Aalto University
School of Electrical
Engineering

ELEC-C7310 Sovellusohjelmointi
Lecture 2: Environment

Risto Järvinen

September 14, 2020

Lecture contents

- Prepping: system calls and error conventions.
- Environment, taking a look at the surroundings.

- Stevens: parts of ch1, ch6 and ch7.
- Kerrisk: parts of ch2, ch3, ch6-11, plus ch35.1.

Prepping: System calls

- Calls from userspace to kernelspace. man 2 syscall
- Perform what can't be done inside userspace.
- Most are defined in unistd.h (=painful to read). man 2 syscalls
- System calls are a definite strain and when optimizing for performance, it's usually worth minimizing them. For example, buffered I/O in libc.

Prepping: Error handling

- Naturally function calls can't always succeed.
- Standardized error reporting.
  - return value
  - errno global variable
- man 3 errno
- Helper functions:
  - void perror(const char *s), prints *s as a prefix and appends error message that corresponds to current errno.
  - char *strerror(int errnum), returns string describing the error.

Error handling: some common errors 1/2

E2BIG	The argument list is too long.
EACCES	Access would be denied
EAGAIN	No data is available, try again later. Used with non-blocking I/O
EINTR	System call was interrupted.
EINVAL	Invalid value in argument.
EPERM	Not enough permissions.

Error handling: some common errors 2/2

EIO	I/O error, usually hardware or unrecoverable error.
EISDIR	Argument is a directory, when function was expecting a file.
ENOENT	No such file or directory.
ENOMEM	Out of memory.
EEXIST	File already exists. When trying to create a file but it already exists.
ENOSPC	Out of space.
EBADF	Bad file number.

Environment 1/2

Each running program has an environment:

- Program state, context.
- Memory layout.
- Process priority.
- Command-line arguments.
- Environmental variables.
- Process ID.
- Credentials.
- Resource limits
- File system root. (next lecture)
- Current working directory. (next lecture)
- Terminal. (next lecture)

Environment 2/2

System has some global environment.

- System capabilities
- System information
- System time and date

Context

- Processor state as it is executing the program.
- When multitasking, the state is stored and restored.
- Each change is called a context switch.
- Usually quite costly, but necessary for pre-emptive multitasking.
- (Alternatives: Co-operative multitasking (Win3.11), co-routines)

Memory layout 1/2

Process memory layout

- text = program code, read-only, executable
- data = initialized data, read-write
- bss = data initialized to zero, read-write (block started by symbol)

Try "size program" to show sizes of these blocks from an executable.

Memory layout 2/2

In addition running programs have:

- heap, for dynamic allocations
- stack, for function calls and stack allocations

"cat /proc/<pid>/maps" to view process detailed memory layout.
Placement depending on processor architecture, on Intel processors, stack grows down and heap grows up.
(vdso/vsyscall are Linux-specific tricks to improve system call performance.)

Memory allocation 1/5

With C, you handle memory allocation and freeing.
Memory allocation functions return a pointer to the memory on success, and NULL on failure. But beware:

- You can kill all performance if you go to swap.
- Linux memory system uses overcommitment; you can malloc() more memory than there actually is! If you do actually try to use all that memory, you get killed when system actually runs out of memory.

Memory allocation 2/5

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Memory allocation 3/5

Previous functions are actually front-ends to:

```
#include <unistd.h>

int brk(void *addr);
void *sbrk(intptr_t increment);
```

These functions merely extend the point where heap memory ends.

Memory allocation 4/5

Also sometimes useful, but not part of POSIX:

```c
#include <alloca.h>

void *alloca(size_t size);
```

Allocates memory from stack frame. Automatically freed when the function returns.
Beware of stack overflows. Usually better to stick to malloc().
C99 added Variable-Length Arrays, which does quite a similar function.

Memory allocation 5/5

Also often used for allocating memory:

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Memory is allocated by for example mapping /dev/zero to given addresses. Main function of mmap() is not memory allocation but in general mapping files to memory. More of this later.

Command-line arguments

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

argc	count of command-line arguments
argv	table of arguments

Command-line arguments from shell are expanded automatically.

```
#include <unistd.h>

int getopt(int argc, char * const argv[],
           const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

Environmental variables

```
#include <unistd.h>

extern char *environ[];
```

NULL-terminated table of strings containing NAME=VALUE pairs.

```
const char *getenv(const char *name);
int putenv(const char *string);
```

Process ID

Process IDs

- Process ID (pid) is a positive integer (type pid_t) that is unique a particular process, within the system.

```
pid_t getpid(void);
pid_t getppid(void);
```

Pid 1 is reserved for init, the first process started by the kernel. Also, processes that lose contact with their parents (direct parent dies) are transferred to be children of pid 1.

Neat tool: pstree

Credentials: overview

User IDs (uid)

- Positive integers that are mapped to user identities
- Traditionally users are defined in /etc/passwd

Group IDs (gid)

- Positive integers that are mapped to group identifiers
- Traditionally groups are defined in /etc/group

Tradition is extended via name service switch (nsswitch), implemented transparently inside libc.

Uids and gids are inherited. Changing uids/gids requires root privileges, and thus it's rarely done in runtime.

Credentials: User IDs

UID 0 is hardcoded for root user. Internally used as integers, symbols shown only for humans.

Each process actually has three uids:

- Real uid, inherited uid.
- Saved uid, original uid in case set-uid programs is run.
- Effective uid, determines the actual access.

Effective uid is the uid that has any actual effect. Saved and real uids are only used when checking if the process can change it's effective uid: any non-root process can only change it's uid to the values stored in real uid and saved uid. root may change to any uid.

Credentials: API

Users

```
int getuid(void);
int geteuid(void);
int setreuid(uid_t ruid, uid_t euid);
int setuid(uid_t uid);
int seteuid(uid_t uid);
```

Groups

```
int getgid(void);
int getegid(void);
int setregid(gid_t rgid, gid_t egid);
int setgid(gid_t gid);
int setegid(uid_t gid);
```

Credentials: Group IDs

Each process has one uid and one or more gids. One gid is primary and works like uid. Other gids are supplemental groups.

```
int setgroups(size_t num, const gid_t *list);
int getgroups(size_t num, gid_t *list);
```

Primary group in /etc/passwd, supplementals in /etc/group.

Process priority 1/2

Priority is a value that defines in which order processor time is distributed. If any higher priority task has work to do, it dominates all lower priority tasks.

```
#include <unistd.h>

int nice(int inc);
```

Adjust process "nice" value. Nice processes let others go first.

Process priority 2/2

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

which = PRIO_PROCESS / PRIO_PGRP / PRIO_USER who = process / process group / user id, respectively.

Resource usage

```c
int getrusage(int who, struct rusage *usage);
```

Resources used by the process.
See /usr/include/bits/resource.h
POSIX only defines ru_utime and ru_stime.

Resource limits

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

Check "ulimit -a"

Global parameters: System capabilities 1/2

```
#include <unistd.h>
long sysconf(int);
```

_SC_CLK_TCK number of kernel internal clock ticks per second

_SC_PAGESIZE Page size in bytes.

Global parameters: System capabilities 2/2

_SC_STREAM_MAX maximum number of C standard I/O streams a process can have open at once

_SC_ARG_MAX maximum length that the command line arguments and environmental variables can be used with any exec() functions.

_SC_OPEN_MAX maximum number of open files.

_SC_LINE_MAX maximum line length text-processing tools are required to take.

_SC_NGROUPS_MAX maximum number of supplemental groups a process can have

Global parameters: System information 1/2

```
#include <sys/utsname.h>

int uname(struct utsname *unameBuf);
```

Returned structure contains:

- Operating system name ("Linux")
- Node name, configured hostname
- OS Release (Linux has kernel version)
- OS Version (Linux has kernel timestamp)
- Hardware identifier (architecture)
- Domain name (GNU extension, has NIS/YP domain name)

Also command "uname".

Global parameters: System information 2/2

```
#include <unistd.h>

int sethostname(const char *name, size_t len);
int setdomainname(const char *name, size_t len);
```

Set nodename and domainname.

Note: these names don’t need to have any relation to names found in DNS.

Global parameters: System time and date 1/5

```
#include <time.h>

time_t time(time_t *t);
```

Unix time is defined as seconds from epoch; midnight Jan 1, 1970 UTC.
On most 32-bit UNIX systems, time_t is signed 32-bit integer. It will overflow on Monday, January 18, 2038. Expect failures.

Global parameters: System time and date 2/5

```
#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv, const struct timezone *tz);


struct timeval {
   int tv_sec;
   int tv_usec;
};

struct timezone {
   int tz_minuteswest;
   int tz_dsttime;
};
```

Global parameters: System time and date 3/5

```
#include <time.h>

int clock_getres(clockid_t clk_id, struct timespec *res);
int clock_gettime(clockid_t clk_id, struct timespec *tp);
int clock_settime(clockid_t clk_id, const struct timespec *tp);
```

```
struct timespec {
   time_t tv_sec;
   long tv_nsec;
};
```

gettimeofday() was obsoleted by clock_gettime() as of POSIX.1-2008.

Global parameters: System time and date 4/5

```
#include <time.h>

char *asctime(const struct tm *tm);
char *asctime_r(const struct tm *tm,char *buf);

char *ctime(const time_t *timep);
char *ctime_r(const time_t *timep,char *buf);

struct tm *gmtime(const time_t *timep);
struct tm *gmtime_r(const time_t *timep,struct tm *res);

struct tm *localtime(const time_t *timep);
struct tm *localtime_r(const time_t *timep,struct tm *res);
```

Global parameters: System time and date 5/5

Additional:

```
#include <time.h>

time_t mktime(struct tm *tm);

time_t timelocal(struct tm *tm);
time_t timegm(struct tm *tm);

size_t strftime(char *s, size_t max, const char *format,
                const struct tm *tm);
```

Is it over yet?

- Questions?
- Next time: Filesystem and file I/O.