname> directory and execute **make** .

One file in *$ips/lib/instlib* does not relate to any particular system call. The file is called trace.c, and it contains the function trace() which is called by nearly every instrumented system call.

This file should compile normally with the Makefile in your architecture's instlib.<archname>, but it contains a significant amount of machine dependent code that will need to be updated for your new architecture. These updates should be done with conditional compilations using `#ifdef` statements. Because most of the updates pertain to timing, we will defer discussion of exactly how to update trace.c until section 6.

4. Vfork

The *vfork()* system call is unique in that it does not fit well into the IPS instrumented system call model. When a process calls vfork(), both the parent and the child process execute in the same address space with the same stack until the child performs an execve(). The child must not return from its current function because it would disrupt the parent's stack. However, when vfork() is instrumented, the child must return to the calling function from the dummy call to reach the user's execve() call.

To fix this conflict you must write the vfork() dummy system call so that the child returns to the caller without disrupting the stack. Usually, you can examine the real vfork() (now called _vfork()) to see how to handle this problem.

5. IPS Initialization

The directory *$ips/lib/crt0.<archname>* will eventually contain all of the IPS initialization code and code for the functions mcount() and mcend(). Most of this code is machine dependent and must be written from scratch. This section gives advice on how to implement the IPS initialization code and explains each function in the crt0 directory at a fairly high level. For a full understanding of the crt0 functions, you should examine their implementations for the currently supported architectures.

First, obtain the file *crt0.c* (sometimes called *crt0.s*) for your machine. This file should contain a call to a function called *monstartup()* . You should modify the call to

monstartup() by passing *argv* as the last parameter.

Next you should start a file called *imon.c* This file will contain the functions *monstartup()* , *_mcleanup()* , *oflow()* , *monitor()* , and *moncontrol()* . It must also contain definitions of the following global variables.

tbuf, bufptr, tbufptr, tbufend, tbuftemp -- Pointers for the trace record buffer.

imonDir, IMONfName, AUXfName -- Names of trace directories and trace files.

IMONfd, auxFD -- Trace file descriptors.

_p_1, _p_2, _s_1, _s_2, _q_1, _q_2 -- Clock pointers.

The function monstartup() is called as a process begins. It must allocate the trace record buffer (TRB), initialize the process timer and wall-clock pointers, open and initialize the trace record files, and write the IPS_INIT trace record. Most of these operations are machine independent, and much of the code may be copied from the monstartup() function of another supported architecture. Details of the process timer and wall-clock are covered in section 6.

The function _mcleanup() is called when a process exits. It must write the IPS_EXIT record, write the TRB out to the trace file, and close the auxiliary trace file. Most of _mcleanup() can probably be copied from another architecture's imon.c file.

The function oflow() is called whenever the TRB fills. It writes the TRB out to the trace file. It's code is machine independent and can probably be copied from another architecture's imon.c file.

The function monitor() calls _mcleanup() under various circumstances. It is machine independent and may be copied from another architecture's imon.c file. The function moncontrol() should never be called, but you must include a declaration for it in imon.c.

The file *mcount.c* must contain the function *mcount()* and a declaration for the global integer *oldUtime* . The function mcount() is called at the beginning of every function call. It can best be described with the following pseudo code:

```
mcount()
 {
  proc_time = current process time (in microseconds);

  if (proc_time == oldUtime) {
    if (TRB cannot take any more records) {
      call oflow();
      reset the TRB pointer;
    }

    temporary = current wall-clock time (in microseconds);
    set the most significant bit of temporary to 1;
    copy temporary to the TRB;
    temporary = return address of mcount();
    set the most significant bit of temporary to 0;
    copy temporary to the TRB;
  }
  else {
    trace(IPS_CALL,return address of mcount());
  }
  return;
 }
```

Usually mcount() must be written in assembly language because there is no other way to access a function's return address. Since mcount() is called at the start of each function call in a user's program, it is important to make it efficient.

The file *mcend.c* must contain the function *mcend()* . The function mcend() is called before the end of each function call. It's operation may be explained with pseudo code:

```
mcend()
 {
  proc_time = current process time (in microseconds);

  if (proc_time == oldUtime) {
    if (TRB cannot take any more records) {
      call oflow();
      reset the TRB pointer;
    }
    temporary = current wall-clock time;
    set the most significant bit of temporary to 0;
    copy temporary to TRB;
  }
  else {
    trace(IPS_RET);
  }
 }
```

Usually, mcend() may be copied from the MIPS crt0 directory. However, you may want to rewrite it for efficiency.

Once you have written all of the code in your crt0 directory, you should write a Makefile to compile it. All of the functions should be linked together into a file called icrt0.o. Be sure that crt0 is compiled (assembled) so that it calls monstartup() before main().

6. Timing

IPS must have access to timers for process time and wall-clock (real) time. Unix provides various system calls for accessing timers including **getrusage()** , and *gettimeofday()* , and IPS will function properly with these system calls. However, you may want to modify your kernel to allow more efficient access to process and wall-clock timers. This section describes how to make sure that IPS timing works properly with getrusage/gettimeofday, and describes modifications that we have made to DEC/Ultrix, BSD, and Sequent/Dynix kernels for reasons of efficiency.

The time spent in the CPU on behalf of a given process (process time) can usually be found in the u-area. The u-area is a structure that contains the swapable part of the process's process table entry. It includes user time (time spent executing outside the kernel) and system time (time spent executing inside the kernel). The u-area is usually defined as a **struct user** in the file */usr/include/sys/user.h* , and its address is usually given by the constant UADDR or the constant USRSTACK.

For the VAX architecture, the u-area may be read directly by a user program. However, the Sequent and MIPS architectures restrict access to the u-area, and we have therefore modified those kernels to allow direct reads of the u-area. The modified Sequent kernel removes read protection of the u-area page, while the MIPS kernel remaps the u-area to an address that may be directly accessed by user code.

To determine whether the u-area on your architecture is readable, you should compile and run the program *ucheck* which may be found in the *$ips/misc/utests* . This program gets the process time using the system call getrusage() and then attempts to get the process time from the u-area. If the program completes with no errors and the process times are comparable, then the u-area is accessible. However, if the program fails to get the correct process time from the u-area, then the kernel will need to be modified to allow u-area access.

Wall-clock time can be obtained with a gettimeofday() system call. However, if we want to directly access the wall-clock with a memory reference then we need to know the virtual address to which the wall-clock is mapped. On the Sequent, this address is called *usclk_base* . For the MIPS and Vax architectures, the wall-clock addresses are not available to the user, and therefore we have modified those kernels to allow memory access to their wall-clocks.

To implement timing in IPS you must first initialize the clock pointers in the function monstartup() in the file *$ips/lib/crt0.<archname>/imon.c* , then you must make sure that the clocks are correctly referenced in the functions mcount(), mcend(), and trace().

The file $ips/config.h should specify whether either of the two clocks are readable. If UNREADABLE is used, then the code should assume that the u-area is unreadable. In this case, there should be no user clock initialization in the function monstartup(). Also mcount() and trace() should use getrusage() to obtain the process times. If WCUNREADABLE is defined, then the code should assume that the wall-clock is not readable, and gettimeofday() should be called when the wall-clock time is needed. It is important for the instrumentation code to handle these cases so that IPS may run on machines that do not have modified kernels.

7. The Compiler Driver

A compiler driver manages the compilation and linking of source code to executable code. To instrument a program for IPS, the compilation and linking process must be modified to insert mcount() and mcend() at the beginning and ending of each function and to link the object files icrt0.o and libipstrace.a into the program. Therefore each port of IPS must include its own compiler driver for each supported language which recognizes the IPS switch -pi and modifies compilation and linking accordingly.

This section describes how to make an IPS compiler driver. Before building compiler drivers for a new port, you should make a directory for your drivers' sources called *$ips/compilers.<archname>* .

7.1. Modifying the Original Compiler Driver

One way to make an IPS compiler driver is to change the original driver's source code. To do this, you must have access to the original compiler driver. It is usually called cc.c, fortran.c, or driver.c, and may or may not be available with your system software.

You must make three changes to the original source code. First, add a command line switch *-pi* . Next, have the compiler insert calls to mcount() at the beginning and mcend() at the end of each function compiled with -pi. Finally, you must make sure that the -pi switch triggers the proper loading of the IPS instrumentation libraries.

To insert calls to mcount() and mcend(), each source file should be compiled to assembly code, and the compiler should then run edit the assembly code before the code is compiled to object format. Typically this editing is accomplished with the help of a stream editor such as sed. The editor must identify the beginning and ending of each function in the assembly code and insert calls to mcount() and mcend().

If the -pi switch is specified on the command line, then the compiler driver should call the loader with parameters in a similar manner as when the -pg switch is specified for gprof. However, icrt0.o should be used instead of mcrt0.o, and libipstrace.a should replace libprof.a.

7.2. Writing a Shell Script

If your compiler has a *dry run* option which previews the steps to be taken by the driver to compile a program, then you can build a shell script to replace the compiler driver. This approach is used for the IPS SPARC C compiler driver.

If the sources for the compiler driver are not available to you, then you must build a shell script to drive the proper compilation steps and insert IPS trace code. This is the approach taken for the IPS CRAY C compiler driver.

8. Porting the Slave

Most of the IPS slave should not need to be changed. However, the file stab_<archname>.h will need to be written for your architecture. This file contains code for reading symbol table information from a binary executable. There are two things to keep in mind. First, your symbol table format might be similar to one of the other architectures. Second, you should use the program $ips/misc/tools/imondump to test your symbol table parsing code.

9. Porting the Master

Very little of the IPS master should need to be changed. You will want to examine the file $ips/config.h and the file $ips/arch.c Since the master is an X application, you may need to change it to account for differences in your system's X windows implementation.

10. Porting the EDCU Library

The EDCU library is located in $ips/edcu. Most machine dependencies are contained in EDCUs themselves and not in the library, so the library should not need to be changed much. You will, however, want to write new EDCUs. See the IPS Reference Guide for information on writing your own EDCUs.