Environment

- When a program is executed...
  - process that does exec can pass command-line arguments to the new program
  - this is part of the UNIX system shells

    ```c
    int main( int argc, char *argv[] )
    {
        int i;

        /* echo all command-line args */
        for ( i = 0 ; i < argc ; i++ )
            printf( "argv[%d]: %s\n", i, argv[i] );
    }
    ```

Environment

- Environment List
  - each program also passed an environment list
  - this list is an array of char pointers, with each pointer containing the address of a null-terminated C string
  - the address of the array of pointers is contained in the global variable `environ`:
    ```c
    extern char **environ;
    ```
Environment

- Example of environment with five strings
  - the null bytes at the end of each string are explicitly shown

![Diagram of environment structure]

Environment

- Terms
  - `environ` is called the `environment pointer`
  - the array of pointers is called the `environment list`
  - the strings they point to are called `environment strings`
Environment

Historically, most UNIX systems have provided a third argument to the main function that is the address to the environment list

```
int main( int argc, char *argv[], char *envp[] );
```

Because ISO C specifies that the main function be written with two arguments, and because this third argument provides no benefit over the global variable `environ`, POSIX.1 specifies that `environ` should be used instead of the (possible) third argument. Access to specific environment variables is normally through the `getenv` and `putenv` functions instead of through the `environ` variable. But to go through the entire environment, the `environ` pointer must be used.

---

Environment

Show the environment

```c
#include <stdio.h>

int main( int argc, char *argv[], char *envp[] )
{
    int i;
    /* echo all environment args */
    for (i = 0 ; envp[i] ; i++)
        printf( "environ[%d]: %s\n", i, envp[i] );
}
```
Environment

- Environment Variables
  - environment strings are usually of the form \texttt{name=value}
  - the Unix kernel never looks at these strings
  - their interpretation is up to the various applications
  - the shell uses numerous environment variables
    - some are automatically set at login, e.g., \texttt{HOME}, \texttt{USER}
    - others are for us to set, e.g., If we set the environment variable \texttt{MAILPATH}, for example, it tells the Bourne shell, GNU Bourne-again shell, and Korn shell where to look for mail.

Environment

- Support for various environment list functions

<table>
<thead>
<tr>
<th>Function</th>
<th>ISO C</th>
<th>POSIX.1</th>
<th>FreeBSD 5.2.1</th>
<th>Linux 2.4.22</th>
<th>Mac OS X 10.3</th>
<th>Solaris 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>getenv</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>putenv</td>
<td></td>
<td>XSI</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>setenv</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>unsetenv</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>clearenv</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
Environment

ISO C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

```
#include <stdlib.h>
char *getenv(const char *name);
```

Returns: pointer to value associated with name, NULL if not found

Note that this function returns a pointer to the value of a name=value string. We should always use `getenv` to fetch a specific value from the environment, instead of accessing `environ` directly.

Environment

Manipulating environment variables

```
#include <stdlib.h>

int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error

The `putenv` function takes a string of the form `name=value` and places it in the environment list. If `name` already exists, its old definition is first removed.
Environment

Manipulating environment variables

```c
#include <stdlib.h>

int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error

The `setenv` function sets `name` to `value`. If `name` already exists in the environment, then (a) if `rewrite` is nonzero, the existing definition for `name` is first removed; (b) if `rewrite` is 0, an existing definition for `name` is not removed, `name` is not set to the new value, and no error occurs.

Environment

Manipulating environment variables

```c
#include <stdlib.h>

int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error

The `unsetenv` function removes any definition of `name`. It is not an error if such a definition does not exist.
Environment

 Manipulating environment variables

```c
#include <stdlib.h>

int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

All return: 0 if OK, nonzero on error

Note the difference between `putenv` and `setenv`. Whereas `setenv` must allocate memory to create the `name=value` string from its arguments, `putenv` is free to place the string passed to it directly into the environment. On Linux and Solaris, the `putenv` implementation places the address of the string we pass to it directly into the environment list. In this case, it would be an error to pass it a string allocated on the stack, since the memory would be reused after we return from the current function.