

Chapter 11

Software Development



***HP-UX Handbook
Revision 13.00***

TERMS OF USE AND LEGAL RESTRICTIONS FOR THE HP-UX RECOVERY HANDBOOK

ATTENTION: PLEASE READ THESE TERMS CAREFULLY BEFORE USING THE HP-UX HANDBOOK. USING THESE MATERIALS INDICATES THAT YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THESE TERMS, DO NOT USE THE HP-UX HANDBOOK.

THE HP-UX HANDBOOK HAS BEEN COMPILED FROM THE NOTES OF HP ENGINEERS AND CONTAINS HP CONFIDENTIAL INFORMATION.

THE HP-UX HANDBOOK IS NOT A PRODUCT QUALITY DOCUMENT AND IS NOT NECESSARILY MAINTAINED OR UP TO DATE. THE HP-UX HANDBOOK IS HERE MADE AVAILABLE TO HP CONTRACT CUSTOMERS FOR THEIR INTERNAL USE ONLY AND ON THE CONDITION THAT NEITHER THE HP-UX HANDBOOK NOR ANY OF THE MATERIALS IT CONTAINS IS PASSED ON TO ANY THIRD PARTY.

Use of the HP-UX Handbook: Hewlett-Packard Company ("HP") authorizes you to view and download the HP-UX Handbook only for internal use by you, a valued HP Contract Customer, provided that you retain all copyright and other proprietary notices contained in the original materials on any copies of the materials. You may not modify the HP-UX Handbook in any way or publicly display, perform, or distribute or otherwise use them for any public or purpose outside your own business. The materials comprising the HP-UX Handbook are copyrighted and any unauthorized use of these materials may violate copyright, trademark, and other laws. If you breach any of these Terms, your authorization to use the HP-UX Handbook automatically terminates and you must immediately destroy any downloaded or printed materials.

Links To Other Web Sites: Links to third party Web sites provided by the HP-UX Handbook are provided solely as a convenience to you. If you use these links, you will leave this Site. HP has not reviewed all of these third party sites and does not control and is not responsible for any of these sites or their content. Thus, HP does not endorse or make any representations about them, or any information, software or other products or materials found there, or any results that may be obtained from using them. If you decide to access any of the third party sites linked to this Site, you do this entirely at your own risk.

Disclaimer: THE HP-UX HANDBOOK IS PROVIDED "AS IS" WITHOUT ANY WARRANTIES OF ANY KIND INCLUDING WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF INTELLECTUAL PROPERTY. HP further does not warrant the accuracy and completeness of the materials in the HP-UX Handbook. HP may make changes to the HP-UX Handbook at any time without notice. The HP-UX Handbook may be out of date, and HP makes no commitment to update the HP-UX Handbook. Information in the HP-UX Handbook may refer to products, programs or services that are not available in your country. Consult your local HP business contact for information regarding the products, programs and services that may be available to you.

Limitation of Liability: IN NO EVENT WILL HP, ITS SUPPLIERS, OR OTHER ANY THIRD PARTIES MENTIONED IN THE HP-UX HANDBOOK BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, THOSE RESULTING FROM LOST PROFITS, LOST DATA OR BUSINESS INTERRUPTION) ARISING OUT OF THE USE, INABILITY TO USE, OR THE RESULTS OF USE OF THE HP-UX HANDBOOK, WHETHER BASED ON WARRANTY, CONTRACT, TORT OR ANY OTHER LEGAL THEORY AND WHETHER OR NOT ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS DOES NOT APPLY IN CASE OF INTENT OR IF LIABILITY IS LEGALLY STIPULATED. IF YOUR USE OF THE HP-UX HANDBOOK RESULTS IN THE NEED FOR SERVICING, REPAIR OR CORRECTION OF EQUIPMENT OR DATA, YOU ASSUME ALL COSTS THEREOF.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Applicable Laws: These Terms will be governed by and construed in accordance with the laws of the State of California, without giving effect to any principles of conflicts of laws.

General: HP may revise these Terms at any time by updating this posting. *Revised Oct 2013*

© Copyright 2000-2006, 2013 Hewlett-Packard Development Company, L.P

FEEDBACK or QUESTIONS:

(please use subject syntax:

please email essam.ackleh@hp.com

HP-UX Handbook v13.00 Chapter <YY> - <Feedback Title>

TABLE OF CONTENTS

Preface	4
The Build Process	4
Source Code	5
Compiling	7
Objects	9
Linking	13
Executables	16
Program Execution	21
Load the Program	23
Load Shared Libraries	23
Resolve Symbols	25
Execute main()	27
Program Termination	28
Process Memory	29
Products On HP-UX	37
Development Tools	38
Runtime Environments	40
Java	41
Scripting Languages	43
Tools	48
Development Utilities	48
Static Information	51
Runtime Analysis	55
Solving Common Problems	64
Installation Problems	64
Build Problems	67
Problems Starting A Program	72
Error Messages From The Dynamic Loader	72
Program Aborts	76
Environment Variables	96
Additional Information	100
Official HP Websites	100
HP Internal Links	101
External Links	101

Preface

This chapter is intended to give an overview over issues on software development in general, and on HP-UX in particular. It is not intended to teach programming languages, and if at all, only basic skills in programming languages, and probably a little shell programming know-how are required to understand its contents.

The Software Development chapter covers topics like building a program, program execution, products and tools are available on HP-UX for software development, common problems and approaches how to solve them.

The Build Process

The process to build an executable program consists basically of the following steps:



A programmer writes a program in his preferred [programming language](#). The program code is stored in an ASCII file usually called the [source file](#). Because the processor does not understand programming languages, the source code must be converted into machine code. This is what the [compiler](#) does. It creates the [object file](#) which contains the machine code and a table of contents and references. Usually a program is made from multiple objects, each having contents and references. The [linker](#) creates the [executable](#) from the objects by binding references to contents.

For easier creation of executables every compiler comes with a so-called front-end command. It is used as a single interface to do one or more steps of the build process at once. It knows which default parameters to pass to each build step to ease the creation of an executable program from a source file.

The compiler front-ends know various input file types and distinguish them by their file name extension. They will pass them directly to the appropriate sub-process:

File Type	Subprocess	Extensions
source files	preprocessor (if used by the language)	C: .c aC++: .C, .cc, .cpp, .cxx Fortran: .f, .F, .f90
intermediate files	compiler	.i, .o
all other files	linker	any

By default, the front-ends will execute all necessary build steps to create an executable from the input files passed to it. In the simplest case the build command to create a program from a source file is:

```
<frontend command> -o <program name> <source file>
```

To execute only one step of the build process, additional arguments must be passed to the frontend. Which options are available and much other information about the frontend and the compiler itself can be found in the appropriate frontend manpages which are installed with the [compiler product](#).

Source Code

This page is intended to give a somewhat simplified view over the structure of programs and its sources. The things discussed will be demonstrated with little pieces of ANSI C code, which hopefully can be understood even without programming knowhow.

Program Structure

Programs are organized in functions. A function is (as you might remember from your math classes) a construct that takes one or more operands, and returns a result. In a program functions are also referenced to as procedures, subroutines or methods, the operands are called arguments or parameters, and the result is usually called the return value. The function body contains the instructions that process the arguments to generate the return value. Often the major task of a function is to process the arguments rather than generating a return value. The instructions can also contain calls to other functions to execute subtasks.

A program is intended to process data, so besides functions it also needs means to store data. A piece of data is called a variable.

Functions and variables are often abstractly called symbols. Symbols are only distinguished by name. We decide between global and local symbols. As the name implies, global symbols can be accessed in the whole program, while local symbols are only available within a certain area of the program. This access range is called the scope of the variable.

Programmatical Conventions

In a program source you typically will find declarations and definitions for symbols. A declaration is used to specify names and types of functions, function arguments, return values and variables. It determines how to use a symbol so it can be referenced, but it doesn't create it.

```
int main();
```

This is a declaration of a function named `main` which takes no argument and returns a value of type `int`. To create the symbol a definition is required:

```
int main() {  
    :  
    /* function code */  
    :  
}
```

The definition of a function contains the function body. It must specify the same arguments and return code as the declaration. A symbol definition implies the declaration, so if there is a symbol definition in a source, no separate declaration is required.

The separation of definitions and declarations allows encapsulation and protection of source code. To reference a symbol only the declaration is needed, so we can hide what's going on inside:

```
int x(int);  
  
int main() {  
    return x(1);  
}
```

Function `x()` might be defined somewhere else, but we can call it here because we have its declaration.

Source Code Organization

Usually there are two kinds of files that reflect the programmatical conventions above. Source files contain symbol definitions. Rather than inserting symbol declarations in all source files, they are collected in header files. The declarations in the header files can then be used by referencing the header files. Header files in turn can also reference other header files.

```
$ cat HelloWorld.c  
#include <stdio.h>  
  
int main() {  
    printf("Hello World!\n");  
}  
$
```

The file `HelloWorld.c` is a source file and we will use it as our test case for the rest of the course. It references the header file `stdio.h` and defines the function `main()`.

You might wonder where this `stdio.h` file can be found. This is one of the header files that are provided with the operating system, because it provides the declarations of symbols that are commonly used by a lot of programs. These files are collected inside the default path for system header files, `/usr/include`. The compiler frontend knows this default path so we don't need to care for it.

From the programmers point of view `main()` is the function which is called first when the program is executed. Our `main()` calls the function `printf()`. Its declaration can be found in `stdio.h`, but not its definition. The function is already compiled and linked into a [library](#) as we will see later. The function `printf()` produces a formatted output of its arguments.

Compiling

When talking of compiling, usually two steps are meant: preprocessing and compiling. But most of the time preprocessing and compiling is looked at as a single step. Note that not every [programming language](#) has the concept of header files (e.g. java, fortran). In such cases you only have source files and there is no preprocessing phase.

Preprocessing

The source file first is passed to the preprocessor. Among other things the preprocessor resolves the references to header files which means wherever it finds a reference to a header file, it includes the contents of the header file at that position, thus generating one big output file, the preprocessed file.

For the [HelloWorld.c](#) source file the preprocessor would insert the contents of the header file `stdio.h` at the beginning of the preprocessed file.



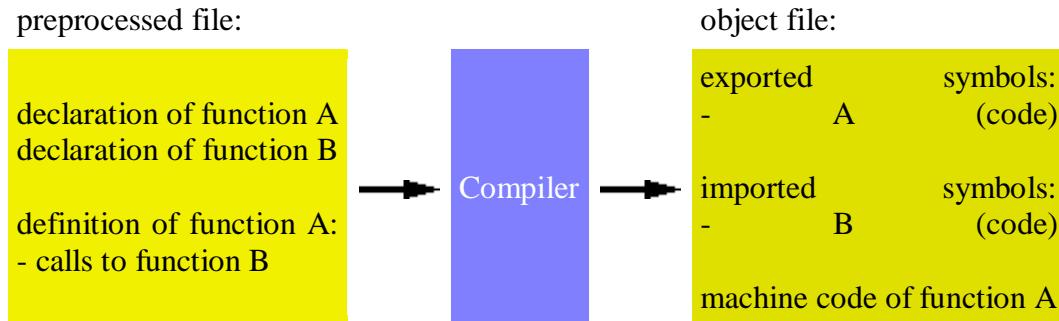
The preprocessor knows some default paths where to look for files to be included. If a header file exists in another path, this path must be passed to the preprocessor via a command line option.

The preprocessed file has the same name as the source file, but with the extension `.i`. To tell the compiler frontend to only preprocess a source, the `-P` option must be added.

Compiling

The `.i` file is then passed to the compiler which translates the source code into machine code, creates a table of contents and references and stores both in an object file. The table of contents is called the symbol table and holds all symbol names that are:

- defined in the object and can be referenced from other objects (exported symbols)
- referenced in the object but not defined here (imported symbols)



To stop the build process after the compilation phase, the option `-c` must be used. The object file has the same name as the source file, but with the extension `.o`.

Compiling `HelloWorld.c`

`cc` is the frontend command of the C compiler. It would subsequently execute the preprocessor, the compiler and the linker and remove intermediate files (`.i` and `.o` files).

At this point we only want to compile a source and keep the object file, so we use the `-c` option. The option `-v` can be used to see the details of the build process. It is common to all HP compiler frontends to switch on verbose mode:

```

On PA:
$ cc -c -v HelloWorld.c
cc: NLS PATH is
/opt/ansic/lib/nls/msg/%L/%N.cat:/opt/ansic/lib/nls/msg/C/%N.cat:
cc: CCOPTS is not set.
cc: INCLUDIR is INCLUDIR=/usr/include
/opt/langtools/lbin/cpp.ansi HelloWorld.c /var/tmp/ctmAAAa29882 -$ -
D_hp9000s700 -D_hp9000s800 -D_hppa -D_hpuX -D_unix -D_XOPEN_UNIX -D_ILP32
-e -D_PA_RISC2_0 -D_HPUX_SOURCE -D_STDC_EXT_
cc: Entering Preprocessor.
/opt/ansic/lbin/ccom /var/tmp/ctmAAAa29882 HelloWorld.o -FHelloWorld.c -
Econstlit -Oq00,al,ag,cn,Lm,sz,Ic,vo,lc,mf,Po,es,rs,sp,in,vc,pi,fa,pe,
Rr,F1,pv,pa,nf,cp,lx,st,ap,Pg,ug,lu,lb,uj,dp,fs,bp,wp,Ex,mp,
rp,ap,dn,Sg,pt,kt,Em,pc,np! -Ae
$
```

On IA:

```
$ cc -c -v HelloWorld.c
/opt/aCC/lbin/ecom -architecture 32 -ia64abi all -ext on -lang c -exception
off -sysdir /usr/include -inline_power 1 -link_type dynamic -fpeval float
-fpevaldec Decimal32 -tls dyn on -target_os 11.31 --sys_include /usr/include
-D BIND LIBCALLS -D Math_errhandling=MATH_ERREXCEPT -D hpx -D unix
-D ia64=1 -D ia64_ =1 -D BIG_ENDIAN=1 -D ILP32 -D HP_cc=62300
-D STDC_EXT -D HPUX_SOURCE -D INCLUDE_LONGLONG -D INLINE_ASM
-D FLT_EVAL_METHOD=0 -D DEC_EVAL_METHOD=0 -ucode hdriver=optlevel%1%
-plusolistoption -O106all! -plusolistoption -O113moderate! -plusoption
-Oq01,al,ag,cn,sz,ic,vo,Mf,Po,es,rs,Rf,Pr,sp,in,cl,om,vc,pi,fa,
pe,rr,pa,pv,nf,cp,lx,Pg,ug,lu,lb,uj,dn,sg,pt,kt,em,np,ar,ap,rd,dl,fs,
,bp,wp,pc,mp,lr,cx,cr,pi,so,Rc,fa,ft,fe,st,lc,B1,sr,Qs,do,ib,pl,
,sd,ll,rl,d1,Lt,ol,f1,lm,ts,rd,dp,If!
HelloWorld.c
```

The verbose output shows which environment variables are used and how they are set, and which arguments are passed to the compilation subprocesses.

On PA, `/opt/langtools/lbin/cpp.ansi` is the preprocessor, `/opt/ansic/lbin/ccom` is the compiler.

On IA, `/opt/aCC/lbin/ecom` is the compiler and preprocessing is built-in.

Compiling In 64-bit Mode

Without adding special options to the compilation command, all compilers will generate 32-bit objects. To create 64-bit objects, the compiler option `+DD64` must be used.

Objects

As mentioned before, objects contain machine code and a symbol table. We can look at the symbol table of `HelloWorld.o` with [nm\(1\)](#):

```
$ nm HelloWorld.o

On PA:
Symbols from HelloWorld.o:

Name           Value  Scope   Type      Subspace
C$10          |      40|static|data    |$LIT$
main          |      0|extern|entry    |$CODE$
printf        |      |undef|code     |
$
```

```
On IA:
Symbols from HelloWorld.o:

[Index] Value Size Type Bind O Shndx Name
[0] | 0| 0|NOTYP|LOCAL|0| UNDEF|
[8] | 0| 0|SECT
|LOCAL|0|.HP.opt_annot|.HP.opt_annot
[7] | 0| 0|SECT
|LOCAL|0|.IA_64.unwind_info|.IA_64.unwind_info
[3] | 0| 0|SECT
|LOCAL|0|.debug_actual|.debug_actual
[2] | 0| 0|SECT |LOCAL|0|.debug_line|.debug_line
[6] | 0| 0|SECT
|LOCAL|0|.debug_procs_abbrev|.debug_procs_abbrev
[4] | 0| 0|SECT |LOCAL|0|.rodata|.rodata
[5] | 0| 0|SECT |LOCAL|0|.text|.text
[1] | 0| 0|FILE |LOCAL|0| ABS>HelloWorld.c
[10] | 0| 112|FUNC |GLOB|0| .text|main
[9] | 0| 0|FUNC |GLOB|0| UNDEF|printf
```

`nm(1)` lists all symbol names and shows address, scope, symbol type and the location of the symbol in the object.

`main` has an external scope which means it is globally accessible, a type of `entry/FUNC` means that the symbol is inside the object, and it is contained in the code section of the object. The scope of `printf` is undefined, it is only a code reference. The value column shows the location (address) of the symbol inside the object.

For 64-bit objects, the `nm(1)` output looks somewhat different, but basically contains the same information:

```
$ nm HelloWorld.o

On PA:
Symbols from HelloWorld.o:

[Index] Value Size Type Bind O Shndx Name
[0] | 0| 0|NOTYP|LOCAL|0| UNDEF|
[4] | 0| 0|OBJT |LOCAL|0| .text|$PIC$0
[2] | 0| 0|SECT |LOCAL|0| .bdata|.bdata
[1] | 0| 0|SECT |LOCAL|0| .btext|.btext
[3] | 0| 14|OBJT |LOCAL|0| .rodata|C$10
[5] | 0| 72|FUNC |GLOB|0| .text|main
[6] | 0| 0|FUNC |GLOB|0| UNDEF|printf

On IA:
```

```
Symbols from HelloWorld.o:
```

[Index]	Value	Size	Type	Bind	O	Shndx	Name
[0]		0	0 NOTYP	LOCAL 0			UNDEF
[8]		0	0 SECT				
LOCAL 0 .HP.opt_annot .HP.opt_annot							
[7]		0	0 SECT				
LOCAL 0 .IA_64.unwind_info .IA_64.unwind_info							
[3]		0	0 SECT				
LOCAL 0 .debug_actual .debug_actual							
[2]		0	0 SECT				
LOCAL 0 .debug_line .debug_line							
[6]		0	0 SECT				
LOCAL 0 .debug_procs_abbrev .debug_procs_abbrev							
[4]		0	0 SECT	LOCAL 0			
.rodata .rodata							
[5]		0	0 SECT	LOCAL 0	.	text	text
[1]		0	0 FILE	LOCAL 0			
ABS HelloWorld.c							
[10]		0	112 FUNC	GLOB 0	.	text	main
[9]		0	0 FUNC	GLOB 0	UNDEF		printf
\$							

All addresses of exported symbols are kept relative, all references to imported symbols are left unspecified. This is important for the link phase. The objects are called relocatable, which means they can be placed at any address in an executable.

Object files can be collected in archive libraries, using the [ar\(1\)](#) command. This archiving does not directly contribute to the build process. It is just a possibility to provide a set of objects in a single file.

Name Mangling

Object oriented programming languages like aC++ and Java allow multiple methods with the same name, but different parameters. This is called “overloading”. Because the [linker](#) doesn’t know and doesn’t care for the parameters of a method, the compiler must somehow encode the parameter types in the symbol names, to distinguish overloaded methods. This is called name mangling. The resulting symbol names in the objects and executables can be displayed with [nm\(1\)](#):

```
$ cat x.C
void sub(int) {};
void sub(char) {};
$ aCC -c x.C
$ nm x.o
```

On PA:

Symbols from x.o:

Name	Value	Scope	Type	Subspace
------	-------	-------	------	----------

sub__Fc		12 extern entry	\$CODE\$	
sub__Fi		0 extern entry	\$CODE\$	
\$				

On IA:

Symbols from x.o:

[Index]	Value	Size	Type	Bind	O Shndx	Name
---------	-------	------	------	------	---------	------

:						
[5]		0	0 SECT	LOCAL 0	.text .text	
[4]		0	0 SECT	LOCAL 0	.text .text	
[11]		0	32 FUNC	GLOB 0	.text _Z3subc	
[10]		0	32 FUNC	GLOB 0	.text _Z3subi	
[1]		0	0 FILE	LOCAL 0	ABS x.C	

To find out the original name of a method, as it exists in the source code, either `nm++(1)` can be used on the object, or the mangled method names can be passed to `c++filt(1)`. These tools revert the mangled name back to the unmangled form, which includes the parameter types:

\$ nm++ x.o

On PA:

Symbols from x.o:

Name	Value	Scope	Type	Subspace
------	-------	-------	------	----------

sub(char)		12 extern entry	\$CODE\$	
sub(int)		0 extern entry	\$CODE\$	

On IA:

Symbols from x.o:

[Index]	Value	Size	Type	Bind	O Shndx	Name
---------	-------	------	------	------	---------	------

:						
[5]		0	0 SECT	LOCAL 0	.text .text	
[4]		0	0 SECT	LOCAL 0	.text .text	
[11]		0	32 FUNC	GLOB 0	.text sub(char)	
[10]		0	32 FUNC	GLOB 0	.text sub(int)	
[1]		0	0 FILE	LOCAL 0	ABS x.C	

On PA:

\$ c++filt sub__Fc sub__Fi
sub(char)
sub(int)
\$

```
On IA:
$ c++filt _Z3subc _Z3subi
sub(char)
sub(int)
$
```

`nm++(1)` is a script that calls `nm(1)` internally and uses `c++filt(1)` to demangle the mangled names in its output. Java and aC++ on HP-UX use the same name mangling algorithm, so `c++filt(1)` can also be used to demangle java method names.

Linking

The purpose of the linker is to create an executable file from a number of objects. Programs and shared libraries are considered to be executable files, and the linker can create both. A shared library is a collection of code and data that can be loaded to a program at [runtime](#). The linker is often also referenced to as the loader.

The Link Process

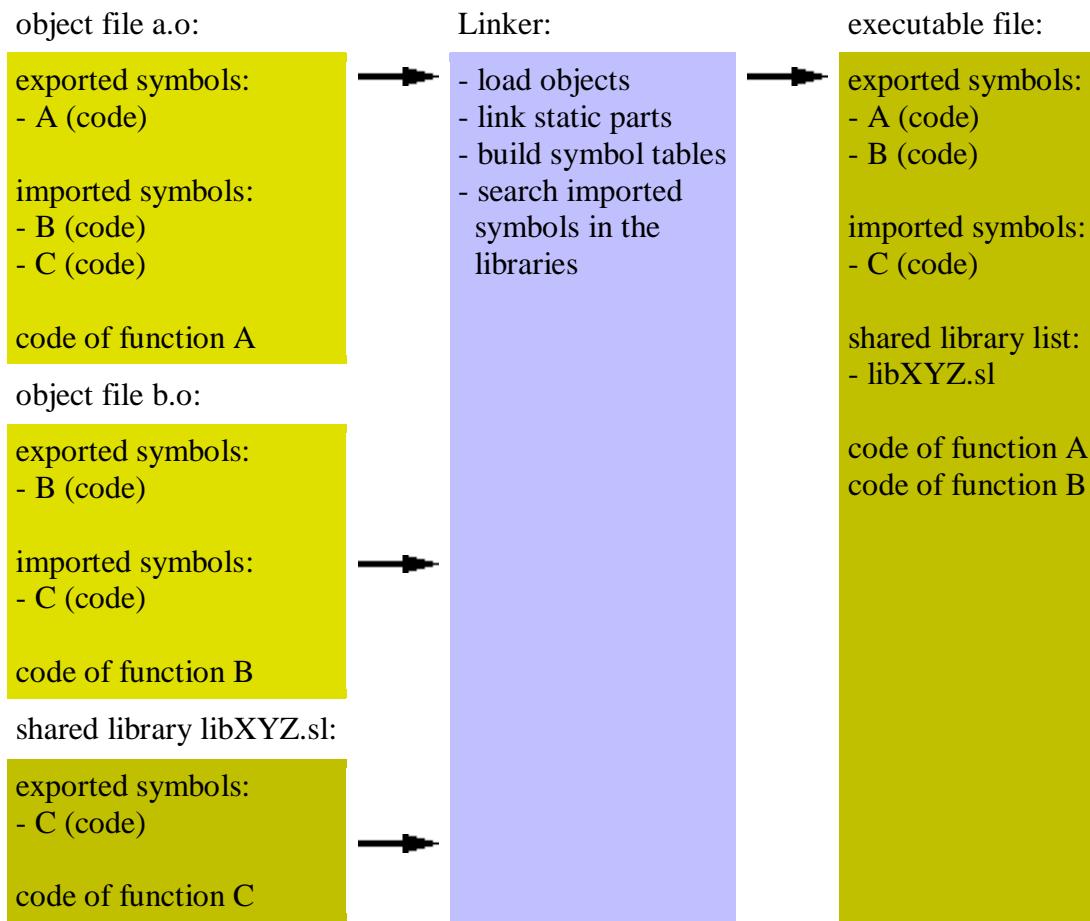
The linker `ld` reads all specified objects, puts them together into the binary file, calculates the final addresses of all symbols and updates references to them. Libraries can be linked to the binary using the `-lname` option. This causes the linker to search for a shared and an archive library (in that order per default) with the name `libname.*`. The default search path and extension of the library differ on the different platforms:

Platform	Location	Extension for	
		Shared Libs	Archive Libs
PA-RISC 32-bit	/usr/lib	.sl	.a
PA-RISC 64-bit	/usr/lib/pa20_64		
IA64 32-bit	/usr/lib/hpux32	.so	
IA64 64-bit	/usr/lib/hpux64		

Other search paths can be added using the `-L` linker option.

If a shared library has been found, only a reference to it is stored in the binary. If the archive version is used, the objects that contain the referenced symbols are extracted and put into the binary in the same way as the objects specified on the command line.

Assuming we have two objects `a.o` and `b.o` and a shared library `libXYZ.sl`, linking them to an executable would work like this:



When linking an executable program, the object `crt0.o` must always be linked in first. This is the startup object which contains code that every executable needs for its initialization (e.g. loading `dld.sl`) and clean termination. If a compiler front end is used to link, it adds `crt0.o` automatically to the linker command.

If a reference to a symbol cannot be resolved within the binary, it is searched for in the list of shared libraries passed to the linker. If the symbol is still not found, an error is reported. Otherwise it is added to the symbol table of the binary as an imported symbol. Symbols of the binary can also be exported, this means they can be referenced from shared libraries. Such symbols are stored in the symbol table as exported symbols. If all imported symbols could be resolved with the shared libraries, the binary is made executable.

In contrast to an archive library, a shared library is not just a collection of objects. Objects cannot be extracted from a shared library. It is linked in the same way as an executable, with the difference that symbols which could not be resolved at link time don't lead to an error. They are stored in the symbol table as imported symbols and resolving them is left to the dynamic loader at runtime, which has more resources to search for symbols that are not available at link time, namely the executable and the other shared libraries linked to it.

Because the linker creates the program file or the shared library, it also gives it its name. If nothing else is specified, the name will be `a.out`. A different name can be selected with the `-o <name>` linker option. Every compiler frontend also knows this option, and if it is specified at the command line, it is passed to the linker.

Linking HelloWorld

It is always recommended to use a compiler frontend to link. We simply pass `HelloWorld.o` to `cc`. We don't need to add the option `-lc`, the frontend command will append it as a default option to the linker command line:

```
cc -v -o HelloWorld HelloWorld.o
cc: NLSPATH is
/opt/ansic/lib/nls/msg/%L/%N.cat:/opt/ansic/lib/nls/msg/C/%N.cat:
cc: CCOPTS is not set.
cc: INCLUDIR is INCLUDIR=/usr/include
cc: LPATH is /usr/lib:/opt/langtools/lib:
/usr/ccs/bin/ld /opt/langtools/lib/crt0.o -o HelloWorld HelloWorld.o -u
main -lc
cc: Entering Link editor.
```

`cc` passes the object and some default parameters to the linker. One of the defaults needed for linking an executable is the object `crt0.o`. It contains the code required for the program startup and initialization, e.g. here is the code that loads `dld.sl|dld.so`. `crt0.o` will always be passed to the linker as the first object.

`ld` picks up our object, finds the entry point `main` and the undefined symbol `printf`. Now it searches subsequently all other objects and libraries to find the definition of `printf`. It finally finds it in the shared library `libc.sl` (PA) or `libc.so` (IA) which is added to the link command via the `-lc` parameter. A reference to `libc.sl | libc.so` is stored in the binary. Herewith `ld` has found all it needs to create the `HelloWorld` executable.

Linking In 64-bit Mode

The linker does not have a special option that switches between 32-bit or 64-bit mode. It simply checks the type of the first object passed to it to decide which executable should be created. When linking executables, this is always `crt0.o`. There is a 32-bit and a 64-bit version of it. Without adding special options to the link command, the frontend will pick the 32-bit `crt0.o` which will cause the linker to do a 32-bit link. With `+DD64` the frontend picks the 64-bit `crt0.o` which will make the linker do a 64-bit link.

Each object and shared library involved in a link phase must be of the same type, either 32-bit or 64-bit. Mixing both does not work and will result in a linker error message.

Also, on IA, the type of executable the compiler creates is either ELF-32 or ELF-64 IA. Either of these types is incompatible with files of type PA-RISC1.1 or PA-RISC2.0 (both are 32-bit) or ELF-64 PA (64-bit).

Building shared Libraries

One major difference between an executable and a shared library is that at runtime the executable will always be loaded to the same virtual address 0x0, while shared libraries must be loadable at any virtual address.

Objects that will be linked into a shared library must contain Position Independent Code (PIC). Position independency means that the symbol address calculation will be done only at runtime, when the shared library is [loaded](#). To generate PIC objects, the `+z` compiler option must be used. Compared to relocatable objects, PIC objects use one more stage of indirection when dereferencing symbol addresses, by using a symbol address table.

Linking Archived

We can also link against the archive library `libc.a`. To achieve this we must change the linker's library search order:

```
$ cc -o HelloWorld HelloWorld.o -Wl,-a,archive_shared
```

`-Wl` can be used to pass the attached comma separated options list to the linker. `-Wl,-a,archive_shared` passes "`-a archive_shared`" to the linker. `-a` changes the library search order. In our case we tell `ld` to search now first for the archive version of `libc`.

Executables

Dynamically Linked Executables

These executables are linked against shared libraries. Just like with objects, we can look at the symbol tables of programs and libraries with [`nm\(1\)`](#):

```
$ nm HelloWorld

On PA:
Symbols from HelloWorld:

Name           Value   Scope   Type      Subspace
:
main          | 10400|extern|entry  |
```

```

main           |      10400|extern|code   | $CODE$
:
printf         |          |undef  |code    |
printf         |      10384|uext  |stub    |
:

On IA:
Symbols from HelloWorld:

[Index]  Value          Size     Type   Bind  O Shndx  Name
:
[59]    |    67110992|      112|FUNC  |GLOB  |0|    .text|main
[42]    |    270842400|          0|FUNC  |WEAK  |0|    UNDEF|printf
$
```

`nm(1)` shows a lot of symbols in the executable. Most of them are from `crt0.o`. The interesting ones from the programmer's point of view are `main` and `printf`.

On PA, there are two entries for each symbol in shared libraries and programs. One describes the symbol itself, the other is the so called stub that provides access to the symbol from external. "extern|entry" and "extern|code" are typical for exported symbols, "undef |code" for imported ones. `printf` is an imported symbol.

On IA, there is only one entry for each symbol. It describes `main` as a global function that is contained in the executable the ".text" reference and `printf` as an undefined function symbol.

The [`chattr\(1\)`](#) command shows the library references of the program:

```

$ chattr HelloWorld

On PA:
HelloWorld:
    shared executable
    shared library dynamic path search:
        SHLIB_PATH      disabled second
        embedded path  disabled first Not Defined
    shared library list:
        dynamic   /usr/lib/libc.2
    shared library binding:
        deferred
    global hash table disabled
    plabel caching disabled
    global hash array size:1103
    global hash array nbuckets:3
    shared vtable support disabled
    static branch prediction disabled
    executable from stack: D (default)
    kernel assisted branch prediction enabled
```

```
lazy swap allocation disabled
text segment locking disabled
data segment locking disabled
third quadrant private data space disabled
fourth quadrant private data space disabled
data page size: D (default)
instruction page size: D (default)
nulptr references disabled

On IA:
HelloWorld:
    32-bit ELF executable
    shared library dynamic path search:
        LD_LIBRARY_PATH      enabled first
        SHLIB_PATH           enabled second
        embedded path        enabled third
/usr/lib/hpux32:/opt/langtools/lib/hpux32
    shared library list:
        libc.so.1
    shared library binding:
        deferred
    global hash table disabled
    global hash table size 1103
    shared library mapped private disabled
    runtime checks disabled
    shared library segment merging disabled
    shared vtable support disabled
    explicit unloading disabled
    linkage table protection disabled
    segments:
        index type      address      flags size
            8 text       04000000      z---c-      D (default)
            9 data       40010000      ---m--      D (default)
    executable from stack: D (default)
    kernel assisted branch prediction enabled
    lazy swap allocation for dynamic segments disabled
    nulptr dereferences trap disabled
    address space model: default
    caliper dynamic instrumentation disabled
```

We will concentrate only on the shared library list of the output. A few of the other things listed here will be discussed later. Our `HelloWorld` program only references `/usr/lib/libc.2` on PA and `/usr/lib/hpux32/libc.so.1` on IA. `printf` must be defined in `libc.2*`, otherwise the linker would have printed an error. To check if this is true, we can again use [nm\(1\)](#):

```
$ nm /usr/lib/libc.2

On PA:
Symbols from /usr/lib/libc.2:
```

```
Name           Value   Scope  Type      Subspace
:
printf        | 959288|extern|entry  |
printf        | 959328|extern|code   |$CODE$
:

On IA:
Symbols from /usr/lib/hpx32/libc.so.1:

[Index]  Value          Size     Type  Bind  O Shndx  Name
:
[3318]  | 268808160|     896|FUNC |WEAK |0|.text|fprintf
:
```

On PA, we find `printf` as exported symbol, just as we found `main` exported in `HelloWorld`.

On IA, we find `printf` as a `.text` symbol, just as we found `main` in `HelloWorld`.

Archive Bound Executables

The [nm\(1\)](#) and [chatr\(1\)](#) outputs for the archive bound program look different in many ways:

```
$ nm HelloWorld

On PA:
Symbols from HelloWorld:

Name           Value   Scope  Type      Subspace
:
main          | 13232|extern|entry  |$CODE$
:
printf        | 14096|extern|entry  |$CODE$
:
```

On PA, this time we see both `main` and `printf` defined in the program. This is because the linker extracted the archive `libc` and linked the appropriate objects directly to the program. Also, we see only one entry per symbol. This is because there is no need to access them from outside, so no stubs are required here.

On IA, this example does not work, as there is no archived version of `libc.so` that is provided, but for any function symbols that are imported, they should be of type “`.text`”.

On PA, consequently, the [chatr\(1\)](#) output will also look differently:

```
$ chatr HelloWorld
HelloWorld:
    shared executable
    static branch prediction disabled
    executable from stack: D (default)
    kernel assisted branch prediction enabled
    lazy swap allocation disabled
    text segment locking disabled
    data segment locking disabled
    third quadrant private data space disabled
    fourth quadrant private data space disabled
    data page size: D (default)
    instruction page size: D (default)
    nulptr references disabled
$
```

There are no library search paths and no library list, and a few other attributes related to shared libraries are also missing, because they're useless in an archive bound program.

Linking archive can be useful if a program should run on systems independently of the [runtime environment](#) installed there, or if it must be run very early in the boot process, where the `/usr` file system might not be mounted yet. All Unix commands in `/sbin` are linked archive for that reason. On the downside, such programs are unaffected by library updates which might bring bug fixes etc., unless they are re-linked. Also, several features of HP-UX are available only in shared libraries and cannot be used by these programs, such as:

- NLS support (localization, i18n)
- Dynamic name resolution services (DNS, NIS, LDAP etc.)*

* Dynamic username and group resolution still works if `pwgrd(1)` is running.

Program Execution

Besides the things the programmer has written, there are several other things that need to be done in a program. A program life cycle consists of the following steps:

- Load the Program
 - Load shared libraries
 - Resolve symbols
 - Execute `main()`
 - Terminate the program

Processes are not allowed to access system resources like files, devices or the network directly. Every process must interact with the kernel via system calls for such actions. Because each of the above steps needs interaction with the kernel, we can observe them by tracing the system calls of the new process with [tusc](#) (the example is from IA; the PA display will be essentially the same):

The trace of the archive version on PA is much shorter, since it does not load shared libraries:

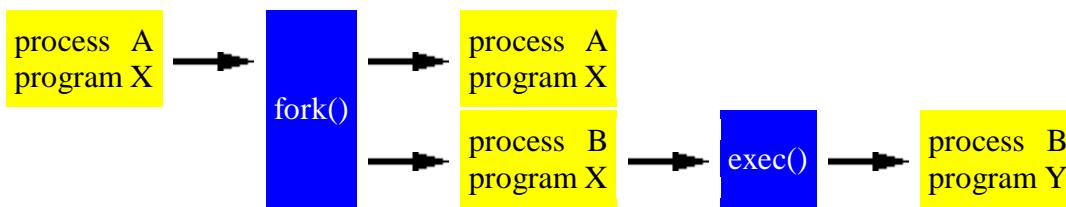
```
execve("./a.out", 0x7dff0a98, 0x7dff0aa0) ..... = 0 [32-bit]
mmap(NULL, 200, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -1,
NULL) = 0x7dfef000
sigsetreturn(0x135b8, 0x6211988, 1392) ..... = 0
sysconf(_SC_CPU_VERSION) ..... = 0x214
brk(0x40007998) ..... = 0
brk(0x4000998c) ..... = 0
brk(0x4000c000) ..... = 0
ioctl(1, TCGETA, 0x7dff0f30) ..... = 0
Hello World!
write(1, "Hello World! \n", 13) ..... = 13
exit(13) ..... WIFEXITED(13)
```

A system call trace does not show what happens in user space (what the program does internally). The internal program flow can be analyzed with a [debugger](#).

In this chapter we will also take a look at how processes access [memory](#).

Load the Program

Starting a program usually consists of two steps. First, a new process is created. This is called forking. The new process is called the child, the other one is the parent process. After forking, both processes are identical except of the process ID (PID), which must be unique on the whole system. In a second step the new program is loaded and executed in the child process, thus replacing the old one which was a copy of the program in the parent process. Afterwards the old program continues running in the parent process, and the new program running in the child process:



The `tusc` output shows:

```
execve("./HelloWorld", 0x87ffffffffffff5b0, 0x87ffffffffffff5c0) = 0 [32-bit]
```

`execve()` loads and starts our `HelloWorld` program, after `tusc` forked a new process. The `fork()` system call is not in the trace because it is executed inside of `tusc`. `execve()` takes 3 arguments, the path to the program file, the argument list and the environment.

A new program can also be executed without prior forking. In this case, the program that calls `execve()` to start the new program, will cease to exist afterwards.

Load Shared Libraries

Every shared library is loaded in the same way. It is mapped into memory using `mmap(2)`. To be more precise, several pieces of the shared library file are copied into memory. There are at least two pieces for each library:

- The code section is mapped shared, to allow multiple loadings of this library to use the already mapped instance, rather than to map it again, in the same or even another process.
- The data section is mapped private, which means each process that uses this shared library gets its own copy of the data part, to not conflict with other processes.

Sharing libraries between multiple processes saves a lot of memory, because the code contained in a shared library only needs to exist and to be loaded once, no matter how many processes load it.

Loading The Dynamic Loader

As mentioned in the [linking](#) section of the previous chapter, every executable starts with the same object, `crt0.o`. It contains the startup code that is necessary to invoke the dynamic loader, which is itself a shared library. The path to the dynamic loader is the same as the default path for shared libraries, and is hardcoded into `crt0.o`:

Executable Type	Path of the Dynamic Loader
PA-RISC 32-bit	/usr/lib/dld.sl
PA-RISC 64-bit	/usr/lib/pa20_64/dld.sl
IA64 32-bit	/usr/lib/hpux32/dld.so
IA64 64-bit	/usr/lib/hpux64/dld.so

The code in `crt0.o` loads it and passes control to it:

Loading Libraries

The dynamic loader reads the shared library list of the executable and loads them one by one. Each library can have a shared library list itself, so the dynamic loader checks if it has one and if yes, it loads the listed libraries recursively.

In the HelloWorld program did so loads the following libraries:

```
open("/usr/lib/hpux32/libc.so.1", O_RDONLY, 0) ..... = 3
:
open("/usr/lib/hpux32/libdl.so.1", O_RDONLY, 0) ..... = 3
:
```

`libc.so.1` and `libdl.so.1` are loaded, although `HelloWorld` only has `libc.so.1` in its [shared library list](#). `libdl.so.1` is loaded because it is listed in the shared library list of `libc.2`:

```
$ chatr /usr/lib/hpux32/libc.so.1
:
shared library list:
libdl.so.1
:
```

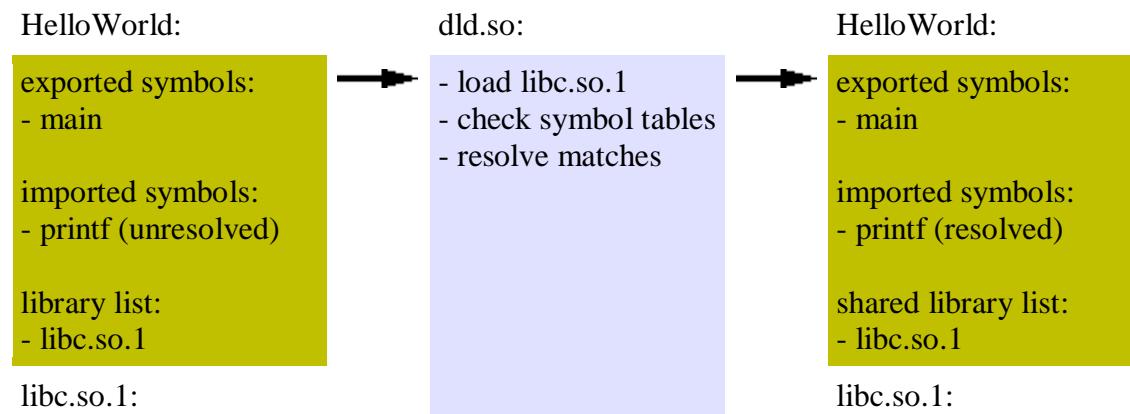
The [ldd\(1\)](#) tool can be used to list all libraries used by a program, without starting it, in the order they were loaded if the program was started.

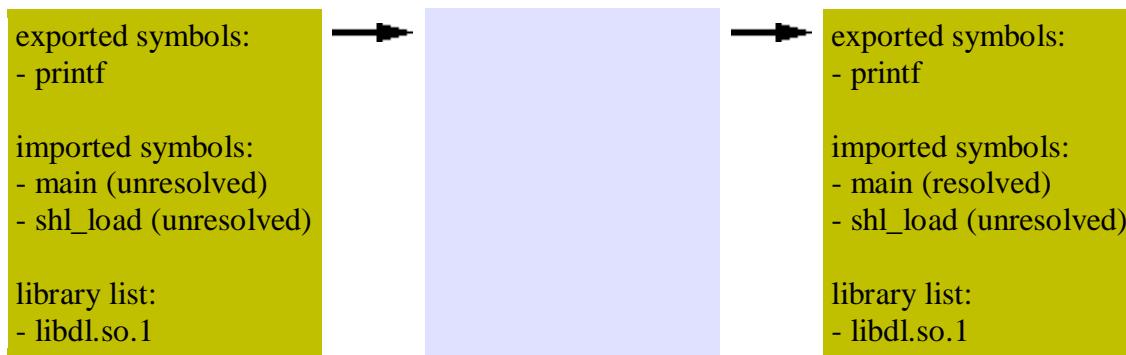
Resolve Symbols

Each time one of the libraries is loaded, all symbols that are still marked as undefined in any symbol import table (of the executable and all libraries loaded up to now) are checked against the exported symbols of the current library. And the symbol import table of the loaded library is checked against all other symbol export tables. References are bound to symbol addresses as matches are found.

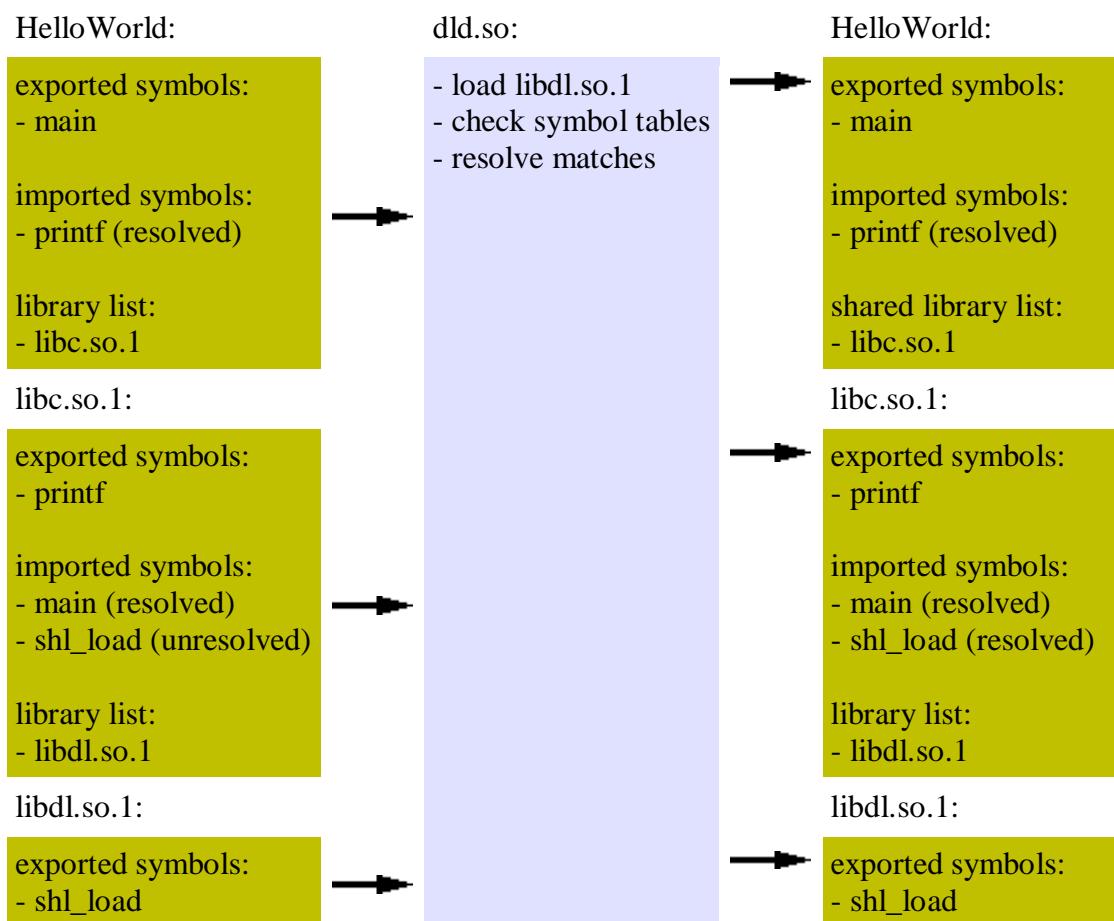
Symbol resolution is nothing but calculating addresses and modifying memory contents. No system calls are involved in that process. The `tusc` output provides no additional information besides showing which libraries are loaded. Symbol resolution in our `HelloWorld` program shall be explained with some diagrams and a few symbols.

When loading `HelloWorld`, it has an unresolved reference to `printf`, and `libc.so.1` is in its shared library list. The library is loaded, the symbol is found in it and can be resolved:





`libc.so.1` introduces new imported symbols, e.g. `main` and `shl_load`. The former can be resolved from `HelloWorld`, but the latter is not there yet. `libc.so.1` has `libdl.so.1` in its shared library list so this one is loaded too:



`libdl.so.1` contains the symbol `shl_load` so the reference from `libc.so.1` can be resolved now too.

If there were no more libraries to load, but there were still symbols that could not be bound to an address (could not be resolved), an error would be reported and the program would abort.

Execute main()

After the libraries are loaded and all symbols could be resolved, `main()` is called to execute the code the program was written for. `HelloWorld` was written to print the text "Hello World!". The required system calls are:

```
ioctl(1, TCGETA, 0x7dff0f30) ..... = 0
Hello World!
write(1, "H e l l o   W o r l d ! \n", 13) ..... = 13
```

In general, `ioctl()` inquires or changes the status of a file, and `write()` sends data to a file. On Unix, every input or output channel is treated as a file, and processes refer to files via file descriptors which are indices for the file table of the process. The first three entries are preset:

Number	Name	Description
0	stdin	The standard input channel
1	stdout	The standard output channel
2	stderr	The error output channel

So both system calls operate on `stdout`. The text appears on `stdout` before the system call is printed by `tusc`, because system calls are logged near the end of the call, when its return code is available.

Threads

In the classic view of process management, a process is an entity that executes program code and has its own private address space. As a consequence, if the program is waiting for some system resource, e.g. disk or network I/O, it can't do anything else. The process is sent to sleep by the system, and waken up again if the resource becomes available. Also, it cannot benefit from a multiprocessor system, because each process can only run on one CPU. To use multiple CPU's, multiple processes are needed.

To increase the performance of a single process, the threads model was introduced with HP-UX 10.X. Since then, a process is only a container. Program execution is done now by one or more threads. Each thread has its own instructions to execute and shares process resources with the other threads. Now, if one thread is waiting for system resources, another thread can be executed, and the process is not forced to be idle.

On HP-UX 10.X there were only user threads (also referred to as 1xN), which means that the kernel has only one thread per process, but the program can handle multiple threads, and it is up to the process to determine which of its threads should run. The kernel cannot distinguish between these threads because it only knows one thread, so there is no way for the scheduler to distribute the threads over multiple CPU's, which means that only one user thread could run at a time. User threads are also referred to as CMA threads, and the functions needed to make a program multithreaded are collected in the library `libcma.sl`.

HP-UX 11.0 introduced the kernel threads (Mx1), which means that the threads are handled by the kernel now, but each kernel thread can handle only one user thread. If a multithreaded process is running on a multiprocessor system, the kernel can schedule multiple threads on different CPU's simultaneously. Kernel threads in HP-UX 11.X conform to the POSIX 1003.1c standard. That's why the kernel threads are also known as POSIX threads, and the appropriate library for these threads is `libpthread.sl` on PA and `libpthread.so` on IA. On HP-UX 11.X the CMA threads are still available for downward compatibility reasons, but CMA threads are not compatible to POSIX threads and cannot be used together within one program.

There also exists the so called MxN threads model, which means that each kernel thread can handle multiple user threads. This model is supported since HP-UX 11.23.

Program Termination

The last action of a program is, of course, to terminate. From the programmers view, programs can terminate in several ways:

- Calling `exit()` causes the program to terminate regularly. The program's resources are freed and the process dies. An exit code can be passed to `exit()`. The exit code can be checked in the parent process, e.g. with the command "echo \$" in a [shell](#).
- Returning explicitly from `main()` has the same effect as calling `exit()`. The return statement can have one argument which will be used as the exit code.
- Reaching the end of a function is an implicit return. Reaching the end of `main()` will exit the program. In this case the exit code of the program is equal to the return code of the function that was executed last.
- Receiving a signal (see `signal(5)`) may terminate a process. Some signals cause the generation of a [core file](#), others don't. Signals can be sent by the kernel, by the process itself or by other processes. If the kernel sent a signal, it most likely detected a program failure, and the signal used by it will cause the generation of a core file. The core file can be [debugged](#) to find out why the program aborted.
- `exec()` also makes a program terminate, but the process stays alive, executing the new program.

Since some of the above are actually the same, there are only three different ways for a program to terminate:

- calling `exit()`
- calling `exec()`
- receiving a signal that causes abnormal termination.

If a process aborts with a coredump, the core file will always be written to the working directory of the process.

Although we do not call `exit()` explicitly in the [HelloWorld.c](#) source code, the system call `exit()` is called automatically after reaching the end of `main()`, to terminate the process normally:

```
exit(13) ..... WIFEXITED(13)
```

The exit code of our program is 13, which is the number of bytes written by the `write()` system call. `printf()` uses `write()` to print the text and forwards its return code to the calling function.

Process Memory

When [debugging](#) a process or core file, it is often necessary to understand and analyze how the process uses its address space, which contains all code and data owned by the process. This section will give an overview about HP-UX process memory management. For a detailed description see the "HP-UX Memory Management White Paper", Part I and II and the "HP-UX Process Management White Paper", Part I, II and III available at the ITRC.

Quadrants

Every 32-bit process can allocate up to $2^{32} = 4$ Gbyte of memory. This is the address space of the process. On HP-UX, this address space is divided into 4 quadrants with 1 GB and different purposes for each:

Quadrant	Address Range	Usage
Q1	0x00000000 – 0x3fffffff	text
Q2	0x40000000 – 0x7fffffff	private data
Q3	0x80000000 – 0xbfffffff	shared text/data
Q4	0xc0000000 – 0xffffffff	shared text/data

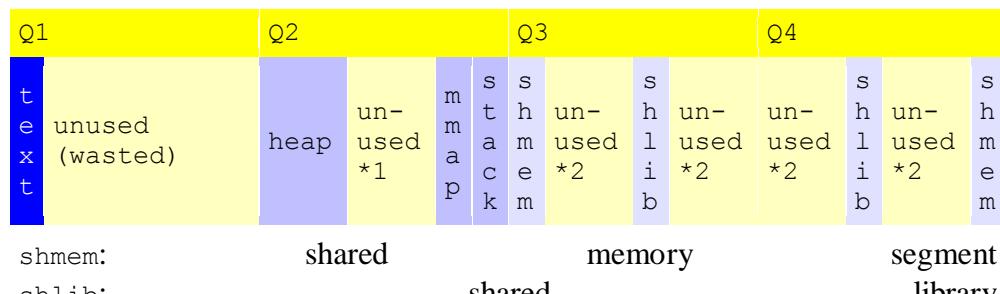
The program code is loaded to the start of Q1 (`0x0`). In terms of process memory, program code is also called text.

Usually no program has as much code to fill the whole quadrant, so it only uses the beginning of the quadrant, the rest of its address range will not be used. All data in the first quadrant is read-only. A process is not allowed to modify its code, because HP-UX will share one instance of the program code if multiple processes are running the same program.

Q2 holds the private data of the process. All addresses of program variables must lie within its address range. This sets an absolute limit at 1 GB for the maximum amount of private data a process can allocate.

Q3 and Q4 are used for shared text and data.

This is an example of address space usage of a process:



*1: can be used to increase the heap, or to map more private regions

*2: can be used for additional shared memory segments and shared libraries, or to map shared regions

The memory of 64-bit processes is also organized in 4 quadrants, but because of the larger address space, each quadrant has a size of 4 TB. Also, the quadrant usage is different:

Quadrant	Address Range	Usage
Q1	0x00000000 00000000 – 0x000003ff ffffffff	shared text/data
Q2	0x40000000 00000000 – 0x400003ff ffffffff	text
Q3	0x80000000 00000000 – 0x800003ff ffffffff	private data
Q4	0xc0000000 00000000 – 0xc00003ff ffffffff	shared text/data

Q1 is used for shared data here to allow sharing between 32-bit and 64-bit processes, using the same addresses.

Process Private Memory

A process has three methods to store private data. It can store it on the process heap, save it on the stack, or in privately mapped regions. All three methods use the second quadrant, as it is the only quadrant for private memory, but they follow different allocation rules. Every program has heap and stack, but it will only have mapped regions if it explicitly creates them.

The heap starts at the beginning of Q2. When executing a new program, the heap has a size of zero, and to store data on the heap, the program must request memory from the kernel via the `brk(2)` or `sbrk(2)` system calls. If a region of the heap is not needed anymore, it can be freed (deallocated) again. This will not return the memory to the kernel, it is only marked to be free for reuse within the program. Memory allocation on the heap is independent from the program flow. Allocation and deallocation can be done at any time by means of the libc functions `malloc(2)` and `free(2)`, which in turn use the `brk(2)` or `sbrk(2)` system calls.

The stack is a reserved area at the end of Q2. Storing data on the stack follows the "last-in first-out" scheme, just like a stack of paper. The last sheet you have put on the stack is on top and will be the first you get if you take one off. The stack has a fixed size which is reserved when the process is created. The program does not need to request memory from the kernel to store data on the stack. But it also cannot request more memory for the stack if it is full.

There is a special processor register, the stack pointer, which always contains the next free address on the top of the stack. Each time a function is called, memory for all local variables is reserved automatically by simply adding the required size to the stack pointer. In the same way the stack pointer is decreased when the program returns from the function, because the local data is not needed anymore.

Besides the local variables, the so called procedure context is also stored on the stack. The procedure context contains information about return addresses, function parameters, saved processor registers and several other important things to continue program execution when the current function returns. The whole program flow control information is stored on the stack.

Privately mapped regions can be allocated and deallocated in the address range between the upper end of the heap and the lower end of the stack via the libc system calls `mmap(2)` and `munmap(2)`. Per default, new regions will be mapped at the highest possible address. The memory allocated by a call to `mmap(2)` will be returned to the system after it has been unmapped with `munmap(2)`.

In a multithreaded program, each thread needs its own stack. The initial thread which was started on process creation, the so called primordial thread, uses the process stack. Whenever a new thread is started with `pthread_create(3T)`, a new private region is mapped with `mmap(2)` which is used as the thread stack. This area is unmapped when the thread exits.

Shared Memory

As the name implies, data in the shared quadrants is shared between processes. There are two types of shared memory on HP-UX, System V shared memory (`shmget(2)`, `shmat(2)` etc.) and shared mappings (`mmap(2)` with the `MAP_SHARED` attribute). Shared library text is made available to processes using shared mappings.

Limitations

There are various limits for memory allocation:

- `ulimit -d`
- kernel parameters
- quadrant size (process address space)
- virtual memory (swap space)

The kernel parameters that delimit the amount of private memory a process can allocate, are:

- `maxdsiz[_64bit]`: Maximum process heap size (not total private data)
- `maxssiz[_64bit]`: Maximum process stacksize

`maxdsiz[_64bit]` represents the maximum amount of private data a process can allocate on the heap using `malloc(2)`. It has no effect on the amount of data that can be allocated by `mmap(2)`. There is no limit for `mmap(2)` other than the private address space that is available to the process, and the total virtual memory available to the system.

Normally, programs don't need much space on the stack, so the default setting of 8 MB for `maxssiz` is sufficient in most cases. When increasing `maxssiz`, be aware that this amount is reserved for the stack and will reduce the space available for heap and privately mapped regions. If the maximum amount of allocatable memory in a program is much less than 1 GB, check your `maxssiz` setting.

The size of the process heap will also be limited by the [posix shell](#) built-in command `ulimit -d`. While `maxdsiz` is a system wide limit, `ulimit -d` only applies to the current process and all its descendants. The initial ulimit for the process heap is equal to `maxdsiz[_64bit]`. The shell will always display the ulimit for 32-bit processes, since it is a 32-bit process itself. But if the ulimit is set in a shell, the setting applies to both 32- and 64-bit processes started by this shell.

The amount of allocatable System V shared memory is not limited on a per process base, because it cannot be associated to a single process. Any process can create and attach to shared regions. There are however a few related kernel parameters:

- `shmmmax`: Maximum segment size
- `shmseg`: Maximum number of segments per process
- `shmmni`: Maximum system wide number of segments

No such limits exist for shared mappings, but of course there is still the address space limit. The third and fourth quadrants are dedicated to shared data and text, and the size of both quadrants, reduced by ca. 250 MB in Q4 which are reserved for the kernel, is the only limit. But this is a system wide limit. If the shared quadrants are full, they are full for all processes on the system.

Beyond The Limits

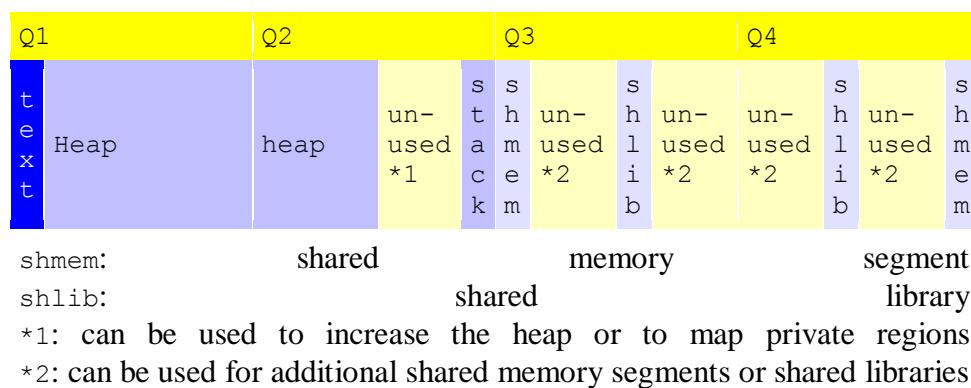
As shown above, the default limits for 32-bit processes are 1 GB of private and 2 GB of shared data. But programs might have different requirements, ie. they might need more private data and less shared memory, or the other way round. Of course, 64-bit programs have no problem with that, because the quadrants (Q3) have a size of 4 TB each. However, there might be reasons why an application cannot be built in 64-bit mode, and instead requires a more flexible way of using the full 4 GB of its 32-bit address space.

HP-UX provides means to change the way how the 4 quadrants can be used by a program, which shall be explained in the following paragraphs.

Allowing Private Data In Q1

At link time (and only then) you can select the usage of Q1. Per default, `ld` creates `SHARE_MAGIC` executables which make use of the quadrants as described above. They cannot allocate data in the first quadrant. The address space of this quadrant is lost.

When adding the linker option `-N` (ie. use `-wl, -N` in the `cc(1)` command that calls the linker), it will create an executable of type `EXEC_MAGIC`, which means that the first quadrant is used for both text and data. The heap will start immediately after the program code, thus allowing nearly 2 GB of private data. As a side effect, Q1 must now be writable and thus private, which means such executables cannot be shared anymore between multiple processes. With this change, the address space will look like this:



Allowing Private Data In Q3 and Q4

The `chattr(1)` command can be used to enable private data in the third and fourth quadrant:

```
chattr [ +q3p enable|disable ] [ +q4p enable|disable ] prog    ### PA-RISC
chattr +as mpas prog                                ### Itanium
```

This can be done independently of the Q1 usage, it is possible for both `SHARE_MAGIC` and `EXEC_MAGIC` executables. No relink is required for this change. This feature is called “Large Data Space”.

On HP-UX 11.0, several patches are required in order to enable the large data space feature. These patches are `PHKL_20222-PHKL_20229` or their successors. HP-UX 11.11 and later include this feature out of the box.

On PA-RISC, when specifying `enable` for the `+q3p` or `+q4p` option, the associated quadrant is made private. The output of the `chattr(1)` command displays how these quadrants will be used when the program is run:

```
$ chattr prog
:
third quadrant private data space disabled
fourth quadrant private data space disabled
:
```

Private quadrants can no longer be used to allocate shared memory. When changing both quadrants to be private, the process cannot request shared memory from the kernel anymore. It is still possible though to load shared libraries, but they will be loaded as private text and cannot be shared between processes anymore. Also note that it is not supported to use only Q4 for private data.

On Itanium, the `chattr(1)` command works slightly different. Instead of being allowed to change the usage of Q3 and Q4 separately, there is an additional address space model, MPAS (Mostly Private Address Space). This means all 4 quadrants can be used for private data, while certain areas, e.g. shared library text, can still be shared.

Using all possibilities together allows a process to use nearly the complete 4 GB address space for private memory:



T e x t	Heap	Heap	heap	s t a c k	Heap	un- used *1	s h l i b	s h l i b
------------------	------	------	------	-----------------------	------	-------------------	-----------------------	-----------------------

shlib: shared library

*1: can be used to increase the heap, for additional shared libraries or to map regions

Remember that `maxdsiz` and `ulimit` must be set high enough. Of course the amount of memory a process can allocate is also limited by the amount of virtual memory (swap space). So if there is not enough available, all these techniques are useless.

Allowing Shared Data in Q2

As mentioned earlier in this section, per default only Q3 and Q4 can hold shared memory segments. This limits the total amount of shared memory in the system to ca. 1.75 GB. This limit can be raised to 2.75 GB by changing Q2 from private to shared usage:

```
chatr -M prog      ##### PA-RISC
chatr +as shmem_magic prog ##### IA64
```

This change is only possible for `EXEC_MAGIC` executables, else there would be no quadrant left for private data. The address space for `SHMEM_MAGIC` executables will look like this:

Q1			Q2			Q3			Q4		
T e x t	heap	Un- used *1	s t a c k	un- used *2	s h m e m	un- used *2	s h m e m	un- used *2	s h l i b	un- used *2	s h l i b

shmem: shared memory segment

shlib: library

*1: can be used to increase the heap

*2: can be used for additional shared memory segments and shared libraries

Note that Q3 and Q4 cannot be private in a `SHMEM_MAGIC` executable. While [chatr\(1\)](#) will allow setting the appropriate flags without giving an error message, `execve(2)` will return `ENOEXEC` when trying to start the program.

Types Of Executables

The usage of the quadrants Q1 and Q2 define the type of an executable. The first line of the `chatr(1)` output on PA-RISC will show:

chatr(1) Label	Executable Type	Quadrant Usage
shared executable	SHARE_MAGIC	Q1 shared, text only, Q2 private
normal executable	EXEC_MAGIC	Q1 private, text+data, Q2 private
normal SHMEM_MAGIC executable	SHMEM_MAGIC	Q1 private, text+data, Q2 shared

On Itanium, the `chatr(1)` output contains a separate line that shows the address space model:

Address space model	Quadrant Usage
SHARE_MAGIC	Q1 shared, text only, Q2 private
EXEC_MAGIC	Q1 private, text+data, Q2 private
SHMEM_MAGIC	Q1 private, text+data, Q2 shared
MPAS	Q1 private, text+data, Q2-Q4 mostly private, shared regions possible

Products On HP-UX

There are a lot of different high level programming languages. Most of them are grown historically and have a focus on certain types of applications. The following list shows the most commonly used languages and their focus:

Language	Focus	URL
C	system programming	http://www.hp.com/go/c
C++	object oriented, technical programming	http://www.hp.com/go/c++
Fortran	numerical calculations	http://www.hp.com/go/fortran
Java	server applications, platform independent	http://www.hp.com/go/java

On HP-UX there are development environments available for all these programming languages. A development environment consists of more than just the compiler. To build an application, [compiler and linker](#) are required, and to run it a [runtime environment](#) is needed. In addition there are [tools](#) for organizing the build process, development in teams and for analyzing objects, libraries and executables.

The basic development environments are a collection of command line tools, but professional software developers prefer to work with an integrated development environment (IDE). IDE's provide a graphical user interface and bundle the tools in one application and make the developer's life a bit easier. HP's Softbench is such an IDE, but it will not be discussed here.

While programming languages are good to create special purpose programs, it is not practical to execute a series of operating system commands with them. At this point the tools of choice are [scripting languages](#). Consider a simple Unix shell command like

```
$ ls -l | more
```

The effort of programming this in C or any other language would be immense compared to this single line.

The sections in this chapter will cover the following themes:

- [Development Tools](#)
- [Runtime Environments](#)
- [Java](#)
- [Scripting Languages](#)

Java is discussed separately because it is different from conventional languages in several ways.

Development Tools

HP Compilers

HP provides compilers for all commonly used programming languages. The following table lists them along with product number and description:

Language	Product Number		Frontend Command	Sourcefile Extensions
	s700	s800		
HP C/ANSI C	B9007AA	B9007AA	/opt/ansic/bin/cc	.c
HP aC++	B9007AA	B9007AA	/opt/aCC/bin/aCC	.c, .C, .cpp, .cxx
HP C/aC++	B9007AA, B9007AAEVAL		See above	see above
Fortran90	B3907DB	B3909DB	/opt/fortran90/bin/f90	.f, .F, .f90, .F90

These compilers are **not** shipped with the HP-UX core. They are separate products which must be purchased separately, are codeword protected and can be installed from the application CD's/DVD or downloaded from the [HP Software Depot \[5\]](#).

The HP C/aC++ bundle B9007AA contains both C and C++ compilers. Customers can obtain a free 60 day evaluation version for this bundle. To upgrade from the evaluation version to the full version, the evaluation version must be swremoved first and the full version must be installed afterwards.

The HP-UX core OS comes with the so called bundled C compiler `/usr/ccs/bin/cc`, which is often misunderstood of being capable compiling ANSI style C code. The bundled `cc` is needed for kernel builds and is not intended to be used for application programming. It can be used to compile K&R style programs only. K&R is an old C dialect which is not used anymore by today's programs. Customers who want to compile ANSI C programs on HP-UX need the C/ANSI C Developers Bundle.

All HP compilers use a common development environment, which is shipped in separate products of the compiler bundles:

Auxiliary-Opt
DebugPrg
WDB

Auxiliary Optimizer for HP Languages
Debugging Support Tools
HP Wildebeest (HP WDB) Debugger

WDB-GUI

GUI for the HP WDB Debugger

WDB is the preferred debugger on HP-UX, and is based on the Gnu debugger gdb.

In addition to the products above, the HP C/aC++ Developers Bundle contains the HP-UX Developer's Toolkit, which consists of the following products:

AudioDevKit
CDEDevKit
ImagingDevKit
X11MotifDevKit

HP-UX Audio Developer Kit
CDE Developer Kit
HP-UX Developer's Toolkit - Imaging
HP-UX Developer's Toolkit - X11, Motif

The HP-UX Developer's Toolkit is not shipped with the other compilers. Customers who do not use HP C, but only one of the other languages, can obtain the HP-UX Developer's Toolkit for free from the [HP Software Depot \[5\]](#). Its product number is B3394BA.

Upgrades for HP compilers come in two different forms, either as a patch, or as a full (base, application) release. Most compiler patches require a certain base release that came out prior to the patch. To upgrade to the latest version, it might be necessary to install the last base release plus the latest patch. A release history for all compilers can be found on their [product pages \[1\]](#).

Development toolkit upgrades always come as regular patches which can be obtained from the HP internal patch servers, or from the [ITRC \[6\]](#). They usually depend on a similar patch level of their [runtime environment](#).

The compiler base releases also contain a current version of the WDB debugger. WDB can also be installed and upgraded separately by downloading the latest version from its [product page \[1\]](#).

Other Compilers

Besides the HP compilers there are also non-HP compilers available for HP-UX. The probably most popular ones are from the Free Software Foundation (FSF), called the [GNU Compiler Collection \[11\]](#).

HP provides a free HP-UX port of the Gnu C and C++ compilers gcc and g++. They have their own [product page \[1\]](#) on the HP website.

Although provided at the HP website, the Gnu compilers are not supported!

The Gnu compilers can also be obtained from the HP-UX [porting center \[10\]](#), but the version from the HP website is to be preferred, as it usually works better.

Related Products

All compiler frontends (including the bundled cc) use the same linker (/usr/ccs/bin/ld) to create their executables. The linker is not part of any compiler. The same applies to many tools described in the [tools section](#). Linker and linker tools are installed with HP-UX in the filesets

```
OS-Core.C-KRN
OS-Core.C-MIN
OS-Core.CMDS-AUX
```

Upgrades for these filesets as well as for several other development related filesets are always released as patches.

Runtime Environments

Every program, no matter in which language is was written, needs a runtime environment. A runtime environment usually consists of one or more shared libraries and possibly other resources that are needed to execute these programs. The following table gives an overview to the runtime environments for each programming language:

Programming Language	Runtime Libraries	Shipped with
any	libc.so/.so	HP-UX core
Fortran	libcl.so/.so	HP-UX core
aC++	libCsup.so/.so libCsup_v2.so/.so libstd.so/.so libstd_v2.so/.so libstream.so/.so librwtool.so/.so	HP-UX core

libc.so/.so is the central library of every Unix OS. Every executable, except if it was completely [archive bound](#), needs it. That's why it is not part of a programming language's runtime environment. It belongs to the HP-UX runtime environment.

libcl.so/.so is the language support library. It provides features that are special to the different languages, e.g. fortran I/O, and other useful things like stack unwind functions. Since HP-UX 11iv2, this library has been split up, so that programs can link only the part that is really required, e.g. now there exists a separate libunwind.so which contains the stack unwind functions. Details about that can be found in the HP-UX 11iv2 release notes which are available at the HP website and on any system running HP-UX 11iv2 or later:

- <http://h20000.www2.hp.com/bc/docs/support/SupportManual/c01917247/c01917247.pdf>

Not listed here, but of course needed by any program that uses shared libraries, is `dld.sl/.so`. The dynamic loader is part of the [linker](#) filesets.

For each HP-UX shared library there normally exist an archive version too (with the extension `.a`). These libraries do not belong to runtime environments because they only can be linked when a program is built. Nevertheless they are shipped with the runtime patches.

Runtime environment upgrades always come as patches.

Java

Conventional programming languages provide source code compatibility. To make programs written in those languages work on different platforms they must be compiled on each platform. Java is implemented in a way to provide platform independency at runtime level. This means a java program can be executed on any platform without recompiling it. This "compile once, run anywhere" concept is implemented by using the java virtual machine (JVM). Instead of providing different versions of applications for each platform, java applications are only required in one version, but each platform needs its own JVM that runs the java application.

Compiling

When java code is compiled, it is not translated into machine code. Instead, the so called bytecode is generated which is optimized to be read and interpreted by the JVM. The java compiler command `javac` is used to translate java source code into bytecode:

```
javac HelloWorld.java
```

There are several conventions regarding the structure of java source code. Usually each source file contains the source code of only one class¹⁾ which has the same name as the source file. If `HelloWorld.java` is our source file, it contains a class named `HelloWorld`, and the above command will generate the bytecode and store it in the file `HelloWorld.class`. The java compiler has some built in intelligence similar to [make\(1\)](#). If it finds references to other classes, it checks if the appropriate class files exist and are up to date, and if not, it compiles them as well.

There is no linker involved when building a java program. A java application is represented only by a collection of class files which are often stored in java archives (jar files). To create and handle java archives, java provides the `jar` command whose usage is very similar to `tar(1)`. Note that jar archives are incompatible to tar archives.

¹⁾In simple words a class is a container for functions (methods) and variables (members) that are related to each other. For details get a good book on object oriented programming.

Executing

A java application must contain one class that provides a method (function) defined exactly as `public static void main(String[] args)`, which is invoked to start the application. To execute a java program, the JVM must be started with the class or jar archive that contains this method as a command line parameter:

```
java HelloWorld  
java -jar HelloWorld.jar
```

Note that the `.class` extension of the class file must be omitted, while the `.jar` extension is required. If the java application is provided as a jar archive, the JVM is able to open it and pick the class files from the archive when methods of these classes are called.

The JVM is responsible for interpreting the bytecode, but running in interpreted mode is extremely slow. To increase the performance of java applications, the JVM has a built-in native compiler (called hotspot compiler) which translates the bytecode from the class files into native machine code while the application is running. Once a method has been compiled, the JVM will call the compiled version instead of interpreting the bytecode again.

Products

Java comes either as a software development kit (JDK, SDK) or a runtime environment (JRE, RTE). The JDK includes the JRE. If a JDK is installed, there is no need to also install the JRE. At the time of this writing HP supports 3 major java releases:

Major Version	Installation Path
Java 2 Version 1.4	/opt/java1.4
Java 2 Version 1.5	/opt/java1.5
Java 2 Version 6	/opt/java6

These major releases are located in different directories, thus can be installed in parallel.

There are different subrelease lines for Java 1.4:

- 1.4.0, 1.4.1, 1.4.2

Versions 1.4.0 and 1.4.1 are obsolete. Customers should upgrade to the latest 1.4.2 version.

Since version 1.4, all HP JVMs contain a 32-bit as well as a 64-bit JVM. To invoke the 64-bit JVM, the java command line option `-d64` must be used.

- JDK and JRE are free. They can be installed from an application CD or downloaded from the java [product page \[1\]](#), which always holds the latest and previous releases.

Java upgrades are not available as patches. New versions can always be downloaded from the above website as they become available. There is however a list of HP-UX patches that are recommended or even required for java. These are listed at

- <http://ftp.hp.com/pub/softlib/hpuxjava-patchinfo/index.html>

Scripting Languages

Programs (scripts) written in scripting languages don't need to be compiled before they can be executed. Instead they are passed to an interpreter which translates them at runtime. This makes scripts easy to handle, but also slow. The interpreter can be seen as the runtime environment for a script.

Shells

The most commonly used script interpreters in a Unix environment are the shells. They are part of the basic functionality of every Unix platform. Not only the user's interaction with the system is managed via a shell, also the `exec(2)` libc functions/system calls rely on a shell to execute programs they do not recognize. The shell provides the environment for using commands, doing arithmetical operations, using an editor (e.g. `vi`) etc. Some nice features are often built-in like history and file name expansion.

The following shells come with HP-UX:

Shell	Program Path
POSIX	/usr/bin/sh, /sbin/sh
Korn	/usr/bin/ksh
Restricted	/usr/bin/rsh, /usr/bin/rksh
C	/usr/bin/csh
Key	/usr/bin/keysh

Both the Posix-Shell and the Kornshell are based on the Bourne-Shell which meanwhile is obsolete. The Kornshell has extended the features of the Bourne-Shell, the Posix-Shell is very similar to the Kornshell and was introduced to create an official standard for UNIX shells.

Other shells, e.g. `bash` which is widely used under Linux, can be downloaded from the HP-UX [porting center \[10\]](#) websites, but are unsupported by the Response Center.

Shell updates are always available as patches.

All shells know three operating modes. When calling a shell with an argument, it enters the **automatic** mode, which expects the argument to be a script. The shell interprets the script and if the script exits, the shell exits too. If no argument is given to the shell, it is started in **interactive** mode where it prints a command prompt and waits for user input. If a shell is started with a leading "-", it is started as a **login shell**, which means that it reads its startup files before it enters the interactive mode. It is not possible to start a login shell from the command line. The leading "-" required for a login shell can only be inserted when using the `exec(2)` libc functions.

The shell which is started after login can be set to any shell available on the system by specifying it in the last field in `/etc/passwd`, e.g.

```
root:2eRyyWzYVvtzc:0:3:,,,:/sbin/sh
```

Default shell variables (environment variables) can be set system wide in `/etc/csh.login` (csh) resp. `/etc/profile` (other shells) and user specific in `~/.login` (csh) resp. `~/.profile` (other shells).

Common **shell features** are described in the following table:

Feature	POSIX	Korn	C	Key
Alias	✓	✓	✓	✓
Arithmetic Operations	✓	✓	✓	✓
Arrays	✓	✓	✓	✓
Autoload	✓	✓		✓
Command line editor	✓	✓	✓	✓
Command substitution	✓	✓	✓	✓
File name expansion	✓	✓	✓	✓
History	✓	✓	✓	✓
Job control	✓	✓	✓	✓
Variable prompt string	✓	✓	✓	✓

POSIX Shell

The POSIX.2 standard requires that, on a POSIX-compliant system, executing the command `sh` will start a POSIX compliant shell. The POSIX shell is in many respects very similar to the Korn shell, but has some additional enhancements.

If a command is invoked in the POSIX shell, it first searches for a function, then for built-in commands and at last for external commands.

There is a special version of the POSIX shell available on HP-UX, `/sbin/sh`. It differs from `/usr/bin/sh` in the way it was linked. `/sbin/sh` is completely archive bound. It is strongly recommended to use it as the login shell of the root user because it doesn't need shared libraries for execution. If e.g. the file system `/usr` cannot be mounted, `/sbin/sh` is the only shell that will still work.

To display special characters like German umlauts, the shell as well as the terminal that displays the shell output must be prepared for this. When using a terminal emulation like hpterm, dtterm or xterm, both terminal and shell must use the same character encoding. This can be achieved by setting the `LC_ALL` environment variable appropriately before starting the terminal, e.g.

```
# export LC_ALL=de_DE.roman8
```

Then the terminal emulation will use the correct font and the shell will use the correct character encoding. With real terminals, additional actions might be required to display special characters. See the terminal's manual.

In the statically linked `/sbin/sh` no umlauts can be displayed.

The POSIX shell is the default shell as of HP-UX 11i. All the HP-UX startup scripts use this shell. Its manual page is '`man sh-posix`'.

Korn Shell

The Korn shell is a superset of the Bourne-shell, featuring history mechanism and job control among other enhanced functions. Its functionality is very similar to the POSIX-shell.

Restricted Shell

This is in fact the POSIX shell, but when invoked as `rsh`, it will apply several restrictions to the user's capabilities, so that only a limited set of commands can be executed. `rsh` makes sense only if specified in `/etc/passwd` as the user's login shell, to prevent the user from escaping out of `rsh`. Details about the restrictions of `rsh` can be found in the manpage of `sh-posix(1)` in the section "rsh Restrictions". There is also a restricted version of the korn shell available with the name `rksh(1)`.

C Shell

The C shell allows C-style scripts. This does not mean it can interpret C programs, but the syntax of C shell commands and scripts is similar to C programs.

Key Shell

The Key shell is an extension of the standard Korn-shell. It uses hierarchical softkey menus and context-sensitive help to aid users in building command-lines, combining the power of the Korn-shell with the ease-of-use of a menu system. The Key-shell continually parses the command-line and always presents the user with an appropriate set of current choices on the softkey labels. At the bottom of the screen you can see the soft keys. Clicking on a button with the mouse or pressing the function key invokes the command combined with a help line saying what to do next.

Example: Pressing Change dir, then User dir invokes the following command with accompanied by a help line

```
# Change_dir user_dir  
Enter the name of the user whose home directory should be moved to.
```



Other Unix Interpreters

There are a few other native Unix tools which can be used to scan or modify ascii texts, e.g. `/usr/bin/awk` and `/usr/bin/sed`. Both have their own scripting language, but they do not have an interactive mode. Scripts can be passed to `awk` and `sed` either within the command line or in a separate file with the `-f` option.

PERL

`perl` (**p**ractical **e**xtraction and **r**eport **l**anguage) combines some of the features of C, `sed(1)`, `awk(1)` and Unix shells. `perl` is often used for generating dynamic web content. It is available for a wide variety of operating systems, e.g. all Unixes, Windows, MacOS, which gives `perl` scripts a similar platform independency as java.

For special purposes, e.g. database access, `perl` can be extended with various modules. A huge list of available `perl` modules can be found on the [CPAN \[12\]](#) website. Most of these modules consist of shared libraries, which are loaded when `perl` finds the appropriate instructions in the `perl` script. These modules are of course platform dependent and need to be built on any platform before they can be used. Additional modules that are downloaded from CPAN are not supported by the Response Center.

Current `perl` versions for HP-UX are available from the [HP Software Depot \[5\]](#).

The Interpreter Line

Scripts cannot be executed directly by the system. They require an interpreter to read and execute the commands contained in the script. The regular way to execute a script is to pass the script name as an argument to its interpreter, e.g.

```
$ /usr/bin/ksh myscript.sh
```

But if the script file has execute permission, it can also be started by typing just the name of the script:

```
$ ./myscript.sh
```

This is a feature of `exec(2)`. It tries to determine the type of the file via its `magic(4)` number. Every executable type has a specific magic number, and scripts can have one too. If the first two characters of the script file are "#!", `exec(2)` recognizes them as the magic number for a script and expects the interpreter name, one optional argument and a newline character to follow, e.g.

```
#! /usr/bin/ksh
```

This is the so called interpreter line. Instead of executing the script, the interpreter is executed and the script name is passed to the interpreter as an argument. Any executable can be specified in the interpreter line. You could e.g. use this feature to create a self-printing readme file:

```
$ cat readme.sh
#! /usr/bin/tail +2
Hello
$ ./readme.sh
Hello
$
```

If a script has no interpreter line, it cannot be recognized by `exec(2)`. Then it is assumed to be a shell script, and a POSIX shell will be started to execute it. In general, `exec(2)` will pass every ascii file that has execute permissions, but no valid magic number, to the POSIX shell. It is then up to the shell to try and interpret it, or to stop with an error message.

Tools

The tools to look at in this workshop can be divided into three categories:

1. [Development utilities](#)
2. [Tools for obtaining static information](#)
3. [Tools for runtime analysis](#)

In the Development Utilities section we will look at build process management and versioning tools.

Static information is considered to be properties of files in general. These tools look into binary files to get information about versions, required libraries, symbols etc.

Runtime analysis tools can obtain information from running processes like executed instructions, system calls and open files.

This section will only give a short view over the tools available. It cannot describe them in detail because some of the tools would require a separate workshop to discuss them in depth.

Development Utilities

[ar\(1\)](#)

Big applications usually are made of a large number of objects. To organize them, `ar(1)` can package them into archives which can be treated as libraries by the linker. Unlike `tar(1)`, `ar(1)` also stores a symbol table inside the library, that contains the symbols of all objects. At link time these libraries can be passed to `ld(1)` in the same way as shared libraries:

```
$ ar -r libmylib.a x.o y.o z.o
$ cc HelloWorld.o -lmylib
```

The `ar(1)` command creates an archive library and adds the specified objects to it. `cc(1)` passes `HelloWorld.o` and the option `-lmylib` to the linker which in turn extracts `libmylib.a` and links all required objects into the executable.

`ar(1)` belongs to the [linker tools](#).

make (1)

As described in [the Build Process](#), applications can consist of multiple executables and shared libraries. Each of them normally is built from multiple objects, and each object is built from multiple header and source files. If source files have changed, you could either rebuild the whole application, which could take very long, or you could just recompile and relink those parts that depend on the changed sources, thus saving a lot of time.

`make(1)` is designed to manage these dependencies. It compares existence and timestamps of files to determine which parts need to be rebuilt. The developer must create a so called makefile, which usually consists of macros and rules. Macros are used to set paths, command line options, file lists etc. that are used multiple times, and rules describe how to build files from others. The basic syntax of such a rule is as follows:

```
target:      dependents  
            commands
```

The **target** is the name of the file that is created by this rule. The **dependents** name all files that are required to build the target, and the **commands** are executed to build the target.

When processing a rule, `make(1)` checks if any **dependents** need to be built first. Therefore it looks for rules that have the **dependents** as **targets** and processes them recursively in a depth first algorithm. If there is no rule and the file exists it is treated as up to date. If there is no rule and the file does not exist, `make(1)` reports an error and stops. If one of the **dependents** is newer than the **target**, or if the **target** doesn't exist, the **target** is built by executing the **commands**. On the other hand, if all **dependents** are up to date and older than the **target**, `make(1)` assumes the existing **target** was already built from the current **dependents** and decides that there's no need to rebuild it.

If the makefile is called `[M|m]akefile`, `make(1)` can be called without arguments, otherwise the name of the makefile must be specified with the `-f` option:

```
make -f <makefile name> [ <target> ]
```

If `<target>` is omitted, the target specified in the first rule of the makefile is built. Otherwise the given target is built.

Consider the following example dependency tree of a C program:

```
myprog  ->  obj_a.o  ->  mydefs.h  
          ->  obj_a.c
```

```
-> obj_b.o -> mydefs.h  
-> obj_b.c
```

`myprog` is the main target. It must be built (linked) from the objects `obj_a.o` and `obj_b.o`. The objects themselves are targets too, they must be built (compiled) from source and header files. A simple makefile to handle these dependencies could look like this:

```
myprog:          obj_a.o obj_b.o  
                 cc -o myprog obj_a.o obj_b.o  
  
obj_a.o:        mydefs.h obj_a.c  
                 cc -c obj_a.c  
  
obj_b.o:        mydefs.h obj_b.c  
                 cc -c obj_b.c
```

`make(1)` belongs to the unix core commands.

Versioning Tools

When developing software it is important to keep track of changes made to sources over time, and to avoid conflicts if more than one person makes changes to the same source file. This is what versioning tools are for.

There are many different tools of this type available. The following table lists only a few:

SCCS	Source Code Control System
RCS	Revision Control System
CVS	Concurrent Version System

SCCS and RCS are part of HP-UX runtime, There also is an RCS from Gnu. This Gnu-based RCS version, as well as CVS, are available on the HP-UX Archiving and [porting center \[10\]](#). However, the Gnu-based RCS and CVS tools are not supported by the Response Center.

The basic principle of these tools is to provide source code in a repository. Each developer can check out files he wants to edit. That means he gets a local copy. When he has made changes to a file he wants to provide to the team he can check this file in again. The versioning tool will check then if someone else has made changes to this file too and request both to solve possible conflicts.

Static Information

file(1)

The `file(1)` command tries to determine the type of a file by checking `magic(4)` numbers. On ascii files it analyzes the first lines and tries to guess a programming language, if any.

If you have compiled a C program named `a.out` which was compiled from a source file named `x.c`, and which unfortunately aborted with a coredump, `file(1)` will give an output similar to the following:

```
$ file x.c x.o a.out core
On                                     PA:
x.c:      c program text
x.o:          PA-RISC2.0      relocatable      object
a.out:     PA-RISC2.0 shared executable dynamically linked -not stripped
core:      core file from 'a.out' - received SIGBUS

On IA:
x.c:      c program text
x.o:      ELF-32 relocatable object file - IA64
a.out:    ELF-32 executable object file - IA64
core:    ELF-32 core file - IA64 from 'a.out' - received SIGBUS
```

The output for the core file is especially interesting, because it provides information on what happened. It tells you:

- Which program generated the core file
- Why the core file was generated (which signal was received)

`file(1)` belongs to the unix core commands.

what(1)

`what(1)` can be used to find out the version of a file. It belongs to [SCCS](#). Every file shipped by HP that requires versioning contains a text that can be printed by the `what(1)` command. A what-string can be added to any file. It is simply a character string that starts with "`@(#)`" to be recognized by `what(1)`.

If you add the line

```
char ident[] = "@(#) x.c version 0.1 from Jan 04 2002";
```

to a C program source file, compile and run it, the what string will be in every file created from it, even in a core dump:

```
$ what x.c x.o a.out core
x.c:
    x.c version 0.1 from Jan 04 2002
x.o:
    x.c version 0.1 from Jan 04 2002
a.out:
    x.c version 0.1 from Jan 04 2002
core:
    x.c version 0.1 from Jan 04 2002
$ PATCH/11.00:PHCO_24723 Sep 13 2001 05:01:45 $
92453-07 dld dld dld.sl B.11.30 011005
```

The core file contains some more what strings than just ours. These come from the shared libraries used by the program, in this case `libc.sl` and `dld.sl` (or on IA `libc.so.1` and `dld` respectively). The library what strings returned from a core file are very helpful when the core file needs to be [debugged](#).

`chatr(1)`

The `chatr(1)` command can display and change internal attributes of a program or shared library. Most interesting at this point is that it prints the list of shared libraries that are loaded when the program is run. See the [Executables section](#) of the chapter "The Build Process" for a complete `chatr(1)` output.

`chatr(1)` can also change the way how libraries are searched when they are loaded. There are several possible resources where the dynamic loader expects paths to search for shared libraries:

- `SHLIB_PATH`
- `LD_LIBRARY_PATH` (not for PA-RISC 32-bit)
- embedded path
- default path

The paths will be searched in the above order.

`SHLIB_PATH` and `LD_LIBRARY_PATH` are environment variables that can be set at runtime, but using them must be enabled by a flag in the executable. The default is that they are disabled for 32-bit PA-RISC and enabled for all other types of executables. Enabling them can be done explicitly with the `+s` linker option. To disable them, use the `+noenvvar` linker option. Or you can use `chatr +s [enable|disable]` on the executable. `chatr(1)` will then show the search paths as follows:

```
$ chatr +s enable a.out
a.out:
    shared executable
    shared library dynamic path search:
```

```
SHLIB_PATH      enabled   second
embedded path  disabled  first   Not Defined
shared library list:
dynamic        /mylib/libmine.sl
dynamic        /usr/lib/libc.2
:
```

The embedded path list can only be defined at link time with the `+b` linker option, and is stored in the executable. The default path is where the library was found at link time. The linker stores it automatically in the executable.

On IA64, `dld.so` always uses the search path `/usr/lib/hpx[32|64]`, if libraries could not be found elsewhere, even if this path is not stored in the executable. The default path is no longer stored separately. It became part of the embedded path list instead:

```
$ chatr a.out
a.out:
 32-bit ELF executable
  shared library dynamic path search:
    LD_LIBRARY_PATH      enabled   first
    SHLIB_PATH           enabled   second
    embedded   path          enabled       third
  /tmp/mario:/usr/lib/hpx32:/opt/langtools/lib/hpx32
  shared library list:
    libmine.so
    libc.so.1
  :
```

Using `SHLIB_PATH` or `LD_LIBRARY_PATH` can be useful if a required library is located in a path not stored in the executable, or if you want to make an executable use a special version of a library. Put the desired library into a certain directory, put this path into `SHLIB_PATH`, and make sure `SHLIB_PATH` is enabled in the executable.

`chatr(1)` belongs to the [linker tools](#).

[`ldd\(1\)`](#)

While `chatr(1)` lists only the directly referenced libraries, `ldd(1)` displays all libraries that are loaded at program start, including the ones that are referenced by other loaded libraries. The `ldd(1)` output displays the order and path how the libraries would be loaded if the program was executed.

`ldd(1)` can be used to check if an `SHLIB_PATH` setting works. If `SHLIB_PATH` is enabled for `a.out` and is set to `/tmp`, where we put a special version of `libc`, the output would be:

```
$ ldd a.out
      =>
/usr/lib/libc.2 => /tmp/libc.2
/usr/lib/libldld.2 => /usr/lib/libldld.2
/usr/lib/libc.2 => /tmp/libc.2
```

`ldd(1)` belongs to the [linker tools](#).

[`nm\(1\)`](#)

This command displays symbol tables of objects, archives, executables and shared libraries. The output shows if a symbol is contained in the file or if it is referenced. See the [objects](#) and [executables](#) in the build section for `nm(1)` outputs.

`nm(1)` is helpful when searching for unsatisfied symbols (reported at link time) or unresolved symbols (reported at runtime). You could use a script like the following that makes use of `nm(1)` to search all libraries in a directory and report occurrences of the symbol:

```
for i in *.sl *.a; do
  nm $i | grep $1 | grep -e CODE -e DATA
  if [ $? -eq 0 ]
  then
    echo "from $i"
  fi
done
```

`nm(1)` belongs to the [linker tools](#).

There is a special version of this command that can display the [unmangled names](#) of C++ functions, `nm++(1)`. This command, which in reality is a wrapper script for `nm(1)`, belongs to the [aC++ compiler](#).

[`odump\(1\) / elfdump\(1\)`](#)

`odump(1)` can display a lot of information about 32-bit PA-RISC objects, archives, executables and shared libraries. `odump -help` prints all available options and gives a short description. The following table lists a few of them:

Option	Description
<code>-compunit</code>	Prints a paragraph for each object, can be used to find out which compiler and which flags were used to compile.
<code>-symbols</code>	Prints all symbols and symbol references. This is similar to <code>nm(1)</code> , but it still works even if the file was stripped (see <code>strip(1)</code>).

-verifyall	Executes a series of tests on the file. This can be used to verify the general structure of the file. It prints error messages if internal structures are corrupted.
------------	--

For detailed explanation of the `odump(1)` outputs, refer to the man page for `a.out(4)`. `odump(1)` can only be used with 32-bit PA-RISC objects. For 64-bit PA-RISC objects or IA64 objects `elfdump(1)` is an equivalent tool.

`odump(1)` and `elfdump(1)` belong to the [linker tools](#).

Runtime Analysis

One of Murphy's Laws says:

A program does not what you want, it does what you write!

The tools described on this page help you to analyze the behaviour of programs and bring what you write closer to what you want.

[gdb\(1\) / wdb\(1\)](#)

`gdb(1)` is the command line oriented Gnu debugger, `wdb(1)` is a graphical user interface, which allows more comfortable debugging. `wdb(1)` will not be discussed here.

`gdb(1)` is free and is shipped with all HP compilers. It can also be downloaded from the [HP WDB product page \[1\]](#).

The purpose of a debugger is to show what is going on in a running program, or to show the state of an aborted program which has been saved to a core file. You can:

- Start a program under control of the debugger or bring an already running process under control by attaching to it.
- Stop program execution on specified conditions or at a specified point of the program code.
- Execute program instructions step by step.
- Examine and change data when the program is stopped.
- Examine the data of a program that aborted with a core file.

`gdb` can be started as follows:

```
gdb program [ core | PID ]
```

It will print a startup message load the specified program. If a core file name is added, it will be loaded too, and the location of the program abort is printed. When adding a process ID, `gdb(1)` will attach to this process, stop it and print the location where it was stopped. Then the `(gdb)` prompt is printed to wait for user input.

When debugging a core file, it is a good practice to give the core file another name than “core”. Else, whenever some other process aborts with a core, the core file to debug is lost.

Programs should be built with the `-g` compiler option if there is a need for debugging. This causes the compilers to add source line and file references to the objects that make debugging much easier. The debugger will then load the source files and show the appropriate code. If no debug information is in the executable, there is no way to find out which machine instruction belongs to which line of the source code. While this makes debugging harder, there is still a lot of helpful information available to track down a problem.

It becomes even more difficult if the program has been stripped (see man `strip(1)`). This will remove even symbol information, so an address in the program cannot be associated to a function name anymore, thus making debugging nearly impossible.

A list of the most common debugger commands can be found in the man page. `gdb(1)` also provides an online help facility. Simply type `help` at the `(gdb)` prompt and follow the instructions. Further documentation including a tutorial can be found on the [HP WDB website \[1\]](#).

Debugging is a very complex work and cannot be discussed in too much detail here. However, a few simple things shall be explained. Let's debug our [HelloWorld](#) program (don't forget to recompile/relink with `-g`):

```
$ gdb HelloWorld
Copyright 1986 - 2001 Free Software Foundation, Inc. Hewlett-Packard
Wildebeest 3.0.01 (based on GDB) is covered by the GNU General Public
License. Type "show copying" to see the conditions to change it and/or
distribute copies. Type "show warranty" for warranty/support.

.
(gdb) list
1      #include <stdio.h>
2
3      int main() {
4          printf("Hello World!\n");
5      }
(gdb) break main
Breakpoint 1 at 0x2754: file HelloWorld.c, line 4.
(gdb) run
```

```
Starting program: /var/tmp/HelloWorld

Breakpoint 1, main () at HelloWorld.c:4
4          printf("Hello World!\n");
(gdb)
```

When invoking `gdb`, it loads the program but does not execute it yet. First we print out the source program to check where we are. Then we tell `gdb` to stop execution when entering `main()` by setting a breakpoint there, and run the program. `gdb` will stop execution at the first executable statement of `main()`. When `gdb` stops execution, it always prints the statement to be executed next.

```
(gdb) step
Hello World!
5      }
(gdb) continue
Continuing.

Program exited with code 013.
(gdb)
```

Now we make a step to the next line of source code. If we had the sources of `printf(3S)` and a debuggable `libc`, the `step` command would jump into `printf(3S)` and display the first executable statement of it. But we haven't, so we find ourselves at the end of `main()` after returning from `printf(3S)`. The program has not exited yet, so we tell `gdb` to continue program execution. If there was another breakpoint it would stop there, but there is none so the program exits. Finally `gdb` prints the exit code of the program.

Many debugger commands can be abbreviated. Pressing <enter> on an empty prompt repeats the last command.

Let's set another breakpoint in `printf(3S)`:

```
(gdb) break printf
Breakpoint 2 at 0x7f7a5360
(gdb) r
warning: Temporarily disabling shared library breakpoints:
warning: breakpoint #2
Starting program: /var/tmp/HelloWorld

Breakpoint 1, main () at HelloWorld.c:4
4          printf("Hello World!\n");
(gdb) c
Continuing.
```

```
Breakpoint 2, 0x7f7a5378 in printf () from /usr/lib/libc.2
(gdb)
```

Remember that `printf(3S)` is in `libc` which has no debug info. That's why `gdb` can only determine the code address of the breakpoint, and not the source file and line number. When re-running the program, it stops again in `main()` because `gdb` keeps breakpoints until they are unset, another program is loaded or `gdb` exits. After continuing, `gdb` stops the program at the first instruction of `printf()`. Although we have no sources for this function we could debug it, but only at assembly level, which is beyond the scope of this workshop. We could do a `disas` now to take a look at the assembly instructions of `printf()`.

If the program has stopped somewhere and we want to know the current location, the procedure backtrace gives an orientation. If the program is multithreaded, make sure to get the stack trace from the correct thread. Therefore, before getting a stacktrace, always verify which thread you look at, and switch to the right one if necessary:

```
(gdb) info threads
* 1 system thread 3785 0x7f7a5378 in printf () from /usr/lib/libc.2
(gdb) thread 1
[Current thread is already 1 (system thread 3785)]
(gdb) bt
#0 0x7f7a5378 in printf () from /usr/lib/libc.2
#1 0x2768 in main () at HelloWorld.c:4
(gdb)
```

In the above case we have only one thread. If there are multiple threads, the "info threads" command lists them all and marks the active thread with an asterisk.

Each line of the trace represents a procedure frame, which tells you the current address within the function, and in which part of the process it is. With debug information it also prints source file and line number. The address in frame #0 is the location where the program stopped, or in case of a core file, where it aborted. For every other frame it shows the address of the instruction to be executed next, if the program returned to this frame.

The backtrace from the `HelloWorld` program reads as follows:

- Frame #0: The program stopped at address `0x7f7a5378` in `printf(3S)`, which is in `/usr/lib/libc.2`.
- Frame #1: `printf(3S)` has been called from `main()` on line 4 in `HelloWorld.c`. When returning from `printf(3S)`, the program would continue with function `main()` at address `0x2768`.

```
(gdb) c
Continuing.
Hello World!

Program exited with code 013.
warning: Temporarily disabling shared library breakpoints:
warning: breakpoint #2
(gdb) quit
$
```

After continuing, the program it exits as before. We leave `gdb` by typing `quit` or simply `q`.

When debugging a program that died on another system, it is not sufficient to take program and core file and start debugging. You must make sure that the debugger uses the same versions of the shared libraries as at program runtime. When using the wrong library versions, the debugger will print incorrect information.

To find out which libraries were used and in which version, you can either login there and use `ldd` and check their versions with `what`, or you can check the `what` string of the core file. It contains the `what` strings of all used libraries, but it can be difficult to determine which `what` string belongs to which library.

All these libraries should be available on your system. You can either put them into their original locations which normally means that you must remove the currently used versions and possibly downgrade your runtime libraries, or you can put them into a dedicated directory and tell `gdb(1)` to pick the libs from there. The environment variable `GDB_SHLIB_PATH` is used for that purpose.

If possible, debug a core file on the system where it was generated. If all else fails, newer `gdb` versions (5.4 and higher) know the `packcore` and `unpackcore` commands. Use `packcore` on the system where the core file was written, to create a `packcore.tar.Z` (default output name) file with the executable and all shared libs in use by the process. Copy the `packcore.tar.Z` file to the system where you want to debug, then use `unpackcore` before debugging, to extract the files.

[coreadm\(1M\)](#)

There are various reasons that might cause difficulties with the creation of core files, such as lack of disk space, permission problems etc. Many problems result from the fact that per default all core files are stored as `$PWD/core`. There is no way to define where and under which name the core file should be stored.

To overcome this limitation, HP-UX 11.31 introduces the command `coreadm(1M)`. It allows setting various core file related attributes like path and exact name, including process specific info. These settings can be applied globally for all processes, or on a per-user or per-process basis.

For example if core files should be stored in a separate file system that provides sufficient disk space, and their names should contain program name and PID of the aborted process, the following command can be used (as root):

```
# coreadm -e global -g /core files/core.%f.%p
```

tusc

While a debugger is helpful to analyze what happens inside a running program, it often is interesting to see how a program interacts with the kernel. Every process does this via system calls. System calls are necessary for reading and writing files, communicating with other processes or over the network, allocating memory etc.

`tusc` is the tool to trace system calls on HP-UX. It is not installed with HP-UX, and it is not a supported product. It can be obtained HP internally (including a manpage) from the [KTOOLS website \[8\]](#) or from the [HP-UX Archiving_&_porting center \[10\]](#).

System calls are defined within the kernel sources. They are identical for all processes. That's why no source code or debug information of a program is required to trace its system calls.

In the [execution example](#) we have already traced the `HelloWorld` program. Similar to a debugger, a program can either be run completely under `tusc` by specifying the program name, or you can attach to an already running process by specifying the process ID:

```
tusc [ options ] program | PID
```

There are various options available to collect additional information like process and thread ID's, timestamps, or to redirect output into a file rather than `stdout`, etc. To see all available parameters, start `tusc` without any options.

`tusc` prints one line for each system call. The line contains at least the name, arguments and return value of the system call. `tusc` tries to interpret the arguments as far as it knows about them. To get info about what the system call does you could look up the manpage of the system call name. Note that the manpages are not for the system calls but for `libc` functions which are wrappers for the system calls with the same name. So the arguments and the return code of the system call might not match the things described in the manpage. Fortunately, for most system calls they do match.

```
open("/tmp/xyz", O_RDWR, 04344) ..... = 3
write(3, "H e l l o   W o r l d ! \n", 13) .... = 13
```

The `open()` system call shows that the process opened the file `"/tmp/xyz"` for reading and writing. The third argument is undocumented in the manpage. This is such a difference between the system call and the `libc` function. The return value 3 is a reference number to be used for subsequent system calls to access this file. This number is called the file descriptor. `write()` used this file descriptor and sent 13 bytes to the file. Note that `tusc` prints two digits for each byte of an input or output buffer to provide monospaced output for printable characters (character + blank) and special characters (symbol or hexadecimal value). The return code 13 confirms that 13 bytes have been written.

An interesting option here might be `-s <namelist>` to make `tusc` filter the system calls and display only those specified in the comma separated name list:

```
$ tusc -s open,write,read,close program|PID
```

This `tusc` invocation will only print the listed system calls. This combination might be useful to see the file access of a process. Alternatively, system calls can be excluded from the trace. This can ease the trace analysis if a process does certain system calls over and over again, but which might not be of interest:

```
$ tusc -s ! sigprocmask program|PID
```

[kmeminfo](#)

`kmeminfo` can provide information about the memory usage of either the complete system or a running process. The latter can be very helpful when analysing memory related problems, because it shows the usage of the process address space. Here is a typical output for the process memory:

```
$ kmeminfo -pid 3523
:
Process's memory regions (in pages):

Process "java", pid 3523, 32bit, R_EXEC_MAGIC:

  type      space          vaddr    ref    virt    phys    swap
  TEXT 0xd22f400.0x0000000000001000    1      5      4      1
  DATA 0xd22f400.0x0000000000006000    1    2042    2041   2048
  MMAP 0xd22f400.0x00000000679ff000    1    513    257    515
  MMAP 0xd22f400.0x0000000067c00000    1   24576   1281   1344
  MMAP 0xd22f400.0x000000006dc00000    1    8192     86   107
  MMAP 0xd22f400.0x000000006fcbd000    1      33     18    34
  MMAP 0xd22f400.0x000000006fcde000    1      33     18    34
  :
```

```

MMAP 0xd22f400.0x000000006ffe6000      1       4       3       5
MMAP 0xd22f400.0x000000006ffea000      1       2       2       3
SHLDATA 0xd22f400.0x000000006ffec000     1       4       4       5
STACK 0xd22f400.0x000000006fff0000     1      528      272      530
SHLTEXT 0x7082400.0x00000000c0004000    109      3       3       1
SHLTEXT 0x7082400.0x00000000c0010000   117      33      33       1
:

```

`kmeminfo` prints one line for each memory region. Most interesting are columns 1, 3 and 5. Column 1 tells what the region is used for. The abbreviations are self-explaining. Column 3 lists the start address and column 5 its size in pages (one page = 4k).

`kmeminfo` reads the kernel memory so you need to be root to use it. Another way would be to set the set-user-ID permission flag of the root owned `kmeminfo` executable with `chmod(1)`. Detailed information on `kmeminfo` is available at

- <http://ktools.hp.com/~hpx/kmeminfo/>

This tool is not officially available but can be sent out to customers on request. It can be obtained from the [KTOOLS website \[8\]](#).

`adb(1)`

When analyzing a core file with `gdb`, `adb` can provide additional, helpful information. E.g. to find out if a pointer or address found in `gdb` is valid, and to which region it belongs, the `$m` command of `adb` can print the address map of the aborted program.

The easiest way to get an address map is to pipe the `$m` command into `adb`. Make sure to surround the `$m` with single quotes, else the shell would interpret it as shell variable and replace it with whatever is in its variable `$m`:

```

$ echo '$m' | adb a.out core
:
Data:          b = 40001000      e = 40008000      f = 56C
MMF:          b = 7AEE9000      e = 7AEEA000      f = 757C
MMF:          b = 7AEEA000      e = 7AEEB000      f = 858C
:
MMF:          b = 7B04E000      e = 7B050000      f = 16C60C
MMF:          b = 7B050000      e = 7B054000      f = 16E61C
Stack:        b = 7F7F0000      e = 7F7F8000      f = 17262C
:

```

For each region it prints its type, and the start and end address. This is similar to the output of `kmeminfo -pid`. `gdb`'s `info target` command also lists the address space, but in much more detail which makes it more difficult to read.

adb(1) is part of the OS-Core commands.

lsof

lsof is a tool that displays information about open file descriptors on a system, including pipes and networking connections, plus the loaded shared libraries. It queries the kernel for that information. To find out the files opened by a certain process you can either filter the output, e.g. using grep(1), or use the -p option. To see all processes that have file <file> open, run *lsof <file>*.

lsof can be downloaded from the the [KTOOLS website \[8\]](#) or the HP-Ux Archiving and [Porting center \[10\]](#).

Solving Common Problems

Software development is a very wide field, and there are not many common problems that have standard solutions. Unlike in other areas, the products described here do not require complex configurations nor are there general rules for their usage. There are however a few standard situations that allow a common approach.

This chapter is under continuous construction and therefore can not be viewed as complete. It can only try to give some general advice, and on the other hand, list some particular problems which are likely to reoccur:

- [Installation Problems](#)
- [Build Problems](#)
- [Problems Starting A Program](#)
- [Error Messages From The Dynamic Loader](#)
- [Program Aborts](#)

Also, there are a few [environment variables](#) that can be helpful when analyzing problems.

Many problems described here are caused by user errors. But there is always a chance that the problem is caused by internal errors or bugs in compilers, the linker or in runtime libraries. In such a case it might become necessary to involve the [WTEC team](#) (HP internal). There are different teams responsible for different issues:

- [Languages WTEC \[7\]](#): compilers, linker, java, debuggers or runtime libraries for programming languages
- [HP-UX WTEC \[7\]](#): libc and other HP-UX runtime libraries, shells and unix commands
- [X/Graphics WTEC \[7\]](#): X11/Motif/3D development and runtime libraries

Installation Problems

General Information

The products discussed here can be divided into three categories regarding their installation. The first category is installed with the HP-UX operating system:

- C, C++, Fortran runtime libraries
- Bundled C compiler
- Linker, linker tools and dynamic loader
- libc header files

Normally there are no known problems installing these products. Upgrading them is done with patches. Every patch can be installed on a base HP-UX installation, but might have dependencies.

The second category of products is also free but must be installed separately:

- Java
- Perl
- wdb/gdb debugger

The products can be downloaded from the [web](#). There are no patches available for these products. Upgrading them always requires installing a complete new version.

In the third category we find the products which must be purchased. These will be discussed in the next paragraph.

Compiler Installation/Upgrade

Installation can be done from the application CD/DVD, but a codeword is required. These products can also be downloaded via the Software Update Manager of the [HPSC \[6\]](#), if the customer has an appropriate contract.

Compiler upgrades are not always straight forward, because they don't have a unique base release to be upgraded to a new version with a patch. Instead, from time to time there will be new base releases, and inbetween patches are released that can only be installed on the most recent base version.

To find out what is the latest version of a compiler, always check their [release history](#). It also contains information about which base release is required for which patch.

There are several ways to find out the currently installed version of a compiler. `aCC -v`, `cc -v` (Version 6.0 or later) and `f90 +version` return the version number. The `-v` option to `aCC` and `cc` also prints the versions of the subprocesses invoked, but you must compile something to make them invoke any subprocess. Every compiler frontend has a `what` string that tells you the exact version. The third way is to look up the SD database with `swlist(1M)`. Note that the version numbering returned by `swlist(1M)` sometimes is different from the `what` strings:

```
$ uname -r
B.11.11
$ aCC -V
aCC: HP ANSI C++ B3910B A.03.27
$ what $(which aCC)
/opt/aCC/bin/aCC:
    HP aC++ B3910B A.03.27
    HP aC++ B3910B A.03.25 Language Support Library
```

```
$ /usr/sbin/swlist | grep -e B3911 -e B3913
B3911DB          C.03.27          HP aC++ Compiler (S700)
$
```

At this 11.11 system we have installed version A.03.27. At the time of this writing, the latest version for 11.11 is A.03.85. To upgrade to the latest version we need to install the A.03.85 base release from the September 2008 application CD/DVD.

To install a certain compiler version other than the latest (might sometimes be required by some third party software vendors), check again if it is available as patch or on CD. In case of a patch, check which base version it requires, and then install that one first.

However, older versions of the Application CDs are generally not available from HP Distribution and the customer must have access to an old archived version at the customer site in order to install older versions of the compiler base release.

Installing Multiple Versions of one Java Major Release

Java application vendors might support their application only with certain java versions. Many vendors ship the required JRE bundled with the application, and it will be installed in the application installation tree and loaded from there as needed. Other vendors don't, and expect the required version to be available on the system. This could lead to the requirement of having multiple versions of e.g. java 1.4.2 , 1.5 or 6 installed. Each major branch of java installs into a different directory. All versions of java 1.4.x.xx install into /opt/java1.4. All versions of java 1.5.x.xx install into /opt/java1.5. All versions of java 6 install into /opt/java6. If it is necessary to install multiple versions of 1.4, for example 1.4.2.10 and 1.4.2.15 this can be achieved as follows:

- Install one (if possible the older vendor required 1.4.2.10) version.
- Rename the installation path, one suggestion would be /opt/java1.4.2.10.
- Install the other newer version.
- Set \$JAVA_HOME appropriately before starting the application

If e.g. one application (app_1) needs version 1.4.2.10, and another one (app_2) needs 1.4.2.15, do the following:

```
# swinstall -s /path/to/rte14_14210_pa.depot '**'
# mv /opt/java1.4 /opt/java1.4.2.10
# swinstall -s /path/to/rte14_14215_pa.depot '**'

For app_1:
$ JAVA_HOME=/opt/java1.4.2.10
For app_2:
$ JAVA_HOME=/opt/java1.4
```

Build Problems

Normally every build process should succeed without any message from the compiler or the linker. Messages should never be ignored, even if the build process succeeds, because they could lead to runtime problems.

Solving most of the build problems means to understand compiler and linker messages and to take appropriate actions like code changes, different compiler or linker options etc. to remove the reasons for the messages. As a basic rule, the first step to resolve such an issue is to look up man pages, programmer's guides and release notes. There is a good chance that one of these sources contains an explanation for the situation encountered.

While this sounds quite easy, it can in fact become very complicated, because the pure mass of information we get from there can make it difficult to find what we're looking for.

Compiler and linker distinguish between warnings and errors. Warnings are issued to tell the developer about potential problems, but the build process will not be aborted. When an error occurs, the build process will not complete, but it will also not stop after the first error. Instead it will continue as far as possible to report subsequent errors, if any.

When fixing coding errors reported by the compiler, always start with the first message because the others might be follow-ups from the first; e.g. a missing ending bracket, parentheses and/or quote character in a C program can cause tons of error messages, and those errors may all disappear when the missing character is added. Or, a missing header file can cause lot of errors about undefined symbols. If you're not familiar with error messages, always fix only the first one and recompile to see which errors remain.

This section will discuss some commonly seen errors and warnings, explain their meaning and what corrections are required.

make(1) Errors

Make: Must be a separator on rules line 2. Stop.

This error is caused by a syntax error in the makefile. All commands in a rule must be preceded by a tab character. Multiple spaces might look the same, but `make(1)` does not accept them. Check the given line and replace leading blanks by one or more tabs. A common root cause for this syntax error is using copy&paste on makefiles. The copying might expand tabs to spaces.

Make: Don't know how to make xxx. Stop.

In the makefile "xxx" was specified as a dependent, but neither does the file exist nor was there a rule to build it. The writer of the makefile must check if this file is really needed to build the targets. If not, remove the file from your dependents. If yes, get a copy for make(1) to find or find out how to create it and add a rule to the makefile to create it.

PA2.0 Warning

```
/usr/ccs/bin/ld: (Warning) At least one PA 2.0 object file (x.o) was  
detected. The linked output may not run on a PA 1.x system.
```

This is a warning issued by older linker versions. It says that the linker has built an executable for PA2.0 (PA8000) CPU's which cannot be run on old PA1.1 (PA7000) systems. Trying to execute such a program on a PA1.1 system will result in an [error message](#).

This message usually occurs when building on PA2.0 systems, because the compilers check the hardware to determine what code to generate. Generating PA2.0 programs is the default on such systems. To create portable code that can also be run on PA1.1 systems, the compiler option `+DAportable` must be used for ALL objects loaded into the executable.

PA1.1 systems have become rare these days, so most of the time this warning can safely be ignored.

Unsatisfied Symbols

```
On PA:  
/usr/ccs/bin/ld: Unsatisfied symbols:  
  a (first referenced in x.o) (code)  
  b (first referenced in x.o) (data)
```

```
On IA:  
/usr/ccs/bin/ld -o a.out -u__exit -umain -x.o -lc  
ld: Unsatisfied symbol "a" in file x.o  
ld: Unsatisfied symbol "b" in file x.o  
2 errors.
```

The linker reports that it found references to one or more symbols, but the symbols could not be found in any of the objects and libraries. The linker will create the binary file, but will not give execute permissions to it because the symbol table is incomplete.

In the above case there was a reference to symbol `a`, type `code`, which means it is a function, and there was a reference to symbol `b`, type `data`, which means it must be a global variable. And we see that both symbols were referenced within the object `x.o`.

Unsatisfied symbol problems can have multiple reasons, and their resolution can become very tricky:

- **The symbol is indeed missing:** You checked all objects and libraries with [nm\(1\)](#) but there were only references (undef). Check if there is a library or an object missing in the link command. Use [nm\(1\)](#) to find it. To find out if a symbol is in an HP-UX runtime library, you could also check if there is a man page available for it. If there is one, it sometimes tells you which library must be used. If the man page does not tell you, search all libraries under `/usr/lib`.
- **The reference is wrong:** If the symbol is referenced by your own code, check if there is a typo. Most compilers would report an undefined symbol error or warning in that case, but e.g. the `cc_bundled` doesn't care for undefined symbols, `cc` might issue a warning about a missing prototype.
- **Incorrect link order:** This can happen especially with archive libraries. The linker processes the objects and libraries in the order they are listed at the command line. If you have an object in an archive lib, then the linker will not load it if there was no reference to any of its symbols at that time. Later references will then lead to unsatisfied symbol problems. Make sure that there is at least one reference to a symbol of that object, before it is loaded.
- **Version conflict:** Often runtime libraries depend on each other, and if a change is introduced to one by applying a patch, another one might need to be patched too. So it sometimes could also help to check for new versions of system libraries like `libc`, `libcl`, aC++ runtime libraries etc.

Unsatisfied Symbols (Special Cases)

While the previous paragraph gave a general approach to find reasons for unsatisfied symbols, there are several common cases where the general approach would be too complicated.

```
/usr/ccs/bin/ld: Unsatisfied symbols:
  _main (first referenced in x.o) (code)
```

If you find `_main` unsatisfied, this could be caused by aC++ objects that are not linked with `aCC`. It is strongly recommended to use `aCC` to call the linker, because aC++ programs need several additional linker options which are added automatically by `aCC`.

```
/usr/ccs/bin/ld: Unsatisfied symbols:
  std::ios_base::Init::Init() (first referenced in x.o) (code)
  std::ios_base::_C_fire_event(std::ios_base::event, bool) (first
referenced in x.o) (code)
  :
```

If the linker reports several unsatisfied symbols that start with `std::`, this is caused by aC++ code that was compiled with the `-AA` option of aC++. This switches on the ANSI compatible mode. If however `-AA` was not present at link time, aCC will link the program against the non-ANSI runtime libs. The solution here is to compile and link everything with the `-AA` option.

```
/usr/ccs/bin/ld: Unsatisfied symbols:  
Iostream_init::~Iostream_init() (first referenced in x.o) (code)  
Iostream_init::Iostream_init()%1 (first referenced in x.o) (code)  
:
```

This is the inverse case. There was at least one object compiled without `-AA`, but it was linked with this option. Again, make sure that either `-AA` is always used, or never. Mixing both is not supported and will not work, because different runtime libraries are needed and they are not compatible. Most probably changes to the source code become necessary when switching from non-ANSI to ANSI mode and vice versa.

```
/usr/ccs/bin/ld: Unsatisfied symbols:  
operator new(unsigned int,void*) (code)  
operator new(unsigned int) (code)
```

This can happen when linking an aC++ program on HP-UX 11.X with objects compiled on UX 10.20. It is not supported to link both 10.X and 11.X objects. To resolve this everything must be recompiled on 11.X.

Mismatched ABI

The following error messages have all the same root cause:

```
/usr/ccs/bin/ld: /usr/lib/pa20_64/libcl.sl: Mismatched ABI. 64-bit PA  
shared library found in 32-bit link.  
  
ld: Mismatched ABI (not an ELF file) for /somepath/libmylib.sl  
  
ld: Mismatched ABI (not an ELF file) for x.o  
  
/usr/ccs/bin/ld: x.o: Mismatched ABI. 64-bit PA object file found in  
32-bit link.
```

The linker finds both 32-bit and 64-bit objects or shared libraries while linking an executable. This is not supported and does not work. Make sure that the objects and shared libraries are either ALL 32-bit or ALL 64-bit. The linker picks the first object from the command line and uses its type to determine if the executable should be 32-bit or 64-bit. On the first object or shared library that is not of the same type, `ld` reports an error like above.

Similar messages will occur when mixing PA-RISC and Itanium objects and/or libraries. This is of course also not supported. Sometimes the root cause of this might be incorrect include paths, e.g. when migrating from PA-RISC to Itanium, but not changing include paths from the PA-RISC to the Itanium versions of libs to be linked.

Compiler/Linker Aborts

Besides the problems described above, which are typically user errors, it also can happen that a compilation/link aborts because of internal errors.

Compilers as well as the linker can abort if there is not enough memory available to build the program. Possible error messages that indicate such a problem are:

```
Error 8203: Exact position unknown; near ["/usr/include/errno.h", line
32]. #
memory exhausted at 966636 Kbytes; try increasing swap space or the
maxdsiz kernel parameter (8203)

cc: error 1405: "/usr/ccs/bin/ld" terminated abnormally with signal 10.
```

While the first message from aCC is very precise about the cause, the latter doesn't really say what's wrong. It could be a memory problem but also an internal linker problem. To verify that lack of memory is really the problem, you could start compiling/linking, watch the size of the processes with `top(1)` and compare it to the setting of `maxdsiz` and to the output of `swapinfo -t`.

The solution to out of memory problems is to provide more memory by increasing `maxdsiz` and/or adding swap space.

Internal errors of compilers or the linker will lead to aborts of the build process. Often they receive a signal due to an illegal operation. The compilers typically print a message like "compiler internal error", possibly print a stacktrace, print the signal that was received, and abort. Sometimes they leave a core file.

In case of a linker abort, check all linker input files (objects and libraries) with [`odump -verifyall`](#) (or `elfdump(1)` on IA64 and 64-bit PA-RISC) to find out if one of them is corrupt, and if so, recompile the defective files.

Otherwise check if the latest version of the compiler/linker is already installed. If not, upgrade it. If yes, check the internal knowledge databases for known problems, contact the [languages WTEC \[7\]](#) to find workarounds and/or to have them submit a new defect report. Try to get a small test case that reproduces the problem, to provide to the labs.

In case of a C/C++ compiler abort, alternatively to a test case, ask the programmer for the [preprocessed output file](#) of his sources. There is a good chance to reproduce the problem with the `.i` file.

Problems Starting A Program

There are a few reasons why a shell cannot execute a program. The most obvious is that the program file has no execute permission. The different shells then give a self-explaining message.

Another reason is that the file has an incorrect magic number. The magic number determines the type of a file, and the system can tell by the magic number if the file contains executable code or not.

```
ksh: ./a.out: Executable file incompatible with hardware
sh: a.out: Execute permission denied.
bash: ./a.out: Invalid argument
a.out: Executable file incompatible with hardware.
```

These errors are printed by the various shells when trying to execute a [PA2.0 executable](#) on a PA1.1 system. As we can see the messages are not always clear about the reason.

If you get such an error message, and you verified the file permissions are correct, then also check its file type with the [file\(1\)](#) command, and compare it with the CPU type of the system:

```
$ file a.out
a.out: PA-RISC2.0 shared executable dynamically linked -not
stripped
$ model
9000/778/B132L+
$ grep B132L /usr/sam/lib/mo/sched.models
B132L 1.1e PA7300
```

Here we have a PA2.0 executable, but an old PA1.1 system which cannot execute PA2.0 programs.

Another reason for denying execute permission is bad attributes set with [chattr\(1\)](#), e.g. SHMEM_MAGIC executables that have q3p or q4p enabled, or other executables that have q4p but not q3p enabled. In such a case, check and change the quadrant usage with [chattr\(1\)](#). Make sure the quadrants used for private data are contiguous, ie. Q1+Q2+Q3, but not Q1+Q3+Q4. For details check the section [Beyond The Limits](#).

Error Messages From The Dynamic Loader

When errors are reported by dld.sl/.so, the program could be started, but a problem occurred while loading libraries or resolving symbols. The dynamic loader will print an appropriate

message and terminate the program. On PA-RISC it uses `abort(3C)` which causes a coredump to be written.

A core file analysis is in such a case normally not required, because the error message should explain what happened. But the core file is not totally useless, because the dld message does not print the name of the program that had a problem. When running a shell script that starts several programs, there is no easy way to tell which of the programs had the problem. To find out, do

```
$ file core
core:           core file from 'a.out' - received SIGABRT
```

The output will tell you the program name.

Library Search Problems

```
/usr/lib/dld.sl: Can't open shared library: /mypath/libmylib.sl
/usr/lib/dld.sl: No such file or directory
```

The message says that the library `libmylib.sl` could not be found in any of the search paths given to the dynamic loader. The path printed in the message is the default path for this library.

To fix this problem, make sure that the library exists on the system. There is no way making the program work without it. Run [chattr\(1\)](#) on the program to find out which paths are searched by `dld.sl`:

```
$ chattr a.out
a.out:
        shared executable
        shared library dynamic path search:
          SHLIB_PATH      disabled   second
          embedded path   enabled    first   /somepath:/usr/lib
        shared library list:
          dynamic   /mypath/libmylib.sl
          dynamic   /usr/lib/libc.2
          :
```

In the above case the dynamic loader will search `libmylib.sl` in `/somepath`, `/usr/lib` (embedded path) and then in the default location `/mypath`, so make sure the library exists in one of these directories.

If the library does not exist on the system, the biggest problem here might be to find out where to get this library from. The default location may give a hint. A default location of `/usr/lib` or anything below that path indicates that this is an HP-UX library. If the path is something like `/opt/somepath/lib`, it might belong to a product that installs in `/opt/somepath`, and maybe

this product is not installed. Other paths might have been available only on the system where your program has been developed.

If the library exists in a different path than the ones searched, point the environment variables [SHLIB_PATH](#) or [LD_LIBRARY_PATH](#) to it.

In any case of uncertainty, ask the vendor of the program from where to get the library and in which path to store it.

Symbol Resolution Problems

```
/usr/lib/dld.sl: Unresolved symbol: x (code) from ./libmylib.sl
Abort(coredump)
```

The message says that there was a reference to a function named `x()` in `libmylib.sl`, but neither the executable nor one of the shared libraries contain it.

Usually there are two reasons for such a problem. Either this is a versioning problem where at runtime different versions of the shared libraries are used than at build time. In general, the first approach to resolve this problem is to get the latest versions of all used shared libraries. Or if this is a third party program, ask the vendor if it requires certain versions of the libraries, and install them.

This problem could also be caused by a link time problem. Normally, the linker checks if there are unsatisfied symbols, but only for executables, not for shared libraries. If a shared library references symbols from another library, but it is not linked against this library, the linker will not complain. When linking an executable against this library, but not against the dependent library, the linker again does not check for unresolved symbols in shared libraries and does not realize that there's something missing.

Such unsatisfied shared library symbols can be made visible at link time by applying the linker option `+vshplibunsats` when linking the executable. This makes `ld` also check for unsatisfied symbols from shared libraries, and report them:

```
/usr/ccs/bin/ld: Unsatisfied shared library symbols:
  x (first referenced in ./libmylib.sl) (code)
```

The developer then can check which library is missing.

This problem can be worked around at runtime, but this should never be considered as a final solution. You must find the shared library that contains the missing symbol with [nm\(1\)](#), and put the path to the library into [LD_PRELOAD](#) before starting the program.

The only clean solution in case of a missing library is to add it at link time.

Exec Format Error

```
/usr/lib/dld.sl: Bad magic number for shared library: ./libmylib.sl
/usr/lib/dld.sl: Exec format error
Abort(coredump)

/usr/lib/dld.sl: Bad system id for shared library: ./libx.sl
/usr/lib/dld.sl: Exec format error
ABORT instruction (core dumped)
```

These or similar errors messages appear if the dynamic loader tries to load a shared library of an executable format which is incompatible either to the executable or to the hardware. Possible causes are loading a 64-bit library to a 32-bit executable or vice versa, or loading a PA2.0 shared library to a PA1.1 executable on a PA1.1 system. To resolve this, check the [file type](#) of the library listed in the error message, and if a library with the same name exists in a different directory, which might be the right one to use. Then change the library search paths accordingly.

Thread Local Storage

```
/usr/lib/dld.sl: Can't shl_load() a library containing Thread Local
Storage: /usr/lib/libcl.2
/usr/lib/dld.sl: Exec format error
```

This error can occur when an already running program executes a call to `shl_load(3X)` or `dlopen(3C)` to load a shared library, which itself contains thread local storage (TLS), or one of its dependent libraries from the shared library list does. This type of dynamic loading of a library with TLS is not possible, as such libraries can only be loaded by `dld.sl` at program startup.

This problem has frequently been observed with [perl](#) modules that require `libcl` and/or `libpthread`, which both contain TLS. There have been perl versions out there which are not linked against both. That's why loading such perl modules will fail with the above error.

A quick way to resolve this problem is to add all required libraries, separated by semi-colons, that contain TLS into the env var [LD_PRELOAD](#). But the correct solution would be to link these libraries directly to the executable. In both cases, the libs will be loaded at program startup, and if later the program code tries to load these libs, they don't need to be loaded again.

In the case of `perl`, you can either relink it with `-lcl -lpthread` (you can because it is open source), or use the [supported perl version](#) which should already be linked properly. If neither is possible, use `LD_PRELOAD` to load these libraries when `perl` is started:

```
LD_PRELOAD=/usr/lib/libcl.2:/usr/lib/libpthread.1 perl <options>
```

Program Aborts

There can be an infinite number of reasons why a program aborts abnormally. We can distinguish between two types of aborts:

- The program encounters an error situation and stops by its own means. In this case it should print an (hopefully self-explaining) error message.
- The system (the kernel) encounters an error while executing the program, and aborts it by sending it a signal. Depending on the signal (see `signal(5)`) the system might write a core file, which is an image of the private memory of the process. The core file can be analyzed with a [debugger](#).

Core file analysis can be a very complex work, and should normally be done by the program developers, as it is they who know their programs best. This shall not be explained in-depth here, only a basic overview will be given, and a few general problems plus hints how to recognize and solve them will be discussed.

If the problem is reproducible, it often is of interest to see what happens before the abort. In that case, tracing the program's system calls with [tusc](#) might be helpful.

If there is no core file, there is still a chance to debug the problem, if it is reproducible. Run the program under the debugger. You must somehow manage to stop process execution immediately before it aborts. If we don't stop it, the program will terminate and will leave nothing to debug. If it receives a signal, the debugger will stop its execution automatically.

There are not many ways for a program to [terminate](#). If it does not abort with a signal, it most probably calls `exit(2)`. There is a good chance to catch the abort by [setting a breakpoint](#) at `exit(2)`. At this point you can do the same analysis as with a program that dumped a core file.

Getting Usable Core files

There are a number of things that can interfere with the writing of a core file. The result can either be a truncated or corrupted core file, or no core file at all. Truncated core files can be caused by the following:

- The limit for the maximum size of core files, `ulimit -c` (see `sh-posix(1)`), is too small.
- There is not enough disk space available to write the core file.
- The file system's `largefiles` option not enabled to write core files > 2 GB.

To check if a core file is complete, use `what(1)`:

```
$ what core
core:
```

```
x.c version 0.1 from Jan 04 2002
$ PATCH/11.00:PHCO_24723 Sep 13 2001 05:01:45 $
92453-07 dld dld dld.sl B.11.30 011005
$
```

If the `what(1)` output does not end with the what string of `dld/dl`, the core file is incomplete and can't be used for debugging. A core file size of 2147483646 (2 GB - 2 Bytes) indicates a missing largefiles option of the target file system. Any other size of an incomplete core file would point to lack of disk space or an insufficient `ulimit -c` setting.

The core file that a process would leave will approximately have the same size as the amount of virtual memory the process is occupying at the time of the abort. The total VSS size in the memory report of `glance(1)` (mainly Data VSS and Other VSS) is a good measure for that. Make sure all limits are set high enough.

If you cannot provide enough free space in the file system, there are ways to redirect the core file to a different location. In HP-UX 11.11 and earlier, you can create a symbolic link named `core` to any other directory, e.g.

```
$ cd /pwd/of/the/program
$ ln -s /dir/with/plenty/space/core .
```

The directory `/dir/with/plenty/space` must exist, while `/dir/with/plenty/space/core` can be a regular file or must not exist. The core file will be written to `/dir/with/plenty/space/core` then.

The same does not work in HP-UX 11.23 and later. If you try that, no core file will be written. See further down in this paragraph for more information about not getting core files. In HP-UX 11.23 there is no way to make the kernel write the core file to a different location, but with 11.31 the command [`coreadm\(1M\)`](#) can be used to define the directory where the core file will be written to.

Core file corruption can occur if multiple processes with the same working directory crash at the same time, because all core files will be stored as `$PWD/core`. To avoid this, there is a kernel flag which can be set as root, and which will make core files to be stored as `$PWD/core.<PID>`:

```
# echo core_addpid/W 1 | adb -w /stand/vmunix /dev/kmem
```

This flag will not be stored permanently. The setting will be lost after a reboot. It can also be reset manually with

```
# echo core_addpid/W 0 | adb -w /stand/vmunix /dev/kmem
```

On HP-UX 11.31 and later, [coreadm\(1M\)](#) can be used by any user to define the name of the core files created by his processes, e.g.

```
$ coreadm -p core.%p
```

Will instruct the kernel to store all core files of this user as `core.<PID>`.

The reasons for not getting a core file can be:

- The process has no write permissions in its working directory.
- The limit for the maximum size of core files, `ulimit -c`, is set to 0.
- The program has caught the signal and exits without leaving a core file.
- HP-UX 11.23 and later: The path where the core file would be written to includes a symbolic link.
- The program's real and effective user ID are not identical. This applies to setuid programs or such that have changed their effective UID actively.

First check the obvious things like access permissions and `ulimit -c`.

Then check for symbolic links, e.g. by changing to the process' working directory and comparing the output of `pwd` and `pwd -P` (see [sh-posix\(1\)](#)). If both commands give a different output, then the path contains a symbolic link. This is considered as a security risk, that's why HP-UX 11.23 and later will not allow the core dumping.

On 11.23 there is nothing you can do about this, except of making sure the path to the process' working directory does not contain a symbolic link. With HP-UX 11.31 and later, the [coreadm\(1M\)](#) can be used which allows setting various core file related attributes, e.g. a dedicated directory where the core files will be written to. It can be used to circumvent the symbolic link problem.

Coredumping of setuid programs also poses a security risk and will not be done, because unauthorized users (the ones with the effective UID) would have read permissions to such a core file, which could contain information that belongs to the user with the real UID. There is however a kernel flag that can be set as root to allow core dumping of such processes:

```
# echo dump_all/W 1 | adb -w /stand/vmunix /dev/kmem
```

Make sure to reset this flag when no longer needed:

```
# echo dump_all/W 0 | adb -w /stand/vmunix /dev/kmem
```

This flag will not be stored permanently. The setting will be lost after a reboot.

Programs which catch fatal signals that would otherwise cause an abort with core dump, should provide at least basic information like stacktraces before they exit. If they don't, the only way to get a core file for further analysis is to attach to the running process with gdb, stop it when the signal is delivered to it, and force a core dump before the signal handler is called:

```
$ gdb -q myprog 13076
(no debugging symbols found)...Attaching to program: myprog, process
13076

warning: The shared libraries were not privately mapped; setting a
breakpoint in a shared library will not work until you rerun the
program; stepping over longjmp calls will not work as expected.

(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...

0xc022edd0 in _nanosleep2_sys+0x10 () from /usr/lib/libc.2
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault
  si_code: 0 - SEGV_UNKNOWN - Unknown Error.
0xc022edd0 in _nanosleep2_sys+0x10 () from /usr/lib/libc.2
(gdb) dump
Dumping core to the core file core.13076
(gdb) c
Continuing.

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb) q
```

Getting a stacktrace from a core file

Usually debugging always starts with the same action - looking for the location of the abort, by getting a stacktrace of the thread that aborted:

```
$ gdb a.out core
HP gdb 5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
:
(gdb) info threads
* 1 system thread 3785 0x7f7a5378 in printf () from /usr/lib/libc.2
```

```
(gdb) thread 1
[Current thread is already 1 (system thread 3785) ]
(gdb) bt
Thread 1 (system thread 10567):
#0  0x7efa5cac in printf+0x18 () from /usr/lib/libc.2
#1  0x29f0 in main () at x.c:4
(gdb)
```

The interesting thread is the one marked with an asterisk in the threads listing. See the section about [gdb \(1\)](#) on how to read the stacktrace.

If frame #0 shows an address in user written code (the executable or a shared library provided with it), it is most probably a programming problem of the vendor, and then he is the one who has to fix this problem. However, it still could be that the problem is caused by a system runtime library, e.g. if a function call from a runtime library returned illegal data which lead to the abort in the user code.

If frame #0 shows an address in a system library, it must be checked what parameters had been passed to the library call, and if they were valid. Most of the time, passing invalid parameters to system function calls is the reason why applications abort.

If it turns out, or at least if it seems to be a problem with an HP provided runtime library, it is always helpful to get the program vendor to provide source code of a little test case that shows the problem. This will make the analysis of the problem much easier for HP support, expert center or the labs.

Out Of Memory

Often it is said that a program crashed because it could not allocate memory. This is many times not true. It is pretty common for an application to abort because it didn't realize that an allocation (malloc, brk, shmget, etc) returned an error code. It then tried to access a memory region that was not allocated (normally a null pointer returned by the faulting call) and received a signal (SIGBUS, SIGSEGV) instead. A well designed program will check the return code and handle the situation properly, e.g. by exiting with an error message.

The problem with such aborts is that they happen only when accessing the invalid address, not when the allocation failure occurred, so usually at the first glance it cannot be said if the program ran out of memory, or if the crash had a different reason.

To find out, we need to check the used process address space against the [system limits](#) and find the type of memory that was exhausted, if any. The solution then is to either provide more resources, or make the program use less.

Three system calls provide different types of memory to a process:

- `brk(2)`: process heap (private)
- `mmap(2)`: memory mapped regions (private or shared)
- `shmget(2)`: System V shared memory

If we don't know how the process makes use of its address space, we can either use `tusc(1)` to check which of the above system calls it executes, or look at the current usage of the process address space with [kmeminfo](#) or `glance(1)`. If the process has already died and left a core file, we can use the `$m` command of `adb(1)` to get a similar output as `kmeminfo`:

```
tusc -s brk,mmap,shmget <PID>
kmeminfo -pid <PID>
echo '$m' | adb <program> <core file>
```

System calls that fail to allocate memory set `errno=ENOMEM` which is shown by `tusc(1)` as return code. This can either be the result of a [memory leak](#), or the program might really need more memory to execute properly.

If memory allocation with `brk(2)` fails, we need to look at `ulimit -d` (see `sh-posix(1)`) and `maxdsiz[_64bit]`. Increase the kernel parameter with `kmtune(1M)` to match the process requirements. The initial `ulimit` will reflect these changes automatically.

For `mmap()`, a system call trace shows if the mapping was private or shared:

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_ANONYMOUS, -1, NULL) ..... = 0x7dfa000
^~~~~~                                     => Q2 (private)
mmap(NULL, 4096, PROT_READ|PROT_EXEC,
MAP_SHARED|MAP_SHLIB, 3, 0x1000) ..... = 0xc0004000
^~~~~~                                     => Q4 (shared)
```

The returned addresses confirm this, as the mappings go into different quadrants.

There are no kernel parameters or other virtual limits that restrict the usage of `mmap(2)`. The only limits are the process address space and the system wide available virtual memory.

Process heap, process stack and privately mapped regions all reside in the private data quadrants. If a program gets `ENOMEM` from `brk()` although the heap size has not reached the ulimit or `maxdsiz`, or if it cannot map another private region, then it ran out of private address space. The free private address space of a running process can be calculated from a `kmeminfo` output:

```
$ kmeminfo -pid <PID>
```

```

:
type      space          vaddr   ref    virt    phys    swap
TEXT 0xb288800.0x0000000000001000   3      5      5      1
DATA 0x15a6400.0x000000040001000   1    1023    766    1026
MMAP 0x15a6400.0x000000072b90000   1      13     13     14
:
STACK 0x15a6400.0x00000007dff0000   1      528    272    530
:
$
```

The `DATA` line describes the heap. Its start address is given in the `vaddr` column, and its size in 4k pages in the `virt` column. The difference between the start of the first `MMAP` region and the end of the heap shows how much is still available:

```

first MMAP starts at:          0x72b90000
Heap ends at:      0x40001000 + 1023 * 4096 = 0x40400000
-----
Available address space:      0x32790000 (ca. 800 MB)
```

The same calculation can be done with a core file, from the output of adb's `$m` command.

The process stack always starts at `(0x80000000 (0xc0000000 for q3p/q4p executables) - maxssiz)`, and the space up to the quadrant's end is reserved for the stack. If the stack doesn't grow that much, a large `maxssiz` setting just wastes address space and decreases the available space for heap and private mappings. It is recommended to reduce it as much as possible, but since `maxssiz` is a system wide limit, it is necessary to check the stack requirements of all processes running on the system.

In the above example, the stack starts at `0x7dff0000` which means `maxssiz` was set to 32 MB. On the other hand, the `virt` column shows the process has 528 pages in use which is only 2 MB, so this process wastes 30 MB of its address space here.

System V shared memory allocated by `shmget(2)`, shared mappings and shared library text (which is nothing else but shared mappings) will all reside in the shared quadrants. If an `shmget()` fails, check against the [SHM related kernel parameters](#) and increase them if required. If it still fails, or if a shared mapping fails, no more address space is available in the shared quadrants. The usage of the shared quadrants can be displayed with `shminfo`.

If a process ran out of address space, either in the shared or private quadrants, the only way out is to [change the quadrant usage](#) from private to shared or vice versa. However, the total amount of memory available for a 32-bit process is limited to 4 GB, even if the system has plenty of virtual memory. If 4 GB is not sufficient, the application must be migrated to 64-bit.

Memory Leaks

If process memory hits a system limit despite having it already increased to a reasonable value, the process is likely to have a memory leak, which means it allocates memory but fails to free it when it is no longer needed. In this case, further increasing the limits will only result in the program taking longer until the limit is hit, but the problem is not gone. Memory leaks are almost always coding errors and typically must be corrected by the programmers.

There are several non-HP tools available on the market that can help finding memory leaks in the process heap. But also the current versions of [WDB \[1\]](#) have heap checking features. To find leaks, run the program under the debugger and switch on the heap-checking:

```
$ gdb a.out
HP gdb 5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.11
:
(gdb) set heap-check on
(gdb) r
:
[interrupt at any time with CTRL-C, or wait until a breakpoint is hit]
:
(gdb) info leaks
Scanning for memory leaks...

Execution of code located on a program's stack is not permitted.
cmd: /tmp/a.out

30000 bytes leaked in 3 blocks

No.    Total bytes      Blocks      Address      Function
0       30000           3          0x4042d2b8    f10()
(gdb) info leak 0
30000 bytes leaked in 3 blocks (100.00% of all bytes leaked)
These range in size from 10000 to 10000 bytes and are allocated
#0  f10() at x.c:7
:
#10  main() at x.c:20
(gdb)
```

`info leaks` lists all leaks that have been detected, and `info leak <num>` shows the details of a certain leak, including the stacktrace from the location where `malloc()` was called. More details about the heap-checking features of `gdb` can be found in the `gdb` online help on any HP-UX system with a recent `gdb` installed:

- <file:///opt/langtools/wdb/doc/html/wdb/C/index.html>

Stack Overflow

In the [process memory](#) section we learned that the stack holds a lot of important information for the program flow. A stack overflow or corruption will normally cause the process to abort with a signal and leave a core file, but debugging such a core file can become very hard, because the debugger might not be able to recover the program flow information. In that case you cannot even [get a valid stacktrace](#) from the core file.

At runtime you cannot see the stack grow with `top(1)` because it doesn't distinguish the type of allocated memory. Also, you can't see stack growth with `tusc(1)` because no system calls are involved when increasing the stack. To find out the current stack size of a running process is to print out the value of its stack pointer using a debugger.

The stack starts at `0x80000000-(maxdsiz+0x1000)` resp. `0xc0000000-(maxdsiz+0x1000)` for q3p/q4p programs, and ends at the end of the quadrant. The additional size of `0x1000` is due to the 4k guard page, which is added automatically by the kernel. The guard page is a region protected against both read and write accesses. When a program tries to access an address within the guard page, it will be aborted with `SIGBUS`.

On PA-RISC the stack grows upwards, and the guard page is located at the end of the stack region. The stack pointer `$sp` always points to the current upper end of the stack:



A stack overflow occurs when `$sp` reaches the guard page. We can find out the value of `$sp` as follows:

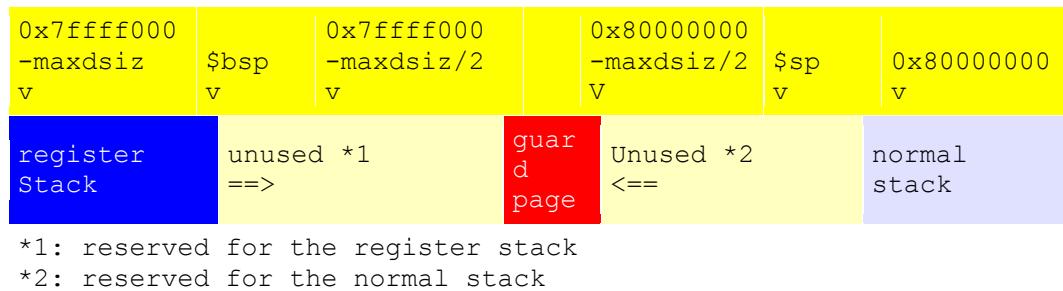
```
$ gdb -q myprog 7753
:
(gdb) p/x $sp
$1 = 0x7f7f0b10
(gdb) x/x 0x7fffff00      ### address in the middle of the guard page
0x7fffff00:      Error accessing memory address 0x7fffff00: Bad address.
(gdb) q
```

Here we still have `0x80e4f0` (ca. 8 MB) Bytes available for further stack growth.

On Itanium the stack works a bit differently. While on PA-RISC the stack holds both register contents and local variables mixed together in the same area, the Itanium architecture splits the stack region into 2 parts and separates the register contents from other local variables.

The register stack starts at the lower end of the stack region and grows upwards, while the normal stack starts at the upper end and grows downwards. The guard page is located in the middle of the stack to separate both parts from each other.

We also have 2 different processor registers for each part of the stack. The backingstore pointer `$bsp` always points to the current upper end of the register stack, and the stack pointer `$sp` always points to the current lower end of the normal stack:



A stack overflow occurs if either `$bsp` or `$sp` reach the guard page. To find out the register values, use `gdb` as follows:

```
$ gdb -q myprog 7565
:
(gdb) p/x $bsp
$3 = 0x200000007efff348
(gdb) p/x $sp
$4 = 0x200000007fffff3d0
(gdb) x/x ($sp+$bsp)/2      ### address in the middle of the guard page
0x200000007f7ff38c:          Error    accessing    memory    address
0x200000007f7ff38c: Bad address.
(gdb) q
```

The same checks can be done with a core file.

In a multithreaded program, each thread needs its own stack. While the initial thread (also called the primordial thread) uses the process stack, as described above, any other thread will use a memory mapped region as its stack, which will be allocated immediately before the thread is created, and which will be unmapped after the thread has exited.

Each thread stack has the same layout as the primordial thread stack, but its size is not determined by the kernel parameter `maxdsiz`, but rather by the program, or by the default `pthread_stacksize`, which is 64k on PA-RISC, and 256k on Itanium. Obviously thread stacks are much

smaller than the primordial thread, that's why stack overflows are much more likely in multithreaded programs.

These default sizes are hardcoded into the POSIX threads library `libpthread[.sl|.so]`. Programs can specify the stacksize for each new thread before it is created with `pthread_attr_setstacksize(3T)`. Alternatively they can also change the default stacksize with `pthread_default_stacksize_np(3T)` before they start creating threads.

The default thread stacksize can also be defined before starting a program, with the environment variable `PTHREAD_THREAD_STACK_SIZE`. This feature is natively implemented in HP-UX 11.31. HP-UX 11.11 needs at least phtread patch PHCO_36229, and on HP-UX 11.23 pthread patch PHCO_34718 or later is required.

Checking for thread stack overflows means looking up stack pointer addresses with the debugger, and comparing them with the thread stack boundaries. Select the thread and frame that received the signal, and print the stack pointer:

```
$ gdb <program> core
:
(gdb) info threads
*      224  system  thread  509883      0xc020c718  in  kill+0x10  ()  from
/usr/lib/libc.2
      223  system  thread  555147      0xc020b540  in  __ksleep+0x10  ()  from
/usr/lib/libc.2
:
$ thread 224
(gdb) thr 224
[Switching to thread 224 (system thread 509883)]
#0  0xc020c718 in kill+0x10  ()  from lib/libc.2
(gdb) bt
:
#8  <signal handler called>
#9          0xcaac2d34      in alloc_object+0xc      ()      from
/opt/java1.4/jre/lib/PA_RISC2.0/hotspot/libjvm.sl
:
(gdb) frame 9
#9          0xcaac2d34      in alloc_object+0xc      ()      from
/opt/java1.4/jre/lib/PA_RISC2.0/hotspot/libjvm.sl
(gdb) p/x $sp
$6 = 0x7360d300
```

Then check e.g. with `adb` how this address fits into the memory mapped thread stacks:

```
$ echo '$m' | adb <program> core
:
MMF:           b = 73509000      e = 7358A000      f = 9A780A4
MMF:           b = 7358A000      e = 7360B000      f = 9AF90B4
```

```
MMF:          b = 7360B000      e = 7368C000      f = 9B7A0C4
              ^^^^^^
MMF:          b = 7368C000      e = 7370D000      f = 9BFB0D4
:
$
```

In the above case the stack pointer is close to the start of the highlighted MMF region, that means only a tiny bit of the thread stack was used and we were far away from a stack overflow. If the stack pointer is near the end of the region, remember that the last 4k belong to the guard page and the real end of the stack is 4k earlier than the end of the region.

On Itanium, both `$sp` and `$bsp` need to be checked. Since the guard page lies in the middle of the thread stack, there might have been a problem if one of the registers reached the middle of the thread stack region.

Stack overflows can be caused by infinite or too deep recursions, or by big local variables. If increasing `maxssiz` or the thread stacksize to a reasonable value doesn't help, the programmers may need to change the code, e.g. by allocating large local variables dynamically with `malloc(3C)`. Fortran programmers can use the `+save` compiler option for that.

Heap Corruption

An abort under `malloc()` or any other heap allocation related function indicates a heap corruption, e.g.:

```
#0 0xc0185588 in mallinfo+0xec8 () from /usr/lib/libc.2
#1 0xc0182c30 in __thread_callback_np+0x4f0 () from /usr/lib/libc.2
#2 0xc01888b0 in malloc+0x198 () from /usr/lib/libc.2
#3 0x294c in main+0x2c ()
```

Heap corruption means that data structures that are required to manage the heap contents, and which are part of the heap, have been overwritten. Most of the time this is NOT bug in libc, but rather a defect in the program, as we will see further down in this section.

The process heap is organized in kind of a linked list of blocks. For every allocated block, `malloc()` stores a so called malloc header right before the block, which contains among other things the length of the block, and a flag that indicates if the block is currently used by the program or is free for reallocation. Right after the allocated block resides the malloc header for the next block.

If a program writes beyond the end of an allocated block, this is called a buffer overrun. If it writes beyond the beginning, this is a buffer underrun. Both will result in overwriting a malloc header, thus corrupting the heap structures. Another cause could be writing to a block that has already been freed (returned to the malloc pool for reuse).

This is even harder to track down than unchecked allocation failures, because again the abort does not occur when overwriting the heap structures, but only on one of the next calls to `malloc()` or `free()`. Also, the overwriting could have happened everywhere in the code, whenever the program accesses the heap. There is no special function call to look for.

WDB's heap checking features can help here. When switching on the heap checking, it inserts wrappers for the libc allocation functions. Whenever a buffer is allocated, the wrappers will allocate a few more bytes, and insert a bit pattern at the beginning and the end of the block. When the buffer is freed, the wrappers check if the patterns are intact, and if not, tell you the stack trace where the buffer has been allocated:

```
(gdb) set heap-check on
(gdb) run
Starting program: /space/mstreibe/cases/heap/a.out
warning: Memory block (size = 10 address = 0x7dff2470) appears to be
corrupted at the end.
Likely block allocation context is

#1 main() at x.c:10
#2 __start() from /usr/lib/libc.2
#3 __start() at ../../Src/gnu/gdb/infrtc.c:2632
#4 $START$() from
warning: Use command backtrace (bt) to see the current context.

Ignore top 3 frames belonging to leak detection library of gdb
(gdb)
```

Here the buffer had a size of 10 bytes and was allocated in `main()`, line 10 of source file `x.c`.

In a second step we can make gdb stop the program when `malloc()` returns the address of the buffer that gets its borders overwritten:

```
(gdb) bt
#0 __rtc_event () at ../../Src/gnu/gdb/infrtc.c:699
#1 0x7de927b0 in check_bounds (pointer=0x4042c448, size=10,
    pclist=0x40407454) at ../../Src/gnu/gdb/infrtc.c:759
#2 0x7de93d50 in rtc_record_free (pointer=0x4042c448,
    free_the_block=1) at ../../Src/gnu/gdb/infrtc.c:1566
#3 0x7de94908 in __rtc_free (pointer=0x4042c448 "1234567890") at
    ../../Src/gnu/gdb/infrtc.c:1932
#4 0x2ab8 in main (argc=2, argv=0x7dff0ad4) at x.c:14
#5 0x7def79e4 in __start+0xc8 () from /usr/lib/libc.2
(gdb) frame 4                                ← go to main()
#4 0x2ab8 in main (argc=2, argv=0x7dff0ad4) at x.c:14
14         free(p[i]);                      ← print the buffer address
(gdb) print p[i]
$1 = 0x4042c448 "1234567890"
(gdb) set heap-check watch 0x4042c448      ← watch the address
```

```
(gdb) r                                     ↵ rerun the program
The program being debugged has been started already.
Start it from the beginning? (y or n) y
warning: Temporarily disabling or deleting shared library breakpoints:
warning: Deleting breakpoint #-50
warning: Deleting breakpoint #-51
Starting program: /space/mstreibe/cases/heap/a.out 1234567890
warning: Watched address 0x4042c448 allocated
__rtc_event () at ../../../../Src/gnu/gdb/infrtc.c:699
699      ../../../../Src/gnu/gdb/infrtc.c: No such file or directory.
          in ../../../../Src/gnu/gdb/infrtc.c
(gdb)
```

Now we could check what the program does with this buffer, e.g. by single stepping until we find the line of code that overwrites the buffer borders.

Or we could set a watchpoint to the buffer borders. This would make gdb stop the program when the values at these addresses change.

Threads-Library Mismatch

```
#0 0xc01339a4 in () from /usr/lib/libc.2
#1 0xc0206b2c in __thread_atfork () from /usr/lib/libc.2
#2 0xc02046a4 in __libc_init () from /usr/lib/libc.2
#3 0xc0adfaf4 in hp__pre_init_libc () from /usr/lib/libcma.2

#0 0xc102c3b4 in pthread_mutex_init () from /usr/lib/libpthread.1
#1 0xc02047c4 in libc_init () from /usr/lib/libc.2
#2 0xc0203f0c in __libc_init () from /usr/lib/libc.2
#3 0xc0adfaf4 in hp__pre_init_libc () from /usr/lib/libcma.2
```

If you see a core file with a stack trace like one of the above which shows things that seem related to threads initialization, check with [ldd\(1\)](#) which libraries are loaded by the program. If both `libcma` (user threads) and `libpthread` (kernel threads) are listed, you most probably have found the cause. Both libraries are incompatible and must not be mixed. One of them must be removed. Which one depends on the program vendor, but most of the time kernel threads are the ones wanted on 11.X and later.

The threads mismatch can occur with programs that are ported from HP-UX 10.X to 11.X. On 10.X there were no kernel threads, and every multithreaded program used CMA threads. When ported to 11.X, and if the program uses third party libraries, e.g. the oracle library `libclntsh.s1`, the mismatch occurs, if the program is not ported to kernel threads, because the oracle library uses kernel threads while the program still uses CMA threads.

Java Abort

If the JVM aborts, it will always try to leave a core file. Depending on the java version and the reason of the abort, it sometimes writes additional information into a file named `hs_err_pid<PID>.log`, where `<PID>` is the process ID of the aborted java process.

When looking at a java core file, make sure to use the correct java executable. `/opt/java1.4/bin/java` is not the real java executable. It is only a wrapper. The correct java executables, e.g. for java 1.4, are:

<code>/opt/java1.4/bin/PA_RISC/java</code>	(PA-RISC 1.1)
<code>/opt/java1.4/bin/PA_RISC2.0/java</code>	(PA-RISC 2.0)
<code>/opt/java1.4/bin/PA_RISC2.0W/java</code>	(PA-RISC 64-bit)
<code>/opt/java1.4/bin/IA64N/java</code>	(IA64 32-bit)
<code>/opt/java1.4/bin/IA64W/java</code>	(IA64 64-bit)

Debugging a java core file normally requires deep knowledge of the internal structure of the JVM. In many cases, debugging java core files means getting a stacktrace and checking in [CHART \[9\]](#) for known problems. If there are no matches, forward the core file to the [Languages WTEC \[7\]](#).

Besides a JVM internal error, it is also possible that the JVM aborts because of a problem with the application running inside the JVM, or, if JNI is used, because of a bug in the native part (3rd party shared libraries). JNI aborts can be handled like any other core file, and checked against the abort reasons discussed earlier in this chapter.

Since gdb version 3.1 and java 1.3.1.02, gdb is able to unwind through the java methods (interpreted as well as compiled). All java depots since 1.3.1.02 contain the library `libjunwind.so`, e.g. for java 1.4:

<code>/opt/java1.4/jre/lib/PA_RISC/server/libjunwind.so</code>	(PA-RISC 1.1)
<code>/opt/java1.4/jre/lib/PA_RISC2.0/server/libjunwind.so</code>	(PA-RISC 2.0)
<code>/opt/java1.4/jre/lib/PA_RISC2.0W/server/libjunwind.so</code>	(PA-RISC 64-bit)
<code>/opt/java1.4/jre/lib/IA64N/server/libjunwind.so</code>	(IA64 32-bit)
<code>/opt/java1.4/jre/lib/IA64W/server/libjunwind.so</code>	(IA64 64-bit)

gdb tries to find the appropriate java unwind library on its own. If it is not able to find it, you can set the environment variable `GDB_JAVA_UNWINDLIB`:

```
$ export
GDB_JAVA_UNWINDLIB=/opt/java1.4/jre/lib/PA_RISC2.0/server/libjunwind.so
$ gdb /opt/java1.4/bin/PA_RISC2.0/java core_1
:
```

```
(gdb) bt
#0 0xc020a6d0 in kill+0x10 () from /usr/lib/libc.2
#1 0xc01a55cc in raise+0x24 () from /usr/lib/libc.2
#2 0xc01e59b0 in abort_C+0x160 () from /usr/lib/libc.2
#3 0xc01e5a0c in abort+0x1c () from /usr/lib/libc.2
#4 0xc455d468 in abort__2ossFb+0x108 () from
/opt/java1.4/jre/lib/PA_RISC2.0/server/libjvm.sl
#5 0xc441f834 in report_error+0xb4 () from
/opt/java1.4/jre/lib/PA_RISC2.0/server/libjvm.sl
#6 0xc441ef8c in report_fatal+0x6c () from
/opt/java1.4/jre/lib/PA_RISC2.0/server/libjvm.sl
#7 0xc47a9a0c in compute_compiled_exc_handler+0xe4 () from /opt/java1
.4/jre/lib/PA_RISC2.0/server/libjvm.sl
#8 0xc47a9698 in handle_exception_C_11OptoRuntimeSFP10JavaThread+0x14
8 () from /opt/java1.4/jre/lib/PA_RISC2.0/server/libjvm.sl
#9 0x78f18108 in exception_stub frame ()
#10 0x795cf574 in compiled frame: java.lang.ClassLoader::defineClass (j
ava.lang.String, byte[], int, int, java.security.ProtectionDomain) ->ja
va.lang.Class ()
#11 0x790429f0 in compiled frame: com.borland.enterprise.module.reflec
tion.classes.custom.CustomClassLoader::findClassInternal (java.lang.Stri
ng) ->java.lang.Class ()
#12 0x7935e2b0 in compiled frame: com.borland.enterprise.module.reflec
tion.classes.custom.CustomClassLoader::findClass (java.lang.String) ->ja
va.lang.Class ()
#13 0x7959f460 in compiled frame: com.borland.enterprise.module.reflec
tion.classes.custom.CustomClassLoader::loadClass (java.lang.String, bool
ean) ->java.lang.Class ()
:
```

In the above example the frames #0–#9 are from native code, all others are methods compiled by the hotspot compiler. Here we see that an exception was thrown in method `java.lang.ClassLoader::defineClass()` which could not be caught.

Without using `libjunwind.sl`, the stacktrace would look like this:

```
$ gdb /opt/java1.4/bin/PA_RISC2.0/java core_1
:
(gdb) bt
#0 0xc020a6d0 in kill+0x10 () from /usr/lib/libc.2
#1 0xc01a55cc in raise+0x24 () from /usr/lib/libc.2
#2 0xc01e59b0 in abort_C+0x160 () from /usr/lib/libc.2
#3 0xc01e5a0c in abort+0x1c () from /usr/lib/libc.2
#4 0xc455d468 in abort__2ossFb+0x108 () from
/opt/java1.3/jre/lib/PA_RISC2.0/server/libjvm.sl
#5 0xc441f834 in report_error+0xb4 () from
/opt/java1.3/jre/lib/PA_RISC2.0/server/libjvm.sl
#6 0xc441ef8c in report_fatal+0x6c () from
/opt/java1.3/jre/lib/PA_RISC2.0/server/libjvm.sl
```

```
#7 0xc47a9a0c in compute_compiled_exc_handler+0x1e4 () from /opt/java1.4/jre/lib/PA_RISC2.0/server/libjvm.sl
#8 0xc47a9698 in handle_exception_C_11OptoRuntimeSFP10JavaThread+0x148 () from /opt/java1.4/jre/lib/PA_RISC2.0/server/libjvm.sl
#9 0x78f18108 in ?? ()
```

Java Out Of Memory

Of course, for java the same rules apply as for all other processes running [out of memory](#). But because java is a virtual machine with its own memory management for the applications running in it, there are a few additional things to mention here.

Normally the JVM reports a `java.lang.OutOfMemoryError` if it wasn't able to allocate space for a new object. So usually it is not difficult to tell if a JVM ran out of memory. But we have to distinguish between different out of memory conditions:

- in the java heap (application level)
- in the process heap (JVM level)
- could not create a new thread (JVM level)

There are a few methods to check if the `java.lang.OutOfMemoryError` occurred in the java heap (for the java application). Since version 1.4.2 the `hs_err_pid<PID>.log` file contains some heap statistics:

```
Heap at VM Abort:
Heap
  def new generation  total 314560K, used 314559K [63400000, 78950000,
  78950000)
    eden space 279616K, 100% used [63400000, 74510000, 74510000)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    from space 34944K,  99% used [74510000, 7672fff8, 76730000)
    to   space 34944K,   0% used [76730000, 76730000, 78950000)
  tenured generation   total 699072K, used 699072K [38800000, 632b0000,
  632b0000)
    the space 699072K, 100% used [38800000, 632b0000, 632b0000,
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
    632b0000)
    compacting perm gen  total 16384K, used 2847K [34800000, 35800000,
  38800000)
      the space 16384K,  17% used [34800000, 34ac7da8, 34ac7e00, 35800000)
```

The java heap is divided into several pieces:

- The new generation heap. New objects are created here. The new space is divided into 2 parts:
 - The eden space.
 - The survivor space is again divided into 2 parts:
 - from space
 - to space
 - The old or tenured generation heap. Long lived objects will be moved to here.
 - The permanent generation heap. Java class information is stored here.

During a garbage collection (GC) the JVM moves objects between these spaces depending on their age, or removes them from the heap if they are no longer in use.

If, as in the output above, the marked lines show 100% used for the old generation heap, the java heap was full.

For older Java versions that do not include this information, the GC log can be checked. To obtain one, the JVM must have been started with the `-Xverbosegc:<logfile>` option. The JVM will print one line after each garbage collection:

<GC: 2 1 105.263506 4 280008 1 286326784 286326768 286326784 35665424
10691400 35782656 715849656 715849712 715849728 3056752 2899480
16777216 29.482876 29.482876>

Here is the same line again, reformatted for better readability:

		before	after	total	
<GC: 2 1	105.263506 4 280008 1	286326784	286326768	286326784	eden
		35665424	10691400	35782656	from/to
		715849656	715849712	715849728	old
		3056752	2899480	16777216	perm
29.482876	29.482876 >				

The above line shows that after the GC eden and old space were still full, the JVM was not able to free space and ran out of memory.

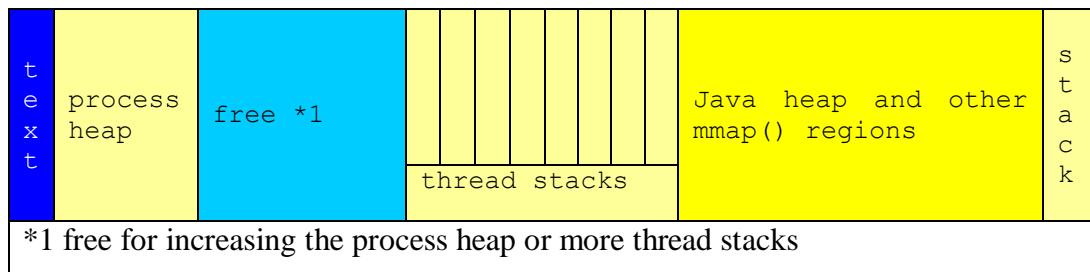
The GC log format may vary from version to version. Details can be found in the output of `java -Xverbosegc;help`.

In case of an application OutOfMemoryError, the GC log should be analyzed with HPjmeter, which can be downloaded from the [HP java website \[1\]](#). Depending on the data from the GC logs, increasing the java heap (java option `-Xmx`) and/or changing other heap related java options can be the solution. But the GC log might as well indicate a memory leak in the application. In this case a java heap analysis might be required to find out what is filling up the heap. This can

also be done with HPjmeter. An excellent document about this and other tasks is the java troubleshooting guide at

- http://h18012.www1.hp.com/java/support/troubleshooting_guide.html

An out of memory condition in the process heap can be treated the same way as an out of memory for any other process. The JVM might have hit `maxdsiz` or `ulimit -d` while trying to increase the process heap. But if both values are large enough, it might be that the JVM ran out of address space. Here is a simplified view of a JVM's private address space:



Memory mapped regions will be allocated top down, from the lower end of the process stack towards the upper end of the process heap. The process heap will grow upwards. The JVM will run out of memory if the free area between the process heap and the thread stacks is used up. To check if this happened, you can e.g. use `adb` on the core file:

```
$ echo '$m' | adb /opt/java1.4/bin/PA_RISC2.0/java core
:
Data:      b = 12000          e = 0xB2E0000          f = 1E05C
          ^^^^^^^^^^
MMF:      b = 5228D000        e = 5230E000          f = 0xB2EC06C
          ^^^^^^^^^^
$
```

By calculating the difference between the start of the first MMF region and the end of the Data segment, we can find out the free address space that was left. In the above case this is $0x5228d000 - 0xb2e0000 = 0x46fad000 = 1.1\text{ GB}$.

To provide enough private address space for different `-Xmx` values, the PA-RISC JVM is shipped with 3 java executables, which are executed depending on the `-Xmx` value, and which use the quadrants in different ways:

Executable	-Xmx values	Quadrant Usage				private address space
		Q1	Q2	Q3	Q4	
java	< 1500m	text + private data	private	shared	shared	2 GB
java_q3p	1500m – 2400m	text + private data	private	private	shared	3 GB

java_q4p	> 2400m	text + private data	private	private	private	4 GB
----------	---------	---------------------	---------	---------	---------	------

The IA64 platform knows another [address space model](#), which is used by the JVM for applications that need a large java heap. Here only 2 java executables are shipped:

Executable	-Xmx values	Quadrant Usage				private address space
		Q1	Q2	Q3	Q4	
java	< 1600m	text + private data	private	shared	shared	2 GB
java_q4p	> 1600m	text + private data	private	mostly private	mostly private	4 GB

The limits for the -Xmx option might change with different java versions.

These -Xmx limits try to make sure that there is enough private address space left for the JVM itself (for the process heap, thread stacks etc) after the java heap has been allocated. There might however be cases where the remaining space is not sufficient, e.g. because the java application has a native (JNI) part which needs a lot of space on the process heap, or because the java application creates many threads which require a lot of space for the thread stacks.

In such a case, [chatr\(1\)](#) can be used on PA-RISC to change the quadrant usage of the java executables. On IA64 java knows the command line option -Xmpas:on, which makes the JVM invoke `java_q4p`, no matter what the -Xmx value is.

To solve JVM out of memory problems, one has to balance the memory requirements of the java application and the JVM, and use the precious address space carefully.

The JVM doesn't need a large process stack, so `maxssiz` should be kept small. The default of 8 MB is plenty for java, but consider that this affects all processes on the system, and other processes might require a larger `maxssiz`.

If the java application creates a lot of threads, the thread stack size may become crucial. E.g. if there are 1000 threads (which is not so unusual for a large server application), a thread stack size of 512k means the thread stacks will occupy 512MB of the address space. Reducing the the stack size carefully with the `-xss` option can save several 100MB in such a case.

Note that the environment variable `PTHREAD_THREAD_STACK_SIZE` has no effect on the stacksize of java threads. The JVM will use its own defaults (128k on PA-RISC and 512k on Itanium, but this might vary between different java versions), or the `-xss` command line setting.

Especially on Itanium, with a high number of threads it might save a lot of space if the thread stacksize is slightly reduced from 512k to 508k. Because of the guard page, a 512k stacksize results in a 516k mapping, and because of the granularity used by the kernel for such mappings, a 516k mapping will in reality occupy 768k (512k+256k), with 252k wasted per thread. With a

reduced stacksize of 508k, both stack and guard page will fit into 2x 256k, and the mapping will occupy exactly 512k, with nothing wasted.

The java heap size (-Xmx option) might need to be adjusted to reflect the real requirements of the java application. A too large java heap wastes address space at the expense of the JVM. The required heap sizes can be found by analyzing GC logs with HPjmeter.

In some cases it might help to increase the java heap. E.g. if the JVM ran out of memory with a java heap of 1400MB (which left nearly 600MB for the JVM), increasing the heap size to 1600MB will cause `java_q3p` being used instead of `java`, and with its 3GB private address space, now nearly 1400MB can be used by the JVM.

If all else fails, the 64-bit JVM can be used with the -d64 option, which is available since java 1.4. But sometimes this is not applicable, especially if JNI is used, because the 3rd party native libraries might not be available as 64-bit versions.

The JVM will also report a `java.lang.OutOfMemoryError` when it was not able to create a new thread. There are two possible reasons for that. Either there was not enough space to allocate (map) a new thread stack. The solution in this case is the same as above, when the JVM runs out of process heap. Or the maximum number of threads per process (`max_thread_proc`) or system wide (`nkthread`) has been reached.

The number of threads can be found out with `gdb` to compare it with the kernel parameters:

```
$ echo info thr | gdb /opt/java1.4/bin/PA_RISC2.0/java core | \
  grep -c "system thread"
224
$
```

To solve this problem, either increase the kernel parameters, or check if the application should really create as many threads as it had. It might be a problem in the application, if it creates an unexpectedly high number of threads.

Environment Variables

This chapter gives an overview to some helpful environment variables that are used by the products described in this document.

Compiler Related Variables

Every HP compiler knows a specific environment variable that is read by the compiler frontend to allow specifying additional command line parameters without making changes to the command line itself.

The contents of this variable are handled as if they were passed to the compiler on the command line. This makes it easy to pass additional parameters to a compile command, even if the command itself is located somewhere in a cascaded makefile structure.

The following table lists these variables for each compiler:

Variable	Compiler
CCOPTS	C (bundled and ANSI)
CXXOPTS	ANSI C++
HP_F90OPTS	Fortran 90

E.g. if we wanted to switch on the verbose mode of cc, we simply could do

```
export CCOPTS=-v      # for sh, ksh
setenv CCOPTS -v      # for csh
```

Then every subsequently executed compiler call would use the `-v` option, although it is not specified at the command line, and so each compilation would be done in verbose mode.

Because often the order of the compiler options is important, these environment variables provide a way to add the options before and after the options specified on the command line. E.g. if we would do the following:

```
export CCOPTS="-v|-lcl"
cc -o HelloWorld HelloWorld.c
```

the resulting command line would be

```
cc -v -o HelloWorld HelloWorld.c -lcl
```

Everything before the pipe symbol is added at the beginning of the command line, and everything after the pipe symbol at the end. Details on using these variables can be found in the appropriate compiler man pages.

Linker Related Variables

In fact, there is only one environment variable for the linker, `LD_OPTS`. It has the same function as the compiler variables, and it also works in the same way.

Runtime Related Variables

Of course, it is totally left to the programmer to determine which environment variables his program reads, if any. But because every program uses `dld.sl`, at least the programs that are linked shared, there are a few variables that allow us to tell the dynamic loader, how to load shared libraries.

`SHLIB_PATH` can contain one or more colon separated directories that `dld.sl` will search first when looking for shared libraries, before looking in the default locations stored in the executable. To put `SHLIB_PATH` in effect for an executable, it must have “library search” via this `+s` variable enabled. This setting is determined by a flag in the executable that can either be set at link time by using the `+s` linker option, or it can be viewed and changed with [chattr\(1\)](#).

`LD_LIBRARY_PATH` has the same purpose, effect and usage as `SHLIB_PATH`, but it only works for ELF programs (IA64 and 64-bit PA-RISC). This variable was introduced for compatibility reasons to other Unix derivates (e.g. SunOS, Linux).

`LD_PRELOAD` can contain one or more colon separated shared library file names which will be loaded first, before any of the shared libraries listed in the executable's shared library list are loaded. The effect is that symbol resolution at runtime will change. Every symbol will first be searched in the libraries listed in `LD_PRELOAD`, before it is searched in the libraries linked to the executable. **Be very careful when using this variable because it can change the behaviour of a program significantly.** When using `LD_PRELOAD` you should never do:

```
$ export LD_PRELOAD=<library>
$ <program>
```

because every subsequently executed program and command would use the library specified in `<library>`, which could lead to undesired results. To make a certain program use an additional library, it is better to start it with:

```
$ LD_PRELOAD=<library> <program>
Or
$ LD_PRELOAD=<library>
$ program
$ unset LD_PRELOAD
```

Then `LD_PRELOAD` is only exported for `<program>`.

There is also a related environment variable called LD_PRELOAD_ONCE. The LD_PRELOAD_ONCE feature is similar to LD_PRELOAD except that the dynamic loader, dld.so, unsets LD_PRELOAD_ONCE after reading it, so that any applications invoked by the current application do not have LD_PRELOAD_ONCE set. This is useful in situations where the current application needs certain libraries preloaded while the child application is adversely affected if these are preloaded (for example, ksh terminates abnormally if LD_PRELOAD contains /usr/lib/hpux[32|64]/libpthread.1).

PTHREAD_THREAD_STACK_SIZE determines the thread stacksize used by the pthread library. It might be helpful to solve thread stack overflows in applications which need more than the default thread stacksize (64k on PA-RISC, 256k on Itanium).

gdb Variables

As mentioned in the [Debugging chapter](#), for correct debugging results it is crucial to debug with the correct shared library versions. Often it is the case that these versions must be copied to the debugging system where they cannot be installed in their original paths. These libraries can be copied into some other directory. gdb will pick them if the directory is listed in the variable GDB_SHLIB_PATH.

For java debugging, putting the correct version of libjunwind.sl into GDB_JAVA_UNWINDLIB enables gdb to unwind stacks through interpreted and compiled java methods, and other java related features, which become more and more with newer java and gdb versions.

Additional Information

HP provides lots of information to software development on its websites at hp.com. But as always, the larger the amount of information available, the harder is it to find something on a specific topic. This chapter lists the most important links that lead to software development related pages on the HP website as well as on other resources.

Official HP Websites

These are the main entry points when looking for documentation about software development in general and to development products.

- [1] <http://www.hp.com/go/<product>>
Replace <product> with the product you are looking for, and it will redirect you to the official product webpage within the HP website. This works for a lot of product names, e.g. “C”, “C++”, “Fortran”, “java”, “gdb”, “wdb”.
- [2] <http://www.hp.com/go/bsc>
- The Business Support Center is the HP website for customers to get access to official documents to HP products. The above link leads to online programmers guides and release notes for compilers, the linker and programming toolkits under the “Manuals” link.
- [3] <http://www.hp.com/go/dspp>
This is the developer & solution partner portal. This site is dedicated to software development on HP systems, not only HP-UX, but also Linux and Windows. You can find links to all development specific information sources like online programmers guides, release notes, online tutorials, sample codes, newsgroups, books and also to non-HP information sources.
- [4] <http://devresource.hp.com/>
This is another entry point when searching development related information. Its focus lies on integration of applications with HP software.
- [5] <http://software.hp.com>
The HP Software Depot. This is the central website for downloading HP software. All products like compilers and developers toolkits that must be purchased, can be ordered here. But also free supported products can be obtained from here.
- [6] <http://www.hp.com/go/hpsc>
The HP Solution Center. Provides contract customers with access to the knowledge database, allows patch and software update downloads, and many things more.

HP Internal Links

- [7] <http://wtec.cup.hp.com/~<team>/>

These are the homepages of the WTEC teams. The most relevant team names are:

- Languages WTEC: lang
- HP-UX WTEC: hpx
- Graphics WTEC: xgraph

These pages provide information to all supported products and other valuable information.

- [8] <http://ktools.france.hp.com/~ktools/cgi-bin/downloads.cgi>

The KTOOLS Website that provides latest versions of helpful tools like tusc, kmeminfo etc.

- [9] http://chart.hp.com/ddts/ddts_main

CHART - the bug tracking system used by the HP labs.

External Links

- [10] <http://hpx.asknet.de>

The german mirror of the HP-UX porting center. Here you can download many freeware packages that have been ported to HP-UX by HP. These packages are available as SD depots, and most of the time the sources are also available. Note that the products offered there are not supported by HP.

- [11] <http://www.gnu.org/directory/gcc>

The Gnu Compiler Collection

- [12] <http://www.cpan.org/>

Comprehensive Perl Archive Network. Repository for all perl programming related resources including modules.