Software Development

INDEX

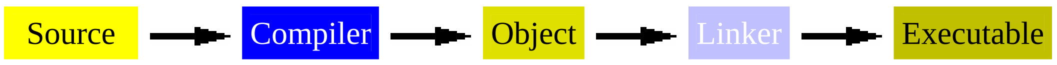
Preface

This chapter is intended to give an overview over issues on software development in general, and on HP-UX in particular. It has been derived from the training materials of the Software Development Workshop held a few times in Germany, some time ago. It is not intended to teach programming languages, and if at all, only basic skills in programming languages, and probably a little shell programming knowhow are required to understand its contents.

The Software Development chapter will cover things like what happens when building a program, what happens when it is executed, which products and tools are available on HP-UX, and it will discuss common problems and give approaches on how to solve them.

The Build Process

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



A programmer writes a program in his preferred programming language. The program code is stored in an ascii file which normally is called the source file. Because the processor doesn't 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 frontend command which 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 so to ease creating an executable program from a source file.

The compiler frontends know various input file types and distinguish them by their file name extension. They will pass them directly to the appropriate subprocess:

File Type	Subprocess	Extensions
source files	compiler	depend on the programming language
intermediate files	compiler	`.i`
other files (objects, libraries, unknown)	linker	`.o, .a, .sl` and other

By default, the frontends will execute all necessary steps build steps to create an executable from the input files passed to it. So 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 many other informations 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 or methods, the operands are also called arguments or parameters, and the result is normally 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 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 Organisation

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.

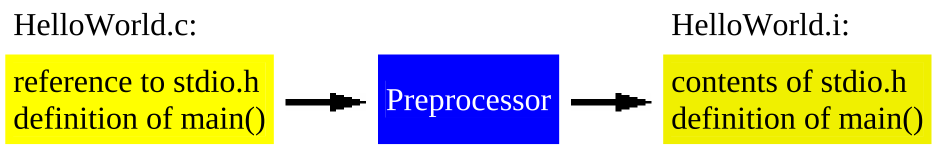
Compiling

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 whereever 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.



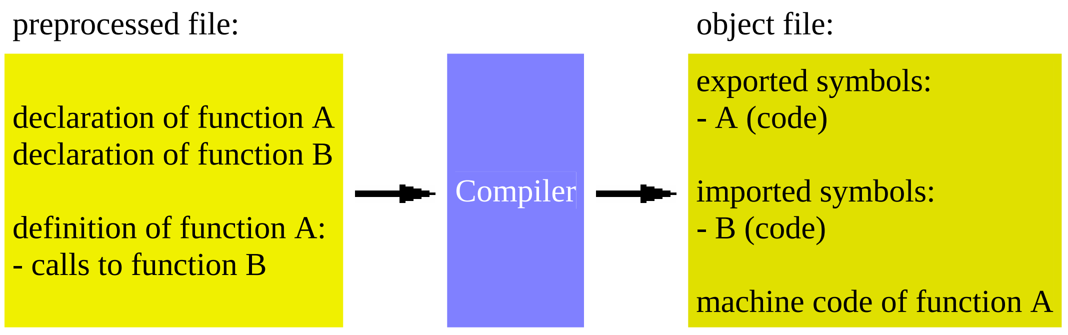
The preprocessor knows some default paths where to look for files to be included. If a header file exists in another path, this path must be passed to the preprocessor via a command line option.

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. Very helpful for seeing details of the build process is the `-v` option. It is common to all HP compiler frontends to switch on verbose mode:

```
$ cc -c -v HelloWorld.c
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
/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 -
ESconstlit -Oq00,al,ag,cn,Lm,sz,Ic,vo,lc,mf,Po,es,rs,sp,in,vc,pi,fa,pe,
Rr,Fl,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
```

The verbose output shows which environment variables are used and how they are set, and which arguments are passed to the compilation subprocesses. `/opt/langtools/lbin/cpp.ansi` is the preprocessor, `/opt/ansic/lbin/ccom` is the compiler.

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, an additional compiler option must be used: `+DA2.0W`. Alternatively, `+DD64` has the same effect.

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

Symbols from HelloWorld.o:

Name                    Value   Scope  Type    Subspace

C$10                |        40|static|data   |$LIT$
main                |         0|extern|entry  |$CODE$
printf              |          |undef |code   |
$
```

`nm` lists all symbol names and shows address, scope, symbol type and the location of the symbol in the object. `C$10` is an internally used symbol and not of interest here. `main` has an external scope which means it is globally accessible, a type of `entry` 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.

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.

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

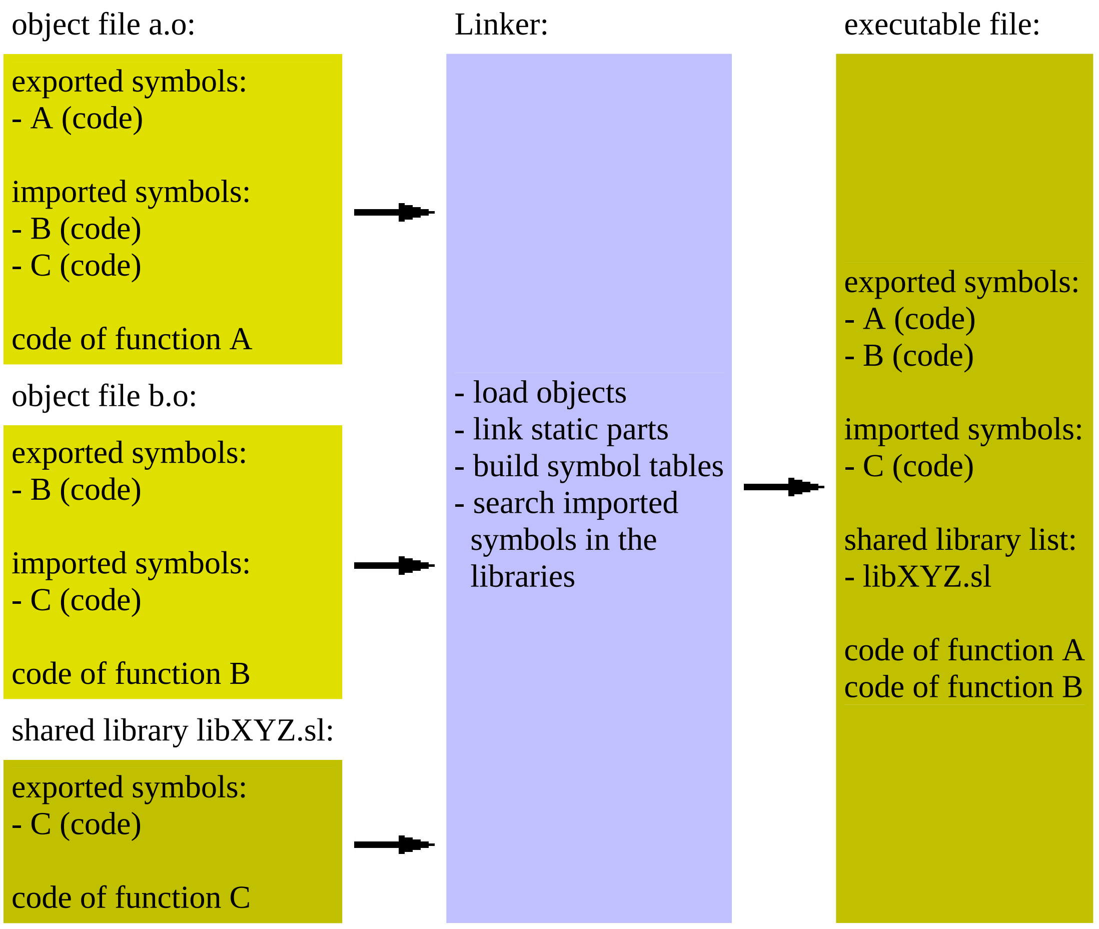
The linker `ld` reads all specified objects, puts them together into the binary file, calculates the final addresses of all symbols and updates references to them. Libraries can be linked to the binary using the `-lname` option. This causes the linker to search for a shared and an archive library (in 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 Libraries	**Archive Libraries**
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 initialisation and clean termination. If a compiler front end is used to link, it adds `crt0.o` automatically.

If a reference to a symbol cannot be resolved within the binary, it is searched in the list of shared libraries passed to the linker. If the symbol could still not be 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.

Objects that will be linked into a shared library must contain position independent code (PIC). Position independency means that the symbol address calculation can be done 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, which means that an address table is used for symbol resolution.

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. `ld` picks up the 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` which is added to the link command via the `-lc` parameter. A reference to `libc.sl` 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 on 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 creating an executable, the first object to link is always `crt0.o`. This file exists as a 32-bit object as well as a 64-bit object. Again, without adding special options to the link command, the frontend will pick the 32-bit version and thus make the linker generate a 32-bit executable. Using `+DA20.W` makes the frontend pick the 64-bit version in order to let the linker create a 64-bit executable.

Note that 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.

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

Symbols from HelloWorld:

Name                        Value   Scope  Type    Subspace

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

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

The `chatr(1)` command shows the library references of the program:

```
$ chatr HelloWorld
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
```

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`. `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

Symbols from /usr/lib/libc.2:

Name                        Value   Scope  Type    Subspace

   :
printf                |     959288|extern|entry  |
printf                |     959328|extern|code   |$CODE$
   :
```

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

Archive Bound Executables

The `nm(1)` and `chatr(1)` outputs for the archive bound program look different in many ways:

```
$ nm HelloWorld

Symbols from HelloWorld:

Name                        Value   Scope  Type    Subspace

   :
main                  |      13232|extern|entry  |$CODE$
   :
printf                |      14096|extern|entry  |$CODE$
   :
```

This time we see both `main` and `printf` resolved in the program. This is because the linker extracted the archive `libc` and linked the appropriate objects to the program. And we see only one entry per symbol. This is because there is no need to access them from outside, so stubs are not required.

```
$ 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
$
```

The `chatr(1)` output is somewhat shorter for an archive bound program. We have no library seach 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 the program should run on other systems independently of the runtime environment installed there. But it also introduces the problem that library updates will not affect those programs unless they are relinked. Also, several features that are only available in shared libraries on HP-UX are not available in archive bound programs, such as:

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

* 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:

```
$ tusc HelloWorld
execve("./HelloWorld", 0x7f7f08e8, 0x7f7f08f0) ... = 0 [32-bit]
utssys("HP-UX", 0, 0) ............................ = 0
open("/usr/lib/dld.sl", O_RDONLY, 04354) ......... = 3
read(3, "02\v010e0512@ \0\0\0\0\0\0\0\0\0".., 128) = 128
lseek(3, 128, SEEK_SET) .......................... = 128
read(3, "10\0\004\0\0\0( \002\0ac\0\0\0\0" .., 48) = 48
mmap(NULL, 131244, PROT_READ|PROT_EXEC, MAP_SHARED|MAP_SHLIB, 3, 0x9000) =
0xc0010000
mmap(NULL, 14696, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_SHLIB, 3,
0x2a000) = 0x7f7ec000
close(3) ......................................... = 0
getuid() ......................................... = 1524(1524)
getuid() ......................................... = 1524(1524)
getgid() ......................................... = 20(20)
getgid() ......................................... = 20(20)
mmap(NULL, 8192, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -1,
NULL) = 0x7f7ea000
sysconf(_SC_CPU_VERSION) ......................... = 0x214
open("/opt/graphics/OpenGL/lib/libogltls.sl", O_RDONLY, 0) = 3
fstat(3, 0x7f7f31c8) ............................. = 0
read(3, "0214010e0512@ \0\0\0\0\0\0\0\0\0".., 128) = 128
lseek(3, 128, SEEK_SET) .......................... = 128
read(3, "10\0\004\0\0\0( \0\014$ \0\010\0" .., 48) = 48
read(3, "80\0\001\0\0\0 9 2 4 5 ", 12) ........... = 12
lseek(3, 8192, SEEK_SET) ......................... = 8192
read(3, "058cy 10\0\0\0H \0\002D \0\0\001".., 112) = 112
mmap(NULL, 8192, PROT_READ|PROT_EXEC, MAP_SHARED|MAP_SHLIB, 3,0x2000) =
0xc0004000
mmap(NULL, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_SHLIB, 3,
0x4000) = 0x7f7e9000
close(3) ......................................... = 0
open("/usr/lib/libc.2", O_RDONLY, 0) ............. = 3
fstat(3, 0x7f7f32c8) ............................. = 0
read(3, "0210010e0512@ \0\0\0\0\0\0\0\0\0".., 128) = 128
lseek(3, 128, SEEK_SET) .......................... = 128
read(3, "10\0\004\0\0\0( \012a6D \0\010\0" .., 48) = 48
read(3, "80\0\0\v\0\0\004\0\0\0\0", 12) .......... = 12
```

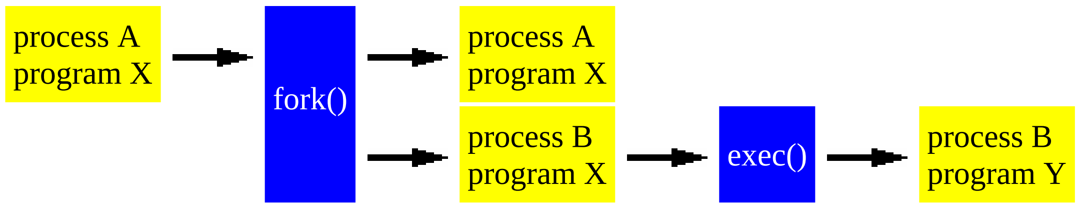
```
lseek(3, 290816, SEEK_SET) ...................... = 290816
read(3, "058cy 10\0\006f0\0\0I ( \0\0\002".., 112) = 112
mmap(NULL, 1224704, PROT_READ|PROT_EXEC, MAP_SHARED|MAP_SHLIB, 3, 0x47000) =
0xc0100000
mmap(NULL, 57344, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_SHLIB, -1, NULL) = 0x7f7db000
mmap(0x7f7d2000, 36864, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_SHLIB, 3, 0x172000) = 0x7f7d2000
mmap(NULL, 16384, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -
1, NULL) = 0x7f7ce000
close(3) ........................................ = 0
open("/usr/lib/libdld.2", O_RDONLY, 0) .......... = 3
fstat(3, 0x7f7f33c8) ............................ = 0
read(3, "02\v010e0512@ \0\0\0\0\0\0\0\0\0".., 128) = 128
lseek(3, 128, SEEK_SET) ......................... = 128
read(3, "10\0\004\0\0\0( \0\0$ e4\0\010\0" .., 48) = 48
read(3, "80\0\0\v\0\0\004\0\0\0\0", 12) ......... = 12
lseek(3, 8192, SEEK_SET) ........................ = 8192
read(3, "058cy 10\0\0\0\f\0\001ac\0\0\001".., 112) = 112
mmap(NULL, 12288, PROT_READ|PROT_EXEC, MAP_SHARED|MAP_SHLIB, 3, 0x2000) =
0xc0006000
mmap(NULL, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_SHLIB, 3,
0x5000) = 0x7f7cd000
close(3) ........................................ = 0
open("/usr/lib/libc.2", O_RDONLY, 0) ............ = 3
fstat(3, 0x7f7f34c8) ............................ = 0
read(3, "0210010e0512@ \0\0\0\0\0\0\0\0\0".., 128) = 128
lseek(3, 128, SEEK_SET) ......................... = 128
read(3, "10\0\004\0\0\0( \012a6D \0\010\0" .., 48) = 48
read(3, "80\0\0\v\0\0\004\0\0\0\0", 12) ......... = 12
lseek(3, 290816, SEEK_SET) ...................... = 290816
read(3, "058cy 10\0\006f0\0\0I ( \0\0\002".., 112) = 112
mmap(NULL, 1224704, PROT_READ|PROT_EXEC, MAP_SHARED|MAP_SHLIB, 3, 0x47000)
ERR#12 ENOMEM
close(3) ........................................ = 0
mmap(NULL, 3336, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -1,
NULL) = 0x7f7cc000
sigsetreturn(0x7f7cfd4e, 0x6211989, 1392) ....... = 0
sysconf(_SC_CPU_VERSION) ........................ = 0x214
brk(0x40001178) ................................. = 0
brk(0x4000316c) ................................. = 0
brk(0x40006000) ................................. = 0
ioctl(1, TCGETA, 0x7f7f0dd0) .................... = 0
Hello World!
write(1, "H e l l o   W o r l d ! \n", 13) ....... = 13
exit(13) ........................................ WIFEXITED(13)
```

Each section will pick a few system calls from the trace and discuss them. 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

When starting a new program, it usually is loaded into a new process that has just been created for it. The creation of a new process is called forking. The new process is called the child, the other is the parent. After forking, both processes are identical except of the process ID (PID), which must be unique on the whole system. To start the new program it is loaded and executed. In the child process. The new program replaces the old one which was a copy of the program in the parent process. The result is our old program continuing to run in the parent process, and the new program running in the child process:



The `tusc` output shows:

```
execve("./HelloWorld", 0x7f7f08e8, 0x7f7f08f0) ... = 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. Arguments like `0xnnnnnnnn` are addresses of data structures where the information can be found.

A new program can also be executed without prior forking, but that means the program that actually loads and starts the new program will cease to exist.

Load Shared Libraries

Every shared library is loaded in the same way, it is mapped into memory. 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 subsequent mappings 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 get in conflict with other processes.

Sharing libraries between multiple processes saves a lot of space on disks as well as in memory, because every shared library only needs to exist and to be loaded once, no matter how many processes need 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:

```
open("/usr/lib/dld.sl", O_RDONLY, 04354) ......... = 3
read(3, "02\v010e0512@ \0\0\0\0\0\0\0\0\0".., 128) = 128
lseek(3, 128, SEEK_SET) .......................... = 128
read(3, "10\0\004\0\0\0( \002\0ac\0\0\0\0" .., 48) = 48
mmap(NULL, 131244, PROT_READ|PROT_EXEC, MAP_SHARED|MAP_SHLIB, 3,0x9000)
= 0xc0010000
mmap(NULL, 14696, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_SHLIB, 3, 0x2a000) = 0x7f7ec000
close(3) ......................................... = 0
```

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 `dld.sl` loads the following libraries:

```
open("/usr/lib/libc.2", O_RDONLY, 0) ............. = 3
  :
open("/usr/lib/libdld.2", O_RDONLY, 0) ........... = 3
  :
```

`libc.2` and `libdld.2` are loaded, although `HelloWorld` only has `libc.2` in its shared library list. `libdld.2` is loaded because it is listed in the shared library list of `libc.2`:

```
$ chatr /usr/lib/libc.2
          :
         shared library list:
             dynamic   /usr/lib/libdld.2
          :
```

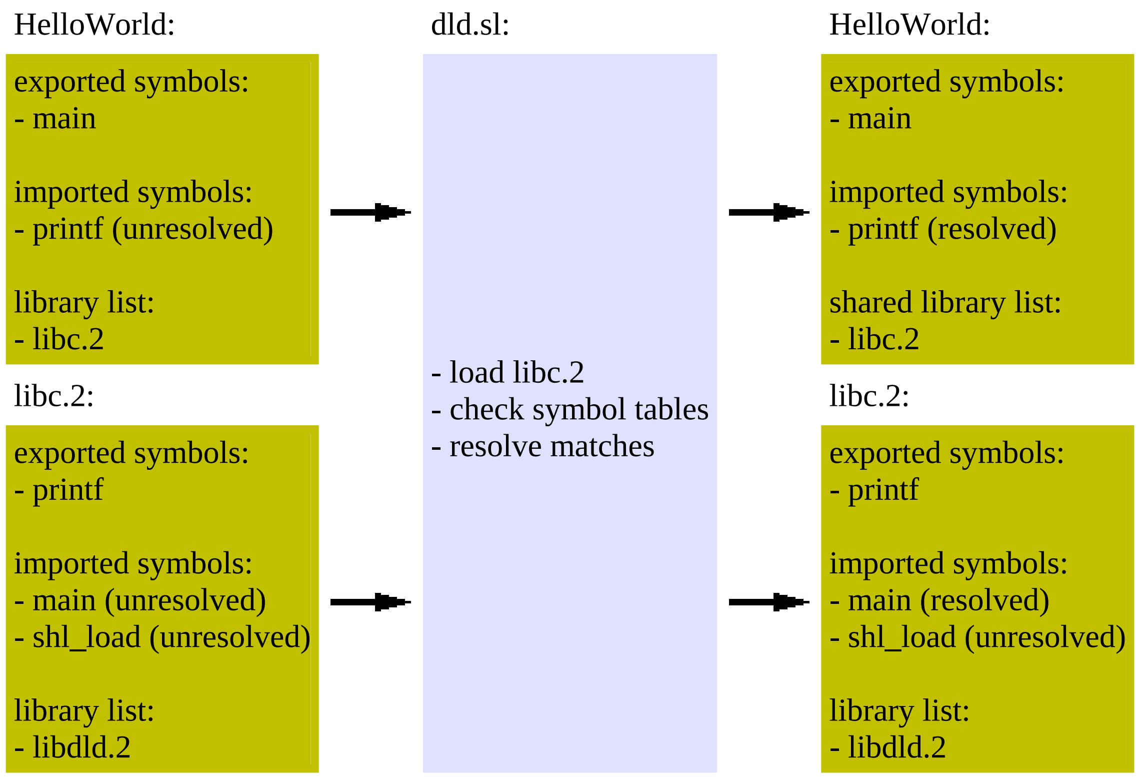
The `ldd(1)` tool can be used to list all libraries used by a program, without starting it, in the order they were loaded if the program was started.

Resolve Symbols

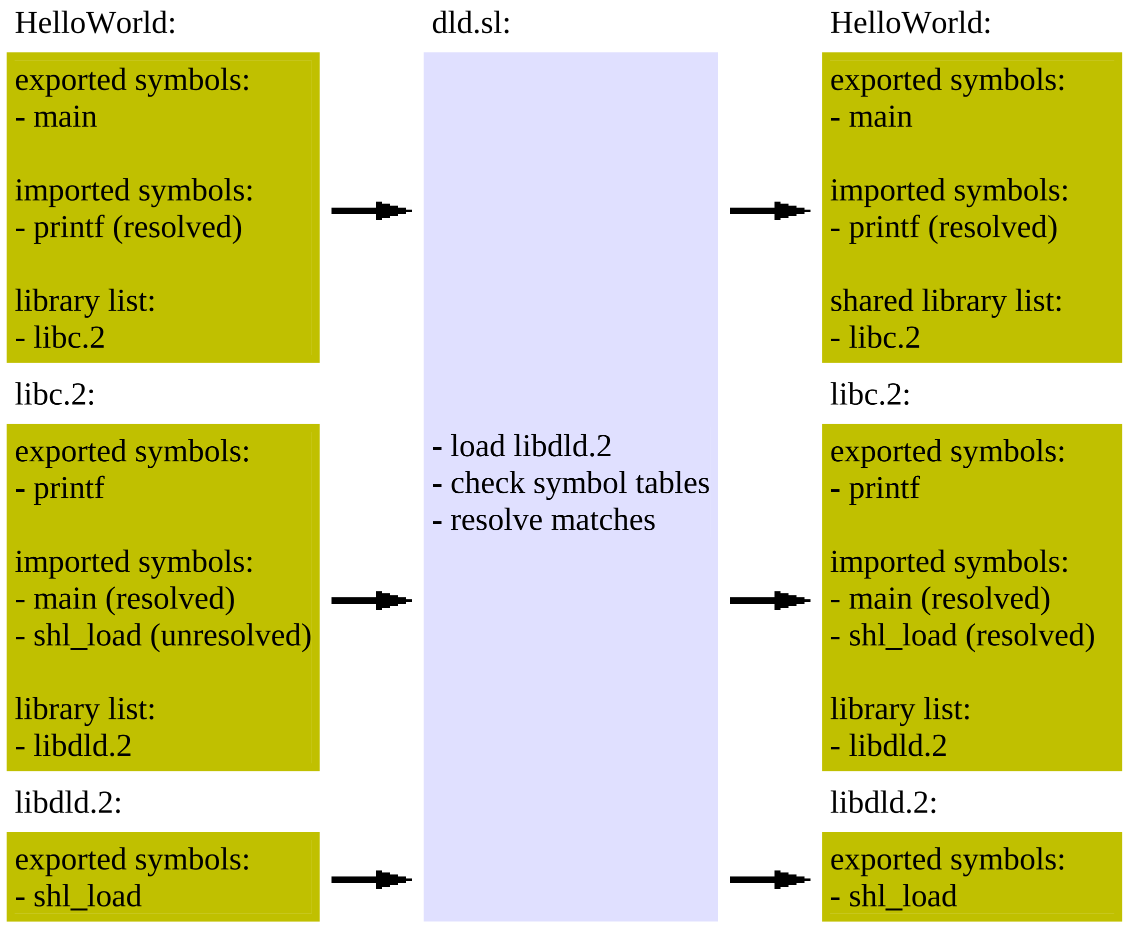
Each time one of the libraries is loaded, all symbols that are still marked as undefined in any symbol import table (of the executable and all libraries loaded up to now) are checked against the exported symbols of the current library. And the symbol import table of the loaded library is checked against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

For 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. So 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.2` is in its shared library list. The library is loaded, the symbol is found in it and can be resolved:



`libc.2` 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.2` has `libdld.2` in its shared library list so this one is loaded too:



libdld.2 contains the symbol shl_load so the reference from libc.2 can be resolved now too.

If there was no more library 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 with a corefile.

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, 0x7f7f0dd0) ..................... = 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 are 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 ouput 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. This is because system calls are logged near the end of the call, when its return code is available.

Threads

From the classic view of process management a process is an entity that will execute its instructions serially, one by one, and has its own private address space. The consequences are that if a process is waiting for some system resource, e.g. disk or network I/O, it can't do anything else so it is sent to sleep by the scheduler, and waken up again if the resource is available. And it cannot benefit from a multiprocessor system, because each process can only run on one CPU.

To increase performance of a single process, the threads model was introduced with HP-UX 10.X. Since then, a process still has its private address space, but instead of executing intructions itself, it becomes a container for one or more threads, and each thread has its own instructions to execute and shares the process resources with the other threads. Thus, if one thread is waiting for system resources, other threads can still be executed so the process is not forced to be idle.

On HP-UX 10.X there are only user threads (1xN), which means that the kernel has only one thread per process, but the program can handle multiple threads. 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. 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 managed now by the kernel, and if the process is running on a multiprocessor system, the kernel can execute multiple threads on different CPU's simultaneously. There exists a POSIX standard for threads, and this standard is implemented in HP-UX 11.X. That's why the kernel threads are also known as POSIX threads, and the appropriate library for these threads is `libpthread.sl`. On HP-UX 11.X the CMA threads are still available for downward compatibility reasons, but you can't use both CMA and POSIX threads together within one program, both models are not compatible.

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 11.22 (Itanium) and 11.23PI (PA-RISC and Itanium).

Program Termination

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.
- Most signals (see `signal(5)`) will end a process abnormally. Some cause the generation of a corefile, others don't. Signals can be sent by the kernel, by the process itself or by other processes. In the first case the kernel has most likely detected a program failure. The corefile is helpful to debug why a 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()`
- receive a signal

Although we do not call `exit()` explicitly in the `HelloWorld.c` source code, the system call `exit()` is executed automatically after reaching the and of `main()`, to terminate the process regularly:

```
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 it is often required to analyze the address space of a process, which contains all code and data owned by the process. This page will give an introduction to this theme. For a detailed description of HP-UX memory management see the "HP-UX Memory Management White Paper" and the "HP-UX Process Management White Paper" in the files

- `/usr/share/doc/mem_mgt.txt`
- `/usr/share/doc/proc_mgt.txt`

on every HP-UX system.

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 the address space is divided into 4 quadrants with 1 GB and different purposes for each:

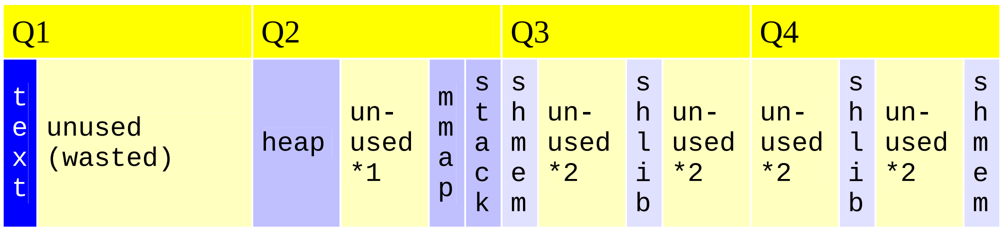
Quadrant	Address Range	Usage
Q1	`0x00000000 - 0x3fffffff`	text
Q2	`0x40000000 - 0x7fffffff`	private data
Q3	`0x80000000 - 0xbfffffff`	shared text/data
Q4	`0xc0000000 - 0xffffffff`	shared text/data

In the address range of quadrant 1 the program code is stored. 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 is unused. All data in the first quadrant is read-only. A process is not allowed to modify its code, because HP-UX allows code sharing between multiple processes. If the same executable is run multiple times, its code must only be loaded once.

The second quadrant 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.

The third and fourth quadrant are reserved for shared text and data. Shared librarie code is mapped into these quadrants, and shared memory is allocated here.

This is an examplary address space usage of a process:



`shmem`: shared memory segment
`shlib`: shared library
`*1`: can be used to increase the heap or to map 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.

Private Data

A process has three methods to store private data. It can store it in allocated space on the process heap, save it on the stack, or in privately mapped regions. All three methods must share the second quadrant, but follow different rules of memory allocation. Every program has a heap and a stack, but it will only have mapped regions if it was designed to use them.

The heap starts at the beginning of the quadrant. 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. If a memory region is not needed anymore, it can be freed (deallocated) again. But 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 functions `malloc()` and `free()`.

The stack is a reserved area at the end of the data quadrant. 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 at the end of the second quadrant when the process is started. 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. This can be achieved with the libc functions `mmap()` and `munmap()`. Per default, new regions will be mapped at the highest possible address. The memory allocated by a mapped region will be returned to the system after it has been unmapped.

Multithreaded Programs

Each thread in a multithreaded program needs its own stack. The initial thread, which was started when the program was executed, uses the process stack. Whenever a new thread is started with `pthread_create()`, a new private region is mapped and used as the thread stack. This area is unmapped when the thread exits.

Limitations

There are several kernel parameters that delimit the amount of memory a process can allocate:

- `maxtsiz`: Maximum size of text (code)
- `maxdsiz`: Maximum amount of private data
- `maxssiz`: Size of the stack

The default setting of `maxtsiz` (64 MB) usually is uncritical because most programs typically don't need more than that for their code.

This is different for `maxdsiz`. It sets the maximum private data a process can allocate on the heap. There are a lot of applications out there that want to allocate many hundreds of MB so the default setting of 64 MB after a standard HP-UX installation is often too small. Even the linker can easily hit that limit while building a large executable.

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 in the second quadrant. If the maximum amount of allocatable memory in a program is much less than 1 GB, check your `maxssiz` setting.

Setting these parameters to a value larger than 1 GB does not make sense, because the size of text and data is limited to 1 GB by the quadrant size.

These parameters apply only to 32-bit processes, but there are equivalent kernel parameters for 64-bit processes:

- `maxtsiz_64bit`
- `maxdsiz_64bit`
- `maxssiz_64bit`

There is no limit for mapped regions, other than the available address range.

The size of the process heap will also be limited by the posix shell built-in command `ulimit -d`. But it can only set limits lower than `maxdsiz`. Note that the default value for `ulimit -d` is the minimum of `maxdsiz` and `maxdsiz_64bit`, so increasing `maxdsiz` might not give more memory to a 32-bit process if `maxdsiz_64bit` is too small which will keep the `ulimit -d` value down.

Shared memory and shared libraries are not limited on a per-process base, because they cannot be associated to a single process. Any process can attach to them. The third and fourth quadrants are dedicated to shared data and text and the size of both is the only limit. That's why programs can allocate up to 2.75 GB of data when using both private and shared quadrants (The missing 250 MB are reserved for the kernel). But handling of shared memory works differently and takes a little more effort.

Beyond The Limits

Nowadays it often happens that programs need to address more than 1 GB of heap. For 64-bit programs this is no problem because their quadrants are much larger than that, so the rest of this chapter applies only to 32-bit processes.

Because it nomally takes more than just a recompilation with `+DA2.0W` to create a 64-bit version of a program, also 32-bit programs need to access more than 1 GB of data.

Fortunately, the 1 GB limits set by the quadrant size can be worked around. There are a few possibilities to modify the way a program uses the quadrants.

Allow Private Data In Q1

At link time you can select the type of an executable. Per default, `ld` creates `SHARE_MAGIC` executables which make use of the quadrants as described above. When passing the option `-N` to 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:

Q1		Q2			Q3				Q4			
text	heap	heap	un-used *1	stack	shmem	un-used *2	shlib	un-used *2	un-used *2	shlib	un-used *2	shmem

`shmem`: shared memory segment
`shlib`: shared library
`*1`: can be used to increase the heap or to map private regions
`*2`: can be used for additional shared memory segments or shared libraries

As a side effect, the program code cannot be shared anymore between multiple program instances. Because there is private data in the first quadrant, the whole quadrant must become writeable, and thus must be private.

If we have a `SHARE_MAGIC` executable which needs more than 1 GB private data, and we cannot relink it, the address space of the first quadrant is lost.

Allow Private Data In Q3 and Q4

The chatr(1) command can be used to change the usage of the third and fourth quadrant. It can tell the program to use these quadrants for private data:

```
Chatr [ +q3p enable|disable ] [ +q4p enable|disable ] prog
```

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

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

When changing both quadrants to be private, the process cannot request shared memory from the kernel anymore. It is of course still possible to load shared libraries, but they will be loaded as private text and cannot be shared between multiple processes anymore.

Note that it is not supported to add only Q4 to the private data of a program.

Changing the usage of the third and fourth quadrant is possible for either `SHARE_MAGIC` and `EXEC_MAGIC` executables, so you can allow more than 1 GB of private data to programs without relinking them. This feature is called “Large Data Space”.

Using all possibilites together allows a process to use nearly the complete 4 GB address space as private memory. Remember that `maxdsiz` and `ulimit` must be set high enough:

Q1		Q2	Q3		Q4			
Text	heap	heap	heap	stack	Heap	un-used *1	shlib	shlib

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

Of course the total amount of memory the whole system can allocate is limited by the amount of virtual memory (swap space). So if there is not enough available, all the tricks are useless.

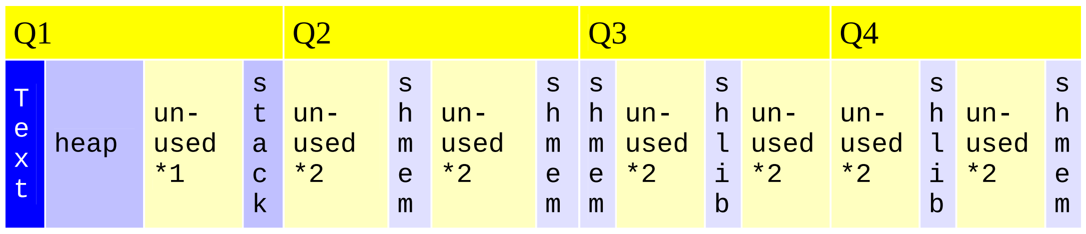
On HP-UX 11.0 there are several patches required in order to enable the large data space feature. These patches are `PHKL_20222-PHKL_20229` (most of them are meanwhile superseded). HP-UX 11.11 and later have this feature implemented within Core-OS.

Allow Shared Data In Q2

While the above steps provide more private data space to a process, there might be other processes which need more shared memory. 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 pushed up to 2.75 GB by changing Q2 from private to shared usage:

```
$ chatr –M prog
           :
         normal SHMEM_MAGIC executable
           :
```

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



shmem: shared memory segment
shlib: shared library
*1: can be used to increase the heap
*2: can be used for additional shared memory segments and shared libraries

Types Of Executables

Depending on the usage of the quadrants the type of the executable is different. The first line of the `chatr(1)` output shows the type of executable:

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

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++
Pascal	technical, study	
Fortran	numeric calculations	http://www.hp.com/go/fortran
Cobol	financial applications	http://www.hp.com/go/cobol
Java	Web based, platform independent	http://www.hp.com/go/java

On HP-UX there are development environments available for all commonly used programming languages. But a development environment is more than just a compiler. To build an application, compiler and linker are required, and to run it a runtime environment is needed. In addition HP provides 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. They allow to write code and to compile and test it without leaving the editor. 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 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 would be immense compared to this single line. It would become even more complicated to program this in one of the other languages. But it is not necessary to do this, because someone has already done it: The developers of the shells and other interpreters.

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		
ANSI C	B3899BA	B3901BA	`/opt/ansic/bin/cc`	`.c`
ANSI C++	B3911DB	B3913DB	`/opt/aCC/bin/aCC`	`.c, .C, .cpp, .cxx`
Fortran90	B3907DB	B3909DB	`/opt/fortran90/bin/f90`	`.f, .F, .f90, .F90`
Pascal	B3903DB	B3905DB	`/opt/pascal/bin/pc`	`.p`
Object Cobol	B2430EB	B2433EB	`/opt/cobol/bin/cob`	`.cob, .cbl`
Server Express	B8840BB		`/opt/lib/cobol/bin/cob`	`.cob, .cbl`

These compilers are **not** shipped with the HP-UX core. They are separate products which are codeword protected on the application CD's and must be purchased separately.

Both cobol products, Object Cobol and Server Express, make an exception here. The Object Cobol developer bundle is not codeword protected. It can be installed freely but before it can be used the flexlm licensing mechanism must be activated and a license must be installed. See the installation problems section for details.

Server Express is the new cobol development environment for HP-UX which supersedes Object Cobol. It provides additional features, e.g. 64-bit support (since version 2.0), and is shipped on a separate CD. It uses the apptrack licensing mechanism which is explained in short in the installation problems section.

Object Cobol and Server Express are not HP products. In fact they are products from Microfocus. In the past these products could be purchased from HP. This is no longer possible, and also support for them will end in 2005. Customers who want to use cobol must get them directly from Microfocus or another Cobol Vendor. Details on that can be found on the HP website at

- http://www.hp.com/go/cobol

The HP-UX core is shipped with the so called bundled C compiler `/usr/ccs/bin/cc` which is often misunderstood to be capable of 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. K&R is an old C standard which is not used anymore by todays programs. Customers who want to compile ANSI C programs will 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          Auxiliary Optimizer for HP Languages
DebugPrg               Debugging Support Tools
```

```
WDB                   HP Wildebeest (HP WDB) Debugger
WDB-GUI               GUI for the HP WDB Debugger
```

The preferred debugger on HP-UX is WDB, which is based on the Gnu debugger gdb. Again, cobol (Object Cobol and Server Express) makes an exception here. It has its own debugging environment called "Animator" and does not include the debugging products.

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

```
AudioDevKit           HP-UX Audio Developer Kit
CDEDevKit             CDE Developer Kit
ImagingDevKit         HP-UX Developer's Toolkit - Imaging
X11MotifDevKit        HP-UX Developer's Toolkit - X11, Motif
```

The HP-UX Developer's Toolkit is not shipped with the other compilers. HP provides the HP-UX Developer's Toolkit for free on

- www.software.hp.com

Its product number is B3394BA.

Upgrades To The HP Products

Upgrades of HP compilers come in two different forms, either as a patch, or as a full (base, application) release. Patches can only be installed on the last 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. The latest versions of the compilers are listed on the HP website:

Product	**URL**	**Name of the Link to follow**
ANSI C	http://www.hp.com/go/c	Patch release history
ANSI C++	http://www.hp.com/go/c++	Latest Versions and Patches
Fortran90	http://www.hp.com/go/fortran	Release history

Development toolkit upgrades always come as patches, at the time of this writing, these are:

Product	**UX 11.00**	**UX 11.11**	**UX 11.22**	**UX 11.23**
X/Motif DevKit	PHSS_30261 (32bit) PHSS_29370 (64bit)	PHSS_31001	PHSS_29374	n/a
CDE DevKit	PHSS_29738	PHSS_30790	n/a	n/a
AudioDevKit	PHSS_24649	PHSS_28548	n/a	n/a

Note that the development kit patches usually depend on a similar patch level of their runtime environment.

The compiler base releases also contain a current version of the WDB debugger, but WDB can also be upgraded separately by downloading the latest version from

- http://www.hp.com/go/wdb

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. See gnu.org (non HP) for details.

HP provides a free HP-UX port of the Gnu C and C++ compilers `gcc` and `g++`. They can be downloaded from

- http://www.hp.com/go/gcc

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

Related Products

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

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

Upgrades to these filesets are always available as patches which can be obtained via the usual ways. Upgrades for several other development related filesets are also available as patches:

Patch	UX 11.00	UX 11.11	UX 11.22	UX 11.23
ld(1) and linker tools	PHSS_30967	PHSS_30968	PHSS_32062	PHSS_31849
millicode library	PHSS_26320	PHSS_26321	PHSS_32067	PHSS_30819
High-level optimizer	PHSS_32004		PHSS_30024	PHSS_30849
Pxdb	PHSS_27983		n/a	PHSS_30820
libc header file	PHCO_26111	PHCO_25569	n/a	n/a

Runtime Environments

Normally 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	Comes with
any	`libc.sl/.so`	HP-UX core
Fortran, Pascal, C++	`libcl.sl/.so`	HP-UX core

aC++	`libCsup.sl/.so`
	`libCsup_v2.sl/.so`
	`libstd.sl/.so`
	`libstd_v2.sl/.so`
	`libstream.sl/.so`
	`librwtool.sl/.so`
		HP-UX core

Object Cobol	`/opt/cobol/cobdir/*`	B2432EB (s700), B2435EB (s800) (Application CD)
Application Server	`/opt/lib/cobol/lib/*`	B8841BB (user license) B8842BC (processor license) (separate Server Express CD)

`libc.sl/.so` is the central library of every Unix. 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. The programming languages that are not listed here, don't need additional runtime libraries (e.g. C, Pascal).

`libcl.sl/.so` is the language support library. It provides features that are special to the different languages, e.g. fortran I/O, pascal Try/Recover, and other useful things like stack unwind functions. Since HP-UX 11iv5, 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 11iv5 release notes which are available at the HP website and on any system running HP-UX 11iv5 or later:

- http://docs.hp.com/hpux/onlinedocs/B9106-90015/B9106-90015.html
- `/usr/share/doc/11iV1_5RelNotes.txt`

Application Server is the runtime environment for Server Express (cobol). It is the only runtime environment that requires a license. It knows two types of licenses. One is bound to the number of processors to run on in parallel, the other is bound to the number of users or cobol programs which can run simultaneously. See the installation problems section for details.

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.

Except of Application Server, the runtime environments don't need to be purchased.

Runtime environment upgrades always come as patches (except of the cobol runtimes):

Patch	UX 11.00	UX 11.11	UX 11.22	UX 11.23
Libc	PHCO_29956	PHCO_31903	PHCO_29329	n/a
LIBCL	PHSS_28302	PHSS_28303	PHSS_27107	n/a
Libunwind	n/a	n/a	PHSS_32064	n/a
HP aC++ -AA Runtime	PHSS_28879	PHSS_28880	PHSS_32065	PHSS_31852
X/Motif Runtime 32-bit	PHSS_30260	PHSS_31000	PHSS_30264	PHSS_31834
X/Motif Runtime 64-bit	PHSS_29369			
CDE Runtime	PHSS_30668	n/a	n/a	n/a
AudioSubsystem	PHSS_27192	PHSS_28547	n/a	PHSS_31840

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

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 nothing else than a special way to invoke the JVM which knows how to translate java 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 class[1)] 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. 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 which is very similar to use as `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

In a java application there must be one class that contains a method (function) called `main()`, which is called first when the application is started. To execute a java program, the JVM must be executed and the class or jar archive that contains this method must be passed 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 executing the appropriate machine code for each bytecode instruction. But interpreting bytecode 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 machine code while the application is running. Once a method has been compiled, the JVM calls the compiled version instead of interpreting the bytecode again.

Products

For java there exists a software development kit (SDK) and a runtime environment (RTE). The SDK includes the RTE so if the SDK is installed, the RTE does not need to be installed in parallel. At the time writing HP provides 4 release lines of java:

Major Version	**Product Number**		**Installation Path**
	s700	**s800**	
Java 2 Version 1.2	B8110AA	B8111AA	`/opt/java1.2`
Java 2 Version 1.3	B9788AA	B9789AA	`/opt/java1.3`
Java 2 Version 1.4	T1456AA	T1457AA	`/opt/java1.4`
Java 2 Version 5.0 Beta	n/a	n/a	`/opt/java5`

As this table shows these release lines are installed in different directories, so they can be installed in parallel without interfering each other.

There are different subrelease lines for Java 1.3 and 1.4:

- 1.3.0, 1.3.1
- 1.4.0, 1.4.1, 1.4.2

Versions 1.3.0, 1.4.0 and 1.4.1 are obsolete. Customers should upgrade to the latest 1.3.1 resp. 1.4.2 version.

For now, Java 2 Version 5 is available only as a beta version. Java 2 Version 1.2 will become obsolete as soon as Java 2 Version 5 is officially released.

Up to version 1.3 all java versions are 32-bit only. Release 1.4.0 was the first to provide a 64-bit JVM. To invoke the 64-bit JVM, the java command line option `-d64` must be used.

Java SDK and RTE are free. They can be installed from an application CD or downloaded from the HP website, which always holds the latest and a few previous releases. There also is an ftp site that provides even more java versions, including the last few Java 1 versions:

- www.hp.com/go/java
- ftp://ftp.hp.com/pub/gsy/

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://www.hp.com/products1/unix/java/infolibrary/patches.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 of course 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 rely on a shell to execute programs it does 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`
Restricted	`/usr/bin/rsh`
Korn	`/usr/bin/ksh`
Bourne	`/usr/old/bin/sh`
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`, can be downloaded from the HP-UX porting center websites.

Shell updates are always available as patches:

Shell	UX 11.00	UX 11.11	UX 11.22	UX 11.23
POSIX Shell	PHCO_31160	PHCO_31815	PHCO_29748	PHCO_31562
ksh	PHCO_27418	PHCO_30316	n/a	PHCO_31552
csh	PHCO_28346	PHCO_30533	PHCO_29623	PHCO_31543

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. To start a shell with a leading "-" the `exec(2)` function must be used. A login shell cannot be started from the command line.

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:2eRyvWzYVvtzc:0:3:,,,:/:/sbin/sh
```

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

Feature	POSIX	Bourne	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` activates the POSIX shell, which is very similar in many respects to the Korn shell but has some additional enhancements. If a command is invoked the POSIX shell searches first 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.

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).

For special characters, like german umlauts, the shell has to load the character set by setting the following variable:

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

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

The POSIX shell is the default shell as of HP-UX 10.20. All the startup scripts use this shell. Manual page is ‘`man sh-posix`’.

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 menu 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”.

Bourne Shell

The Bourne shell is the traditional UNIX shell usually invoked by `sh`, but due to POSIX conformity it resides in `/usr/old/bin/sh` now. Its functionality is very limited, so it is not used very often. It has a modern and enhanced follow-up, the Bourne-again-shell or bash, which is the standard shell on many Linux systems.

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.

C Shell

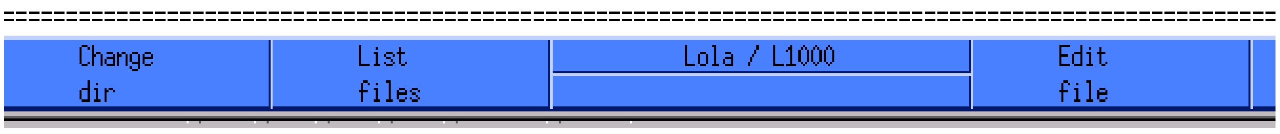
The C shell allows C-style scripts. This does not mean it can interpret C programs, but the syntax of C shell scripts is very similar to C programs. Other features are a command history buffer and job control facilities.

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.
```



For more details on shells go to the *Shells User's Guide* for all HP-UX releases:

- http://docs.hp.com/hpux/onlinedocs/B2355-90046/B2355-90046.html

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 seperate file with the `–f` option.

`perl`

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

Perl often needs additional modules, e.g. for jobs like database access. These modules are in fact shared libraries which are loaded when perl finds the appropriate instructions in the `perl` script. These modules are of course platform dependent.

In HP-UX 11.0 and 11.11, fileset `OS-Core.Q4` provides a `perl` executable in the file `/usr/contrib/bin/perl`. At least on 11.0 this version (4.0) does not match the requirements of todays perl scripts and should not be used other than for Q4 kernel debugging. `perl` for HP-UX should be downloaded from

- http://www.software.hp.com/cgi-bin/swdepot_parser.cgi/cgi/displayProductInfo.pl?productNumber=PERL

The Interpreter Line

Normally a script cannot be executed directly by the system, because an interpreter is required. So 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
```

If a script has execute permissions, 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 to execute by its magic number. Executables have a specific magic number, 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.

If a script has no interpreter line and thus has no magic number, it is assumed to be a shell script, and a POSIX shell will be started to execute it. More general, every file that has execute permissions but no magic number is executed by passing it to the POSIX shell.

Tools

The tools to look at in this workshop can be divided into three types of things to do with them:

1. Development Utilities
2. Static Information
3. Runtime Analysis

In the Development Utilities section we will look at build process management and version administration.

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. At link time they can be passed to `ld` like 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. `ar(1)` also creates a symbol table inside the library that lists the symbols from all objects. `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 an executable and multiple shared libraries. Each of them can be linked from multiple objects, and each object can be compiled from multiple headers 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) was designed to manage dependencies in this way. It compares existence and timestamps of files to determine which parts need to be rebuilt. The developer must create the 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 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. Its dependents are obj_a.o and obj_b.o, and a possible rule to build myprog would be to link the objects. The objects themselves are targets too, and their dependents are the appropriate source files and the headers that are included in the source files. The normal command to create objects from sources is to invoke a compiler.

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, CVS is a freeware tool and is available on the porting center. There also is an RCS from Gnu. This version is also available on the porting 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 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 a.out core
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
```

Especially the output for the corefile is interesting, because it provides two informations interesting for a first analysis on what happened. It tells you:

- Which program generated the corefile
- Why the corefile 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 returned by the `what(1)` command. A what-string can be added to any file. It is simply a character string which is marked 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 is everywhere, even if it dumps core:

```
$ 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 corefile 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`. The library what strings returned from a corefile are very helpful when the corefile 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 resources where the dynamic loader expects paths to search for shared libraries:

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

The embedded path can only be defined at link time with the `+b` linker option, and is stored in the executable.

`SHLIB_PATH` and `LD_LIBRARY_PATH` can be set at runtime, but if they are really used is determined by a flag in the executable. The default is that they are disabled. Using these variables must explicitly be enabled, either by linking with the `+s` linker option, or by issueing the following `chatr(1)` command:

```
$ chatr +s enable a.out
a.out:
              :
         shared executable
         shared library dynamic path search:
             SHLIB_PATH      enabled   second
             embedded path   disabled  first  Not Defined
              :
```

The default path is generated by the linker and is stored in the shared library list. This is the path where the library was found at link time. The default path is always searched last.

Using `SHLIB_PATH` or `LD_LIBRARY_PATH` can be useful 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.

So we can e.g. check if our `SHLIB_PATH` setting works. Assuming we have enabled `SHLIB_PATH` in `a.out` and set it 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/libdld.2 =>    /usr/lib/libdld.2
        /usr/lib/libc.2 =>      /tmp/libc.2
```

`ldd(1)` belongs to the linker tools.

`nm(1)`

Use this command to display 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 occurences 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.

`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
`-compunit`	Prints a paragraph for each object, can be used to find out which compiler and which flags were used to compile.
`-symbols`	Prints all symbols and symbol references. This is similar to `nm(1)`, but it still works even if the file was stripped (see `strip(1)`).
`-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 explaination of the `odump(1)` outputs see `a.out(4)`. `odump(1)` can only be used with 32-bit 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 the `write` closer to the `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. There is another, native HP debugger, `dde(1)`. But the development of `dde(1)` has been discontinued, and it is no longer shipped, so `gdb(1)` is to be preferred.

`gdb(1)` is free and comes with all HP compilers. It can also be downloaded from the web at

- http://www.hp.com/go/wdb

The purpose of a debugger is to allow you to see what is going on in a program while it executes. 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.

If a program aborted with a corefile you can also do a post-mortem analysis and try to find out why the program died.

gdb can be started as follows:

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

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

When debugging a corefile, it is a good practice to give the corefile another name. Else, whenever some other process aborts with a corefile, the corefile 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. 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 manpage. `gdb(1)` also provides an online help. Simply type `help` at the `(gdb)` prompt and follow the instructions. A detailed debugging tutorial can be found on

- http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,1677,00.html.

A few simple things shall be explained here. 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)
```

`gdb` loads the program but does not execute it yet. First we print out the source program. This is of course optional and doesn't make sense for large source files, but it helps keeping the overview here. 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()`.`gdb` always prints the next statement when it stops execution.

```
(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()` and a debuggable `libc`, the `step` command would jump into `printf()` and display the first executable statement of this function. But we haven't so we find ourselves at the end of `main()` after returning from `printf()`. 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()`:

```
(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 this function is in `libc` which has no debug info. That's why `gdb` can only determine the memory address of the breakpoint and not the source file and line number. When rerunning 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 far 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, a commonly made error here is to get the stack trace from the wrong thread. Therefore, before getting a stacktrace, you should 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 only have 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 procedure (function), and in which part of the process it is. If debug information is available for the function, source file and line number are printed. The address of frame #0 is the location where the program stopped. When debugging a corefile, this is the location at which the program died. For other frames it shows the address of the instruction that called the function in the previous frame. The procedure backtrace from the HelloWorld program reads as follows:

- Frame #0: The program stopped at address `0x7f7a5378` in `printf()` which is contained in `/usr/lib/libc.2`.
- Frame #1: `printf()` in frame #0 was called from `main()` at address `0x2768` which belongs to line 4 of the source file `HelloWorld.c`.

```
 (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 corefile and start debugging. You must make sure that the debugger uses the same versions of the shared libraries as at program runtime. If you use the wrong library versions, the debugger will most probably print only garbage.

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 corefile. 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 ever possible, debug a corefile on the system where it was generated.

`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 HP-UX labs at

- http://rpm-www.rose.hp.com/~chrisb/ttrace.html (HP internal)

Because the system calls are defined within the kernel sources, they are identical for the whole operating system. 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()` uses this file descriptor and sends the 13 given bytes to our 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` invokation will only print the listed system calls. This combination might be useful to see the file access of a process.

`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).

Detailed information and downloads are available at

- http://oceanie.grenoble.hp.com/georges/kmeminfo.html

This tool is not officially available but can be sent out to customers on request.

`adb(1)`

When analyzing a corefile with `gdb`, `adb` can provide additional, helpful information. E.g. to find out if an 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.

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

lsof

`lsof` is a tool that lists information about open files on a system. It does not work on a per-process base, it asks the kernel to return all open files. So if you want to know the files opened by a certain process you can either filter the output, e.g. using `grep(1)`, or use the `-p` option of `lsof`. Besides regular files it also lists open network connections, pipes etc.

`lsof` is a freeware tool that can be downloaded from the porting center.

Solving Common Problems

It is rather seldom that there are common problems that have standard solutions in the area of software development. 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 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 common approaches, 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.

A lot of 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: compilers, the linker, java, debuggers or runtime libraries for programming languages
- HP-UX WTEC: libc and other HP-UX runtime libraries, shells and unix commands
- X/Graphics WTEC: X11/Motif/3D programming 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, and every patch can be installed on the base HP-UX installation.

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

- Java runtime environment
- Java development toolkit
- Object Cobol runtime environment
- wdb/gdb debugger

The Object Cobol runtime environment can be installed from the application CD, there is no codeword required. The other 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, but a codeword is required. These products can also be downloaded from the ITRC's Software Update Manager, 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 comes a new base release with the application CD, and in between patches are released that can only be installed on the recent base version.

The Object Cobol compiler always comes as full version on the application CD. Server Express is available on a separate CD. There are no cobol patches. So to upgrade these compilers to the latest version, simply get the latest CD and install from there. Note that the Cobol products can no longer be purchased, and support for them will end in 2005. Customers who want to continue to use Cobol products on HP-UX can purchase it directly from the manufacturer.

For the other compilers, to learn which is the latest version, always check its 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` and `f90 +version` return the version number. `cc -V` prints the versions of the subprocesses invoked, but you must compile something to make `cc` 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 number returned by `swlist(1M)` sometimes is different from the what strings, because both might use a different versioning.

To find out the version of the installed aC++ compiler, do:

```
$ uname –r
B.11.00
```

```
$ 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 11.0 system we have installed version A.03.27. At the time of this writing, the latest version for 11.0 is A.03.57, and the most recent base release is A.03.55. So to upgrade to the latest version we need to install A.03.55 from a current application CD first, and then PHSS_30765.

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 this one first.

Cobol Installation/Upgrade

The Cobol compiler/runtime installation itself is quite easy. It can be installed from the application CD (Object Cobol) or from a separate CD (Server Express/Application Server) without needing a codeword. Note that the cobol development environments always contain the runtime environment. They must not be installed together on one system.

For Cobol to work properly, a few environment variables must be set:

```
COBDIR=/opt/lib/cobol/cobdir     # Server Express
# COBDIR=/opt/cobol/cobdir       # Object Cobol
SHLIB_PATH=$COBDIR/lib
PATH=$PATH:$COBDIR/bin
```

After the installation of Object Cobol (development), Server Express and Application Server, the license mechanism must be set up and a valid license must be installed. The licensing is described in detail below.

Upgrading cobol always means to install a new version from the appropriate CD. Previously installed licenses will stay valid.

Installing or upgrading Object Cobol runtime is simply installing the appropriate product from the application CD. No license is required.

Object Cobol Licensing

Object Cobol uses Flexlm which is a network based licensing system. This means you can choose any system as your license server and every development system can obtain licenses from it, as long as it is in the same network domain. Usually one of the development systems will serve as the license server.

To install flexlm two files are needed:

- `Hpmfocdsini:` Installation Script
- `hpmfocdsini.cpio:` Installation Data

After Installation of Object Cobol these files can be found in `$COBDIR`. If you want to install flexlm on a system where no Object cobol is installed, copy these files to that system.

Change to the directory where these files are, run `hpmfocdsini` (as root) and answer all questions (most of the time, simply press enter). The flexlm installation is complete now.

If you have no license file (named `license.dat`) yet, write down the host ID and hostname that have been printed by the `hpmfocdsini` and send an email to the HP licensing center (europe_codeword@hp.com) with the following contents:

- Hostname
- Host-ID
- System handle or Order Number
- Customer Company
- Contact Name
- Email

Provided your contract data is correct (must contain product number B2433EB) they will send you the license file within 24 hours as a reply.

Copy the 3 or 4 lines of that mail marked as your license file into the file `license.dat` in the directory `/usr/local/flexlm/licenses/`. If this path does not exist, create it manually. Do that on every system where either flexlm or Object Cobol should run. The license file should look like this:

```
SERVER hprtdu94 78261A71 7788
DAEMON HPCUPLANGS /usr/local/flexlm/daemons/HPCUPLANGS
INCREMENT hpmfcobol HPCUPLANGS 1.000 1-Jan-0000 3 0CD3160F360B7F927B2A\
             OVERDRAFT=1 ck=21
```

In the SERVER line you should find the hostname and host ID of the system where you have installed flexlm. Finally, start the license daemon `/usr/local/flexlm/bin/lmgrd` on that system. For security reasons the deamon should be started as a non-root user.

To verify if the installation is complete and the license works, pick one of the cobol demos from `$COBDIR/demo`, e.g. `tictac.cbl`, and try compiling it with `cob –x <file name>`. The compilation must finish without error messages.

Server Express / Application Server Licensing

Server Express and Application Server use a licensing system called apptrack. It must not be installed separately, but a license must be added. apptrack works different than flexlm. It is not network based, so each System where cobol programs should be run or developed has its own licensing.

Apptrack licensing distinguishes between developer and runtime licenses, and both are handled in different ways.

Contrary to flexlm, to get license keys, no system specific data must be sent to the licensing center. If a customer has licenses on his contract, he can simply send an email with the contract number to europe_codeword@hp.com and he will get the license keys in reply. Each key consists of a serial number and a license number.

To add developer license keys run the program `$COBDIR/mflmf/mflmadm`. It provides a terminal based user interface, which is easy to handle. Its main screen looks like this:

```
+----------------------------------------------------------------------------+
|                  Micro Focus License Administration Services               |
|                               Main Screen                                  |
|                                                                            |
|   Product Details:      Name                                               |
|                         Version                                            |
|                                                                            |
|   License Details:      Seat/Process                                       |
|                         Units                                              |
|                         Duration                                           |
|                         Type                                               |
|                                                                            |
|   Issuing Details:      Issue Point         00000                          |
|                         Issuer                                             |
|                         Foundry                                            |
|                         Key Version         00                             |
|                                                                            |
|    License Key:    Serial  No                                              |
|                    License No                                              |
|                                                                            |
|                                                                            |
|----------------------------------------------------------------------------|
|F1=help F2=key F3=install F4=uninstall F5=browse F6=more F7=refresh   Escape |
+----------------------------------------------------------------------------+
```

All available options and how to activate them are listed on the bottom of the screen. Enter both numbers after pressing `F2`, after returning to the main screen, press `F3` to install the license. To check which licenses are available, press `F6` twice and it will show an overview. To leave a screen of `mflmadm`, or to leave the program, always press `Esc`.

To make the licenses available to the Server Express development tools, the license server daemon must be started by entering `./mflmman`. The licenses are stored in a database, and if a license is added while the license manager daemon is running, it must be either restarted or notified that the license database has changed. The latter can be done by pressing `F7` in `mflmadm`.

To add Application Server licenses, start the program `$COBDIR/aslmf/apptrack`. It prints an options list:

```
Application Server License Administration System - AppTrack
-----------------------------------------------------------
1.   License List
2.   License Summary
3.   License Install
4.   License Uninstall
5.   Change/Set Password
6.   Reinitialize Licensing
7.   Current Users
9.   Quit
Enter the Menu Selection
```

Select `3` to install a license, then follow the instructions to enter the keys. To check which licenses are installed, select `1` (detailed) or `2` (summary).

Note that every Server Express license installed with `mflmadm` includes 5 Application Server licenses which are shown by `apptrack` in the license summary:

```
2002/12/02 14:43:46  Summary of AS Licenses Installed, In System  and
In Use:
--------------------------------------------------------------------------
   Installed   Sys   In Use
   ---------   ---   ------
**Developer Usage
        5        0       0             Server Express 64-bit Edition
```

Application Server licensing does not use a license manager daemon. Instead, the cobol runtime itself looks up the license database.

It is always a good idea to keep a copy of the license databases. To do so, backup the files `$COBDIR/mflmf/mflmdb*` and `/var/mfaslmf/mfasdb*`.

There are three types of Application Server licenses, which differ in their availability and price. The **standard license** is given to applications when they start and will be released for re-use 30 seconds after the application terminated. The **transaction license** can be re-used immediately. The **server license** is an unlimited transaction license.

More details on Server Express and Application Server Licensing can be found on the Microfocus Websites:

- License Management Facility Administrators Guide:
  http://supportline.microfocus.com/documentation/books/sx2011sp1/slpubb.htm

- Managing Application Server Licenses with AppTrack:
  http://supportline.microfocus.com/documentation/books/sx2011sp1/adpubb.htm

Java3D Installation

Java3D is a free add-on package to the java versions 1.2 and 1.3. It can be downloaded from the java download site. Unfortunately its install scripts don't check any dependencies, although the product cannot be used unless an appropriate java version and OpenGL are installed. OpenGL is part of the graphics technical computing environment.

Make sure that J3D is installed separately after its prerequisites have been installed. If it is installed within the same session as java, it can happen that J3D is installed first, and the following java installation will overwrite resp. remove parts of J3D, and J3D might not work then.

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 on information we get from there can make it difficult to find what we're looking for. The only thing that helps searching is experience.

Compiler and linker distinguish between warnings and errors. While warnings are issued to tell the developer that there might be a problem, the build process will not be aborted. When an error occurs, the build process cannot succeed, 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 bracket in a C program can cause lots of error messages, and they are all gone when the bracket is added. 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, and explain the meanings and what corrections are required. In fact, we will mainly discuss linker messages, because compiler messages are usually caused by programming errors, and their discussion would lead deep into the details of the programming languages.

`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.

```
Make: Don't know how to make x. Stop.
```

In the makefile "x" was specified as a dependent, but neither does the file exist nor was there a rule to build it. If you are the writer of the makefile, check if you really need this file to build your targets. If not, remove the file from your dependents, if yes, get it from where ever or find out how to create it and add a rule to the makefile. If you are not the writer of the makefile, ask the writer the above question.

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 linker warning. It tells us that it 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.

Unsatisfied Symbols

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

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 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 libaries 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 in that case so the object wouldn't be created, but e.g. the bundled `cc` doesn't care for undefined symbols and the linker would issue the above messages instead.
- **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, the linker does 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. However, at link time `-AA` was not present, which makes aCC 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 UX 11.X. The problem is caused by objects that have been compiled on UX 10.20. It is not supported to link both 10.X and 11.X objects. To resolve this you must recompile everything 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.

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 here is to provide more memory by increasing `maxdsiz` and/or adding swap space.

Internal errors of compilers or the linker will also lead to aborts. Most of the time 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 corefile.

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 defect 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 to find workarounds and submit a new defect report. Try to get a small test case that reproduces the problem, to provide it to the labs.

In case of a 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. # csh
```

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 and they are OK, 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 see the dilemma. We have a PA2.0 executable, but an old PA1.1 system.

Error Messages From The Dynamic Loader

When problems are reported by `dld.sl/.so`, the program could be started, but there are problems loading libraries or resolving symbols. The dynamic loader will always abort the program in a way that it writes a corefile.

Normally there is no need to analyze the corefile because `dld.sl` also writes a message to `stdout` that explains what happened. But the corefile is not totally useless because the message does not print the name of the program that has the problem. Imagine you run a shell script that starts several programs. If you get error messages from the dynamic loader, you cannot tell which program doesn't work. To find out, simply enter

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

and 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, namely the path where it was found at link time.

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

```
$ chatr 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 searched `libmylib` in `/somepath`, `/usr/lib` (embedded path) and then in the default location `/mypath`, but it wasn't in any 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 gives us a hint here. 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`, check which product uses the path `/opt/somepath`, maybe this product is not installed. Any other path could be some kind of path which was only available on the development system. At runtime the library is either expected in one of the embedded paths or in a path specified in the environment variables `SHLIB_PATH` or `LD_LIBRARY_PATH`. 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 `+vshlibunsats` 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 missing symbol with `nm(1)`, and put the path to the library where you found the symbol 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. This situation is similar to the things described at the start of this page. To resolve this, check the file type of the library listed in the error message.

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
```

The situation in which this error can occur is, that an already running program executes a call to `shl_load()` which allows to load a shared library, initiated by the program code, instead of loading it immediately at program startup. If this library, or any other library that is loaded too because it was found in the shared library list, contains thread local storage (TLS), it cannot be loaded with `shl_load()`.

To resolve this problem, all libraries that contain TLS and might be loaded while the program is running, must be linked into the program to make sure they are loaded at startup. If `shl_load()` is called then to load these libs, they are not loaded again, because they are already there. Normally the only affected libraries are `libcl` and `libpthread`.

This situation can often be observed with `perl` when using modules. These modules are in fact shared libraries. And various modules, e.g. database modules, are linked against `libcl` and `libpthread`, while some `perl` versions are not. Both libraries contain TLS, so if `perl` tries to load these modules with `shl_load()` this will fail with the above error.

In the case of `perl`, you can either relink it with `-lcl -lpthread` (you normally can because its sources are free), or use the supported `perl` version which is already linked against these libraries. If neither is possible, use `LD_PRELOAD` to load these libraries when the program is started:

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

When `perl` tries to `shl_load()` one of these libraries, the loader functions will realize that the libraries are already loaded, and not load them again.

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. If not, the only one that can really help here is the program vendor.
- 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 corefile, which is an image of the DATA part of the process address space.

This section will discuss a few special problems and how to recognize them.

If the program abort left a corefile, it can be analyzed with a debugger. If the problem is reproducable, it often is of interest to see what happens before the abort. In that case tracing the programs system calls with `tusc` will be helpful.

If there is no corefile, 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 too many ways for a program to terminate. If it does not abort with a signal, it most probably calls `exit()`. There is a good chance to catch the abort by setting a breakpoint at `exit()`. At this point you can do the same analysis as with an aborted program that left a corefile.

No Or Unusable Corefile

There are a few reasons that can interfere the writing of a corefile:

- No write permissions in the current working directory
- The program has caught the signal and exits without leaving a corefile
- The limit for the maximum size of corefiles, `ulimit –c` (see `sh-posix(1)`), is too small.
- There is not enough disk space available to write the corefile.

These reasons can either leave an empty corefile, a truncated corefile, or no corefile at all. If a corefile is truncated can be checked with `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 from the corefile does not end with the what string of `dld.sl/.so`, the corefile is incomplete and can't be used for debugging.

If a program aborted and left no corefile, although you know it should have been written, check the above issues. The size of the corefile will always be somewhat larger than the amount of data held by the process, so make sure the limits are set high enough.

Corefile Analysis

Corefile analysis is a very complex and complicated work, and should be done by the program developers, or higher level language support. It will not be discussed in depth here. But it at least starts always with the same action: Looking for the location of the abort, by getting a stacktrace of all threads:

```
$ gdb a.out core
HP gdb 5.0 for PA-RISC 1.1 or 2.0 (narrow), HP-UX 11.00
   :
(gdb) thread apply all 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 stacktrace shows in which part of the executable the abort occured, e.g. in the executable itself or in a shared library. If the application has a lot of threads, the above output can become very large.

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 are passed to the library call, and if they were valid. Most of the time this is really hard because digging in system libraries means handling assembly code. In case of doubt we must assume this is a bug in the runtime library and let WTEC have a look.

If it turns out, or at least if it seems to be a problem with a runtime library, it is always helpful to get the program vendor to provide the sources for a little test case that shows the problem. We, the expert center and the labs can analyze the problem then much easier.

Memory Overflow

As described in the process memory section, there are various limitations for a process regarding the amount of allocatable memory. If a process hits such a limit, this usually doesn't happen suddenly. Most of the time a process grows more or less slowly until it hits the limit. This growth can be observed with `top(1)` or `glance(1)`, `kmeminfo` can give a detailed view at the address space of the process, and with tusc(1) we have a very precise means to observe a process growing:

```
$ tusc -s brk,mmap <PID>
```

All processes use brk() to increase the heap and mmap() to request blocks of memory independent from the heap. The only argument of brk(), the so called brk value, enables us to calculate the data segment size of the process. Bearing in mind that normally the data segment of a process starts at the beginning of Q2, the data segment size is brk value minus 0x40000000, the start address of Q2. The above program for example has allocated 0x22000 bytes (139264 in decimal) successfully.

If a brk() or mmap() system call was not successful, tusc reports ENOMEM instead of "0" as return code. The reason for this is that the process is not allowed to allocate more memory, because it hits either a physical (no more memory/swap available) or a virtual (maxdsiz, ulimit) limit.

If you watch a growing process abort, check its size right before the abort against the system imposed limits.

If the process aborted and left a corefile, the $m command of adb(1) can print a similar map of the process address space as kmeminfo:

```
$ echo ‘$m’ | adb <program> <core>
```

Often it is said that a program crashed because it could not allocate memory. This is not true. If it crashed, then because it didn't realize that brk() (or malloc() from the programmer’s view) returned an error code. It then tried to access a memory region that was not allocated (often a null pointer) and received a signal (SIGBUS, SIGSEGV) instead. A well designed program would check the return code and handle the situation properly, e.g. by exiting with an error message.

Programs that abort due to a signal usually leave a corefile. The problem with these corefiles is that the abort happens when accessing an invalid address. When debugging this, you might be able to find the invalid address, probably 0x00000000, but you probably cannot tell where it came from. The brk()/malloc() that failed could have been a million instructions in the past.

If a program hits a system limit, it might really need more memory to execute properly. In that case you have no other chance than to increase the limits and to provide more resources. How to do that is discussed in the section Out Of Memory later in this chapter.

Memory Leaks

If a process grows without bounds, it is likely to have a memory leak, which means it allocates blocks of memory, but doesn't free the blocks when they’re no longer needed. In this case 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 always coding errors and must be corrected by the programmers.

There are several non-HP tools available on the market that can help finding memory leaks. The current versions of WDB also have heap checking features. The simplest way find leaks with gdb, is to 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 11.00
    :
(gdb) set heap-check on
(gdb) set heap-check frame-count 15
(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 discovered, 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/gdb.html#SEC128

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 also cause the process to abort with a signal and leave a corefile, but debugging such a corefile 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 corefile.

At runtime you can't 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. One way to find out the actual process stack size (in 4k pages) is with `kmeminfo`:

```
$ kmeminfo –pid <PID> | grep –i –e stack –e vaddr
    type      space              vaddr  ref   virt   phys   swap
   STACK 0xb1f8400.0x000000007eff0000    1      8      8      9
```

```
$
```

The above output shows 8 pages (32 kBytes) for the stack. `kmeminfo` reads the kernel memory so you need to be root to have permissions to do that.

Another way to find out the current stack size of a running process is to print out the value of its stack pointer using a debugger:

```
$ echo ‘p/x $sp’ | gdb <program> <PID>
:
$1 = 0x7f7f0b10
Detaching from program: /usr/bin/ksh, process 7753
$
```

`p/x $sp` will print the current value of the stack pointer. While the debugger is attached to the running process, it will be stopped, and continue to run when the debugger detaches.

Normal executables have their stack located at the end of Q2 (32-bit PA-RISC). So we can calculate the unused stack space by 0x80000000 minus `$sp`. In the above case there still are 0x80f4f0 Bytes (approx. 8MB) free. Compare that value with the setting of `maxssiz` to find out how much is already in use. If the program has most part of the stack already in use, it is likely to take the rest soon and abort.

If a program has already aborted, we can also use the debugger to get the value of the stack pointer from the corefile and see if it was due to a stack overflow. But remember that the whole stack could be corrupted and so could be the stack pointer because when leaving a function it will throw away one stack frame and restore its value to the previous frame, and this address is taken from the stack. So don't always trust the value you get. A sure way to find out the stack size from a corefile is with `adb`:

```
$ echo '$m' | adb <program> core | grep -i stack
Stack:          b = 7EFF0000    e = 7EFF8000    f = 5365C
$
```

The actual stack size is the difference between the “`e =`” and “`b =`” values, 0x8000 (32kBytes) in the above case.

In a multithreaded program stack overflows are much more likely because thread stacks usually are much smaller than the process stack. Finding out if there was a thread stack overflow can only be achieved with a debugger. First select the thread that received the signal, then go to the frame below the entry `<signal handler called>`, if any (if not, use frame #0), 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
#0  0xc020c718 in kill+0x10 () from lib/libc.2
   :
#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 with adb how this address fits into the mmap'ed regions:

```
$ 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 well within the borders of the marked MMF region.

If you see very strange things in a corefile, and you verified that you use the same library versions for debugging as at program runtime, it is very probable that you have a stack corruption here.

If you are quite sure that the abort was due to a stack overflow, try increasing `maxssiz` up to a convenient value, which depends on the program. If this makes the program take longer until it aborts or keeps the program from aborting, it looks like you were right.

If a value larger than 100 MB is still not sufficient, the programmer should seriously consider changing his code. Most of the time heavy stack usage is caused by large local variables in functions. A probable change could be using `malloc()` to allocate memory on the heap instead. With fortran programs, try using the `+save` compiler option.

It is normally not possible to change the thread stack size from outside a program. This is done by a pthreads library function.

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 corefile 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.sl`, 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 leave a corefile. 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.

Debugging the corefile of a JVM requires very deep knowledge of the internal structure of java. Except of maybe getting a stacktrace just like for any other corefile, java debugging is a thing that's preserved to the languages WTEC.

Besides a JVM internal error, it is also possible that the JVM aborts because of a problem with the application running inside the JVM. 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 versions since 1.3.1.02 come with the library `libjunwind.sl`, e.g. for java 1.4:

```
/opt/java1.4/jre/lib/PA_RISC/server/libjunwind.sl      (PA-RISC 1.1)
/opt/java1.4/jre/lib/PA_RISC2.0/server/libjunwind.sl   (PA-RISC 2.0)
/opt/java1.4/jre/lib/PA_RISC2.0W/server/libjunwind.sl  (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/libj
unwind.sl
$ gdb /opt/java1.4/bin/PA_RISC2.0/native_threads/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+0x5b4 () 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
.3/jre/lib/PA_RISC2.0/server/libjvm.sl
#8  0xc47a9698 in handle_exception_C__11OptoRuntimeSFP10JavaThread+0x14
8 () from /opt/java1.3/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.reflect
ion.classes.custom.CustomClassLoader::findClassInternal (java.lang.Stri
ng) ->java.lang.Class ()
#12 0x7935e2b0 in compiled frame: com.borland.enterprise.module.reflect
ion.classes.custom.CustomClassLoader::findClass (java.lang.String) ->ja
va.lang.Class ()
#13 0x7959f460 in compiled frame: com.borland.enterprise.module.reflect
ion.classes.custom.CustomClassLoader::loadClass (java.lang.String, bool
ean) ->java.lang.Class ()
   :
```

In the above example the frames `#0-#9` are from native functions, 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 cought.

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

```
$ gdb /opt/java1.2/bin/PA_RISC2.0/native_threads/java core
    :
(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+0x5b4 () 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
.3/jre/lib/PA_RISC2.0/server/libjvm.sl
#8  0xc47a9698 in handle_exception_C__11OptoRuntimeSFP10JavaThread+0x14
8 () from /opt/java1.3/jre/lib/PA_RISC2.0/server/libjvm.sl
#9  0x78f18108 in ?? ()
```

Another thing we can do if java aborts is the same as at compiler/linker aborts. Check if there was a memory resource problem. Although java usually reports a java.lang.OutOfMemoryError, sometimes it does not.

Other things are to check for newer java versions on the web, or check if similar stack traces can be found in CHART (HP internal) or KMine (HP internal). If the problem occurs with the latest java version, forward the problem to the languages WTEC.

Out Of Memory

The section Process Memory described how processes use memory, which limits exist and how to modify the usage of the address space. This section shows step by step how to proceed if a program doesn't get enough memory.

To determine which limit might have been hit we need to know how much memory the process was able to allocate and which type of memory it used. There are three system calls that provide different types of memory to a process:

- `brk()`: process heap
- `mmap()`: memory mapped regions
- `shmget()`: 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 search for the system call that failed with an error code, or look at the current usage of the process address space with `kmeminfo`, or, if the process has left a corefile, we can use the `$m` command of adb(1) which gives similar output as `kmeminfo`:

```
$ tusc –s brk,mmap,shmget <PID>

$ kmeminfo –pid <PID>

$ echo '$m ' | adb <program> <corefile>
```

The following table shows the existing limits for memory allocation:

- `ulimit`
- kernel parameters
- quadrant usage
- virtual memory

It is assumed that we have enough virtual memory available on the system.

If a process cannot allocate enough memory with **`brk(2)`**, we need to look at `ulimit` (`ulimit –d`, see `sh-posix(1)`) and the kernel parameters `maxdsiz/maxssiz` (`kmtune(1M)`). Increase `maxdsiz/maxdsiz_64bit` to match the process requirements. The initial value for `ulimit` is the minimum of these two kernel parameters so it will follow changes to them automatically.

Remember that the stack shares the data quadrant with the heap, so a large `maxssiz` decreases the available space for the heap. Normally no large value for `maxssiz` is required.

Note that changes to kernel parameters require a kernel rebuild followed by a reboot in order to take effect.

There are no kernel parameters that restrict the usage of **`mmap(2)`**. Private mappings will be allocated in the private data quadrants. The only limit is the number of private quadrants available to the executable, reduced by the size of process text (`EXEC_MAGIC` executables only), heap and stack which also reside in the private quadrants. Shared mappings will end up in the shared quadrants together with shared memory segments and shared library text segments.

If a process uses System V shared memory with **shmget(2)**, the shared memory segments will reside in the shared quadrants only and the same restrictions apply as for shared mappings. There are a few kernel parameters which limit the usage of shared memory. Not the total amount of shared memory is limited, but the maximum segment size (shmmax) and the number of segments per process (shmseg) and system wide (shmmni).

If a 32-bit process needs more than 1 GB private data (heap plus stack plus mapped regions), we can change the usage of the quadrants. This is described in detail in the Process Memory section. If 2 GB shared memory are not enough, we can change the usage of Q2 as shared data. But however we twist and turn, the total amount of memory available for a 32-bit process is limited by its address space to 4 GB, even if the system has plenty of virtual memory.

Executables which want to share memory with a SHMEM_MAGIC program (Q2 shared), should be SHMEM_MAGIC too. Otherwise they cannot access shared segments which reside in Q2.

Java Out Of Memory

Normally, the JVM reports a java.lang.OutOfMemoryError, if it wasn't able to fulfill a request to allocate a new object. So usually it is not difficult to tell if a JVM ran out of memory. But we have to distinguish between an out of memory condition for the JVM itself and for the java application running inside the JVM.

There are a few methods to check if the OutOfMemoryError occurred in the java heap (for the java application). If the JVM aborts, it usually leaves a file called hs_err_pidXXXXX.log, where XXXXX is the process ID of the aborted JVM. Since version 1.4.2 this 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)
```

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

For older java versions that do not include this information, the garbage collection (GC) log can be checked. To obtain one, the JVM must be started with the-Xverbosegc:<logfile> option. The logfile will contain one line for 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 gen
     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.

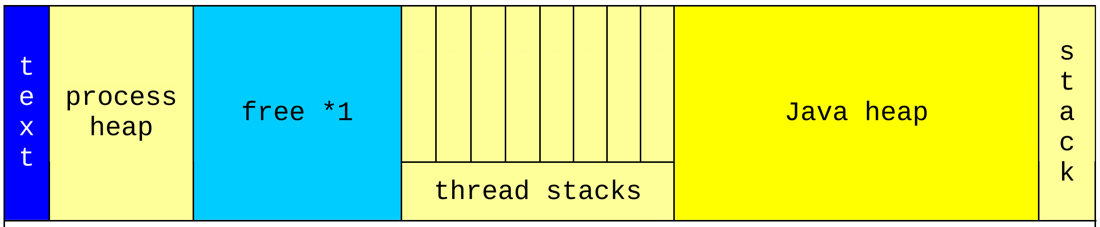
Details about the GC log format can be found in the output of `java -Xverbosegc:help`.

Increasing the heap with the **`-Xmx`** java option can be the solution. But the java heap cannot be increased infinitely. The total amount of private data must fit into the private quadrants.

The java heaps, the thread stacks, and various other things used by the JVM are allocated using `mmap()`. There is no kernel limit for mapped regions, but there is a kernel limit for the number of threads, `max_thread_proc` (per process) and `nkthread` (system wide). If the JVM cannot create a new thread because it hit one of these limits, it will also report an `OutOfMemoryError`. The number of threads can be found out with `gdb`:

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

As for any other process, the JVM might 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:



*1 free for increasing the process heap or more thread stacks

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, use `adb` on the corefile:

```
$ 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.

To be able to provide enough private address space for different `-Xmx` values, the HP 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

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). But there might 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.

To solve JVM out of memory problems, one has to balance the memory requirements of the java application and the JVM, and use 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 can become crucial. E.g. if there are 1000 threads (which can be perfectly normal 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.

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.

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.

Of course, if it is really hard to supply enough memory to JVM and java application, the 64-bit JVM can be used with the **`-d64`** option, which is available since java 1.4.

Environment Variables

This page gives an overview to some helpful environment variables that allow modifying the way programs are built or executed.

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
`PCOPTS`	Pascal

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, `LDOPTS`. 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 `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 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 `chatr(1)`.

`LD_LIBRARY_PATH` has the same purpose, effect and usage as `SHLIB_PATH`, but it only works for ELF (64-bit) programs. 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>
```

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

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 page lists the most important links that lead to software development related pages.

General Links

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

- http://docs.hp.com/hpux/dev/
  docs.hp.com 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.

- http://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.

- http://devresource.hp.com/
  This is another entry point when searching development related information. Its focus lies on integration of applications with HP software.

- http://software.hp.com
  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.

- http://wtec.cup.hp.com/~lang/ (HP internal)
  This is the homepage of the languages WTEC team. It provides information to all supported, software development related products. Of course this is an HP internal site which is not accessible to customers.

- http://hpux.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.

Special Links

- **GNU Compiler Collection**
  www.gnu.org/directory/gcc (non HP)

- **Gnu Compiler Collection for HP-UX**
  Contains C, C++ and F77 compilers.
  http://hpux.asknet.de/hppd/hpux/Gnu/gcc-3.2/ (non HP)

- **Gnu Compilers provided by HP**
  C/C++ compilers only.
  http://www.hp.com/go/gcc