

Grundlagen der Informatik

Vorlesungsskript

Ausgewählte Funktionen der libc Standardbibliothek

Prof. Dr.-Ing. B. Lang
FH Osnabrück

Die Beschreibungen wurden der Dokumentation

„*The GNU C Library*“ entnommen.

(Siehe z.B. http://www.delorie.com/gnu/docs/glibc/libc_toc.html)

Argument Access Macros

Here are descriptions of the macros used to retrieve variable arguments. These macros are defined in the header file ``stdarg.h'`.

Data Type: `va_list`

The type `va_list` is used for argument pointer variables.

Macro: `void va_start (va_list ap, last-required)`

This macro initializes the argument pointer variable `ap` to point to the first of the optional arguments of the current function; `last-required` must be the last required argument to the function. See section Old-Style Variadic Functions, for an alternate definition of `va_start` found in the header file ``varargs.h'`.

Macro: `type va_arg (va_list ap, type)`

The `va_arg` macro returns the value of the next optional argument, and modifies the value of `ap` to point to the subsequent argument. Thus, successive uses of `va_arg` return successive optional arguments.

The type of the value returned by `va_arg` is `type` as specified in the call. `type` must be a self-promoting type (not `char` or `short int` or `float`) that matches the type of the actual argument.

Macro: `void va_end (va_list ap)`

This ends the use of `ap`. After a `va_end` call, further `va_arg` calls with the same `ap` may not work. You should invoke `va_end` before returning from the function in which `va_start` was invoked with the same `ap` argument.

In the GNU C library, `va_end` does nothing, and you need not ever use it except for reasons of portability.

Variable Argumentenliste (Beispiel 1)

```
#include <stdio.h>
#include <stdarg.h>

int find_the_sum(int count, ...) {
    va_list ap;
    int i;
    int total = 0;

    va_start(ap, count);
    for (i = 0; i < count; i++) {
        total += va_arg(ap, int);
    }
    va_end(ap);
    return total;
}

int main() {
    int sum;
    sum = find_the_sum(6, 1, 2, 3, 4,
                      5, 6);
    printf("Summe1:%i\n", sum);
    sum = find_the_sum(8, 1, 2, 3, 4,
                      5, 6, 7, 8);
    printf("Summe2:%i\n", sum);
    return 0;
}
```

Variable Argumentenliste (Beispiel 2)

```
#include <stdio.h>
#include <stdarg.h>

int debugmessage(int line, char *filename,
                 char *fmt, ...) {
    va_list ap;
    char buffer[1000];
    va_start(ap, fmt);
    vsprintf(buffer, fmt, ap);
    va_end(ap);
    return printf("Datei %s, Zeile %i: %s",
                 filename, line, buffer);
}

#define __POS__      __LINE__, __FILE__
#define DEBUGMESSAGE(a...) \
    debugmessage(__LINE__, __FILE__, a);
#define printf(a...) \
    debugmessage(__LINE__, __FILE__, a);

int main() {
    debugmessage(__POS__, "%i %i %i\n", 1, 2, 3);
    debugmessage(__POS__, "%i %i %i\n", 1, 2, 3);
    DEBUGMESSAGE("%i %i %i\n", 1, 2, 3);
    DEBUGMESSAGE("%i %i %i\n", 1, 2, 3);
    printf("\n");
    return 0;
}
```

```
Datei varargs2.c, Zeile 19: 1 2 3
Datei varargs2.c, Zeile 20: 1 2 3
Datei varargs2.c, Zeile 21: 1 2 3
Datei varargs2.c, Zeile 22: 1 2 3
Datei varargs2.c, Zeile 23:
```

Copying and Concatenation

```
void *memcpy(void *to, const void *from,  
             size_t size)
```

The **memcpy** function copies **size** bytes from the object beginning at **from** into the object beginning at **to**. The behavior of this function is undefined if the two arrays **to** and **from** overlap; use **memmove** instead if overlapping is possible.

The value returned by **memcpy** is the value of **to**.

Here is an example of how you might use **memcpy** to copy the contents of an array:

```
    struct foo *oldarray, *newarray;  
    int arraysize;  
    ...  
    memcpy (new, old,  
           arraysize * sizeof (struct foo));
```

```
void *memmove(void *to, const void *from,  
             size_t size)
```

memmove copies the **size** bytes at **from** into the **size** bytes at **to**, even if those two blocks of space overlap. In the case of overlap, **memmove** is careful to copy the original values of the bytes in the block at **from**, including those bytes which also belong to the block at **to**.

```
void *memset(void *block, int c,  
            size_t size)
```

This function copies the value of **c** (converted to an unsigned char) into each of the first **size** bytes of the object beginning at **block**. It returns the value of **block**.

Speicherfunktionen: *memcpy*, *memmove*, *memset*

```
#include <stdio.h>
#include <string.h>
int main () {
    char string1[100] = "Hello world";
    char string2[100];
    struct {
        int a;
        char b;
        float c;
    } struct1, struct2;
    memcpy(string2,string1,100);
    printf("%s, %s\n",string1,string2);
    struct1.a = 1234567;
    struct1.b = 'b';
    struct1.c = 1.234567;
    memcpy(&struct2,&struct1,sizeof(struct1));
    printf("(%i '%c' %f), (%i '%c' %f)\n",
           struct1.a,struct1.b,struct1.c,
           struct2.a,struct2.b,struct2.c);
    memmove(string1,string1+6,94);
    printf("%s\n",string1);
    memset(&struct1,0,sizeof(struct1));
    printf("(%i '%c' %f)\n",
           struct1.a,struct1.b,struct1.c);
    return 0;
}
```

Ein langer Sprungbefehl: *setjmp*, *longjmp*

```
#include <stdio.h>
#include <setjmp.h>

void one(jmp_buf env) {
    printf("one\n");
    longjmp(env,1);
}

void two(jmp_buf env) {
    printf("two\n");
    longjmp(env,2);
}

int main() {
    jmp_buf env;

    switch (setjmp(env)) {
        case 0:
            one(env);
            break;
        case 1:
            two(env);
            break;
        default:
            break;
    }
    return 0;
}
```

Details of Non-Local Exits

Here are the details on the functions and data structures used for performing non-local exits. These facilities are declared in `'setjmp.h'`.

Data Type: `jmp_buf`

Objects of type `jmp_buf` hold the state information to be restored by a non-local exit. The contents of a `jmp_buf` identify a specific place to return to.

Macro: `int setjmp (jmp_buf state)`

When called normally, `setjmp` stores information about the execution state of the program in `state` and returns zero. If `longjmp` is later used to perform a non-local exit to this state, `setjmp` returns a nonzero value.

Function: `void longjmp (jmp_buf state, int value)`

This function restores current execution to the state saved in `state`, and continues execution from the call to `setjmp` that established that return point. Returning from `setjmp` by means of `longjmp` returns the value argument that was passed to `longjmp`, rather than 0. (But value is given as 0, `setjmp` returns 1).

There are a lot of obscure but important restrictions on the use of `setjmp` and `longjmp`. Most of these restrictions are present because non-local exits require a fair amount of magic on the part of the C compiler and can interact with other parts of the language in strange ways.

Details of Non-Local Exits

The **setjmp** function is actually a macro without an actual function definition, so you shouldn't try to `#undef` it or take its address. In addition, calls to **setjmp** are safe in only the following contexts:

- As the test expression of a selection or iteration statement (such as `if`, `switch`, or `while`).
- As one operand of a equality or comparison operator that appears as the test expression of a selection or iteration statement. The other operand must be an integer constant expression.
- As the operand of a unary `!` operator, that appears as the test expression of a selection or iteration statement.
- By itself as an expression statement.

Return points are valid only during the dynamic extent of the function that called **setjmp** to establish them. If you **longjmp** to a return point that was established in a function that has already returned, unpredictable and disastrous things are likely to happen.

You should use a nonzero value argument to **longjmp**. While **longjmp** refuses to pass back a zero argument as the return value from **setjmp**, this is intended as a safety net against accidental misuse and is not really good programming style.

When you perform a non-local exit, accessible objects generally retain whatever values they had at the time **longjmp** was called. The exception is that the values of automatic variables local to the function containing the **setjmp** call that have been changed since the call to **setjmp** are indeterminate, unless you have declared them **volatile**.

Ein langer Sprungbefehl: *setjmp*, *longjmp*

```
#include <stdio.h>
#include <setjmp.h>

void one(jmp_buf env) {
    printf("one\n");
    longjmp(env,1);
}

void two(jmp_buf env) {
    printf("two\n");
    longjmp(env,2);
}

int main() {
    jmp_buf env;

    switch (setjmp(env)) {
        case 0:
            one(env);
            break;
        case 1:
            two(env);
            break;
        default:
            break;
    }
    return 0;
}
```

```
> ./longjmp.exe
one
two
```

Basic CPU Time Inquiry

To get the elapsed CPU time used by a process, you can use the `clock` function. This facility is declared in the header file `time.h`.

In typical usage, you call the `clock` function at the beginning and end of the interval you want to time, subtract the values, and then divide by `CLOCKS_PER_SEC` (the number of clock ticks per second), like this:

```
#include <time.h>

clock_t start, end;
double elapsed;

start = clock();
... /* Do the work. */
end = clock();
elapsed = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Different computers and operating systems vary wildly in how they keep track of processor time. It's common for the internal processor clock to have a resolution somewhere between hundredths and millionths of a second.

In the GNU system, `clock_t` is equivalent to `long int` and `CLOCKS_PER_SEC` is an integer value. But in other systems, both `clock_t` and the type of the macro `CLOCKS_PER_SEC` can be either integer or floating-point types. Casting processor time values to double, as in the example above, makes sure that operations such as arithmetic and printing work properly and consistently no matter what the underlying representation is.

Macro: `int CLOCKS_PER_SEC`

The value of this macro is the number of clock ticks per second measured by the `clock` function.

Data Type: `clock_t`

This is the type of the value returned by the `clock` function. Values of type `clock_t` are in units of clock ticks.

Function: `clock_t clock (void)`

This function returns the elapsed processor time. The base time is arbitrary but doesn't change within a single process. If the processor time is not available or cannot be represented, `clock` returns the value `(clock_t)(-1)`.

Relative Zeitmessung: clock

```
#include <stdio.h>
#include <time.h>
main(){
    int i;
    clock_t initial;
    initial = clock();
    printf("Initial clock value: %d\n",
           initial/CLOCKS_PER_SEC);
    for (i=0;i<80000000;i++) {
        i++;
        --i;
    }
    printf("%d seconds have elapsed\n",
           (clock()-initial)/CLOCKS_PER_SEC);

    return 0;
}
```

Simple Calendar Time

This section describes the `time_t` data type for representing calendar time, and the functions which operate on calendar time objects. These facilities are declared in the header file `time.h`.

Data Type: `time_t`

This is the data type used to represent calendar time. When interpreted as an absolute time value, it represents the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time. (This date is sometimes referred to as the epoch.) POSIX requires that this count ignore leap seconds, but on some hosts this count includes leap seconds if you set TZ to certain values (see section Specifying the Time Zone with TZ).

In the GNU C library, `time_t` is equivalent to `long int`. In other systems, `time_t` might be either an integer or floating-point type.

Function: `double difftime (time_t time1, time_t time0)`

The `difftime` function returns the number of seconds elapsed between time `time1` and time `time0`, as a value of type `double`. The difference ignores leap seconds unless leap second support is enabled.

In the GNU system, you can simply subtract `time_t` values. But on other systems, the `time_t` data type might use some other encoding where subtraction doesn't work directly.

Function: `time_t time (time_t *result)`

The `time` function returns the current time as a value of type `time_t`. If the argument `result` is not a null pointer, the time value is also stored in `*result`. If the calendar time is not available, the value `(time_t)(-1)` is returned.

Broken-down Time

Calendar time is represented as a number of seconds. This is convenient for calculation, but has no resemblance to the way people normally represent dates and times. By contrast, broken-down time is a binary representation separated into year, month, day, and so on. Broken down time values are not useful for calculations, but they are useful for printing human readable time.

A broken-down time value is always relative to a choice of local time zone, and it also indicates which time zone was used.

The symbols in this section are declared in the header file `time.h`.

Broken-down Time

Data Type: `struct tm`

This is the data type used to represent a broken-down time. The structure contains at least the following members, which can appear in any order:

`int tm_sec`

This is the number of seconds after the minute, normally in the range 0 through 59. (The actual upper limit is 60, to allow for leap seconds if leap second support is available.)

`int tm_min`

This is the number of minutes after the hour, in the range 0 through 59.

`int tm_hour`

This is the number of hours past midnight, in the range 0 through 23.

`int tm_mday`

This is the day of the month, in the range 1 through 31.

`int tm_mon`

This is the number of months since January, in the range 0 through 11.

`int tm_year`

This is the number of years since 1900.

`int tm_wday`

This is the number of days since Sunday, in the range 0 through 6.

`int tm_yday`

This is the number of days since January 1, in the range 0 through 365.

`int tm_isdst`

This is a flag that indicates whether Daylight Saving Time is (or was, or will be) in effect at the time described. The value is positive if Daylight Saving Time is in effect, zero if it is not, and negative if the information is not available.

`long int tm_gmtoff`

This field describes the time zone that was used to compute this broken-down time value, including any adjustment for daylight saving; it is the number of seconds that you must add to UTC to get local time. You can also think of this as the number of seconds east of UTC. For example, for U.S. Eastern Standard Time, the value is $-5*60*60$. The `tm_gmtoff` field is derived from BSD and is a GNU library extension; it is not visible in a strict ISO C environment.

`const char *tm_zone`

This field is the name for the time zone that was used to compute this broken-down time value. Like `tm_gmtoff`, this field is a GNU extension, and is not visible in a strict ISO C environment.

Broken-down Time

Function: struct tm * localtime (const time_t *time)

The **localtime** function converts the calendar time pointed to by **time** to broken-down time representation, expressed relative to the user's specified time zone.

The return value is a pointer to a static broken-down time structure, which might be overwritten by subsequent calls to **ctime**, **gmtime**, or **localtime**. (But no other library function overwrites the contents of this object.)

Calling **localtime** has one other effect: it sets the variable **tzname** with information about the current time zone. See section Functions and Variables for Time Zones.

Function: struct tm * gmtime (const time_t *time)

This function is similar to **localtime**, except that the broken-down time is expressed as Coordinated Universal Time (UTC)---that is, as Greenwich Mean Time (GMT)---rather than relative to the local time zone.

Recall that calendar times are always expressed in coordinated universal time.

Function: time_t mktime (struct tm *broketime)

The **mktime** function is used to convert a broken-down time structure to a calendar time representation. It also "normalizes" the contents of the broken-down time structure, by filling in the day of week and day of year based on the other date and time components.

The **mktime** function ignores the specified contents of the **tm_wday** and **tm_yday** members of the broken-down time structure. It uses the values of the other components to compute the calendar time; it's permissible for these components to have unnormalized values outside of their normal ranges. The last thing that **mktime** does is adjust the components of the broketime structure (including the **tm_wday** and **tm_yday**).

If the specified broken-down time cannot be represented as a calendar time, **mktime** returns a value of **(time_t)(-1)** and does not modify the contents of broketime.

Calling **mktime** also sets the variable **tzname** with information about the current time zone. See section Functions and Variables for Time Zones.

Absolute Zeitmessung: *time*, *ctime*, *localtime*

```
#include <stdio.h>
#include <time.h>
const char *Wochentage[] = {"Montag",
                             "Dienstag", "Mittwoch", "Donnerstag",
                             "Freitag", "Samstag", "Sonntag" };
main(){
    time_t zeit;
    struct tm ZeitInfo;
    int tag, minute, sekunde;

    printf("Zeit ist %i\n", time(&zeit));
    printf("Zeit ist %s\n", ctime(&zeit));
    ZeitInfo = *localtime(&zeit);
    printf("%s den %d.%d.%d um %d:%d:%d\n",
           Wochentage[ZeitInfo.tm_wday],
           ZeitInfo.tm_mday, ZeitInfo.tm_mon,
           ZeitInfo.tm_year+1900,
           ZeitInfo.tm_hour, ZeitInfo.tm_min,
           ZeitInfo.tm_sec );
    ZeitInfo = *gmtime(&zeit);
    printf("%s den %d.%d.%d um %d:%d:%d\n",
           Wochentage[ZeitInfo.tm_wday],
           ZeitInfo.tm_mday, ZeitInfo.tm_mon,
           ZeitInfo.tm_year+1900,
           ZeitInfo.tm_hour,
           ZeitInfo.tm_min, ZeitInfo.tm_sec );

    return 0;
}
```

Creating a Process

The fork function is the primitive for creating a process. It is declared in the header file `'unistd.h'`.

Function: `pid_t fork (void)`

The fork function creates a new process.

If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process.

If process creation failed, fork returns a value of -1 in the parent process. The following errno error conditions are defined for fork:

EAGAIN

There aren't enough system resources to create another process, or the user already has too many processes running.

This means exceeding the `RLIMIT_NPROC` resource limit, which can usually be increased; see section Limiting Resource Usage.

ENOMEM

The process requires more space than the system can supply.

The specific attributes of the child process that differ from the parent process are:

- The child process has its own unique process ID.
- The parent process ID of the child process is the process ID of its parent process.
- The child process gets its own copies of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. See section Control Operations on Files.
- However, the file position associated with each descriptor is shared by both processes; see section File Position.
- The elapsed processor times for the child process are set to zero; see section Processor Time.
- The child doesn't inherit file locks set by the parent process. See section Control Operations on Files.
- The child doesn't inherit alarms set by the parent process. See section Setting an Alarm.
- The set of pending signals (see section How Signals Are Delivered) for the child process is cleared. (The child process inherits its mask of blocked signals and signal actions from the parent process.)

Executing a File

This section describes the exec family of functions, for executing a file as a process image. You can use these functions to make a child process execute a new program after it has been forked.

The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file `unistd.h`.

```
int execv(const char *filename, char *const argv[])
```

The `execv` function executes the file named by `filename` as a new process image.

The `argv` argument is an array of null-terminated strings that is used to provide a value for the `argv` argument to the `main` function of the program to be executed. The last element of this array must be a null pointer. By convention, the first element of this array is the file name of the program sans directory names. See section Program Arguments, for full details on how programs can access these arguments.

The environment for the new process image is taken from the environment variable of the current process image; see section Environment Variables, for information about environments.

```
int execl(const char *filename, const char *arg0, ...)
```

This is similar to `execv`, but the `argv` strings are specified individually instead of as an array. A null pointer must be passed as the last such argument.

```
int execve(const char *fn, char *const argv[],  
            char *const env[])
```

```
int execle(const char *filename, const char *arg0,  
            char *const env[], ...)
```

```
int execvp(const char *filename, char *const argv[])
```

```
int execlp(const char *filename, const char *arg0,  
            ...)
```

Prozesse: *fork*, *execl*

```
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Execute the command using this shell program. */
#define SHELL "/bin/sh"

int
my_system (const char *command)
{
    int status;
    pid_t pid;

    pid = fork ();
    if (pid == 0)
    {
        /* This is the child process.
         * Execute the shell command. */
        execl (SHELL, SHELL, "-c", command, NULL);
        _exit (EXIT_FAILURE);
    }
    else if (pid < 0)
        status = -1; /* The fork failed. Report failure. */
    else
        /* This is the parent process.
         * Wait for the child to complete. */
        if (waitpid (pid, &status, 0) != pid) status = -1;
    return status;
}

int main() {
    my_system("ls -l");
    return 0;
}
```