Process Environment

System Programming

2019 여름 계절학기

한양대학교 공과대학 컴퓨터소프트웨어학부
홍석준

- How a program is executed
- How command-line arguments are passed to the new program
- What the typical memory layout looks like
  Different ways to terminate
- Environment variable, `longjmp/setjmp` functions, and process resource limits

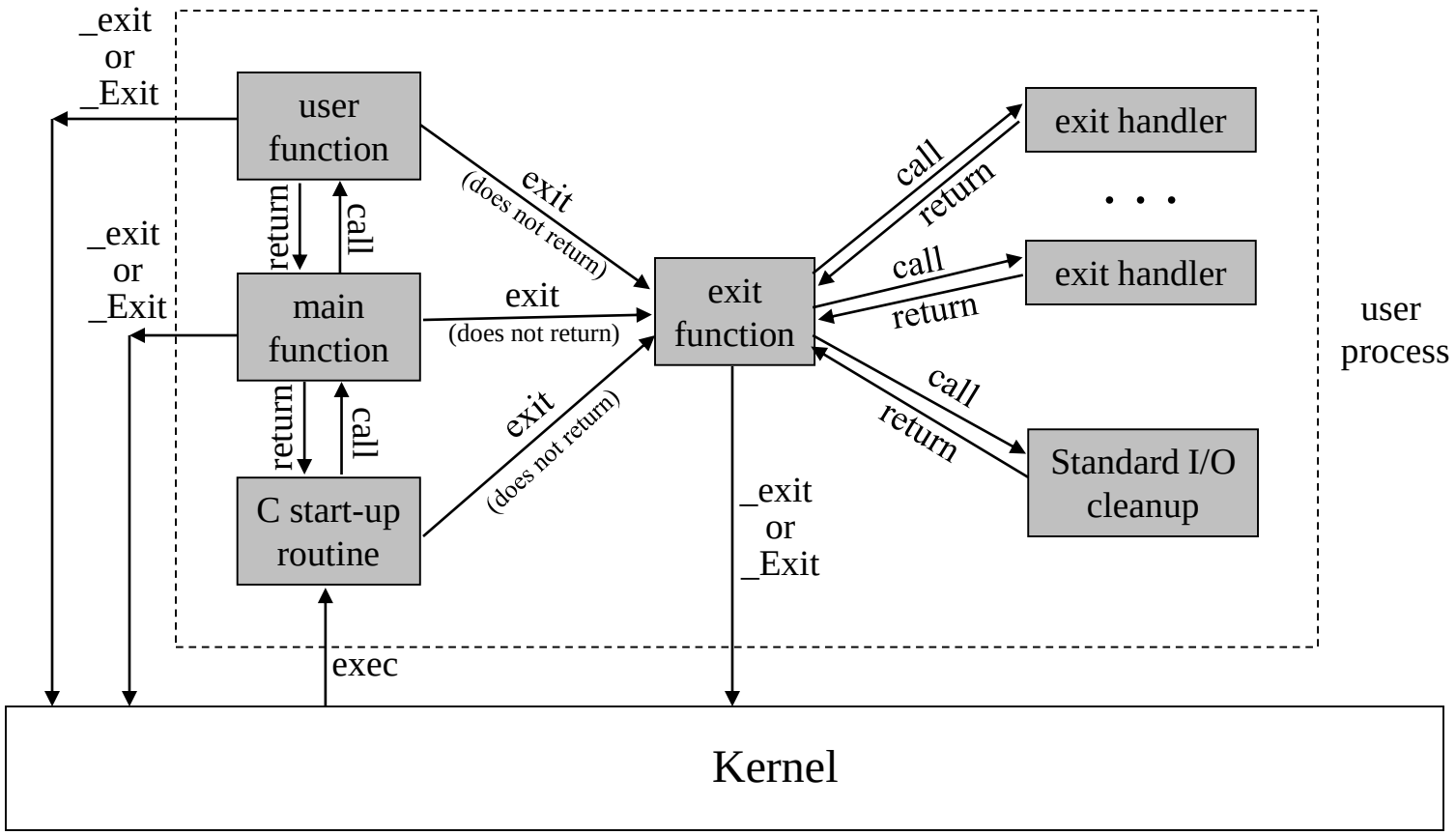
- `int main(int args, char *argv[])`
- **When executed by the kernel *exec*, a special start-up routine sets up things, including the command-line arguments, so that the `main` function can be called.**
- **Five ways for a process to terminate**
  - Normal termination: return from `main`, calling `exit`, and calling `_exit` or `_Exit`
  - Abnormal termination: calling `abort` and terminated by a signal

I. Process Termination

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);
#include <unistd.h>
void _exit(int status);
```

- **`exit` performs certain *cleanup processing* and then returns to the kernel, while `_exit`/`_Exit` returns to the kernel *immediately*.**
  - In the case of `exit`, `fclose` is called for all open streams.
- **The *exit status* is undefined if**
  - Any of these functions is called without an *exit status*,
  - `main` does a `return` without a return value, or
  - `main` is not declared to return an integer.

I. How a C Program is Started and Terminated



- **ISO C *exit handlers*: up to 32 functions that are automatically called by `exit`**

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

- **Called in reverse order of their registration**
- **Program 7.3**

I. Process Termination

```
#include "apue.h"
static void my_exit1(void);
static void my_exit2(void);
int main(void)
{
    if (atexit(my_exit2) != 0)
       err_sys("can't register my_exit2");
    if (atexit(my_exit1) != 0)
       err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    printf("main is done\n");
    return(0);
}
static void my_exit1(void) {
    printf("first exit handler\n");
}
static void my_exit2(void) {
    printf("second exit handler\n");
}
```

Program 7.3

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

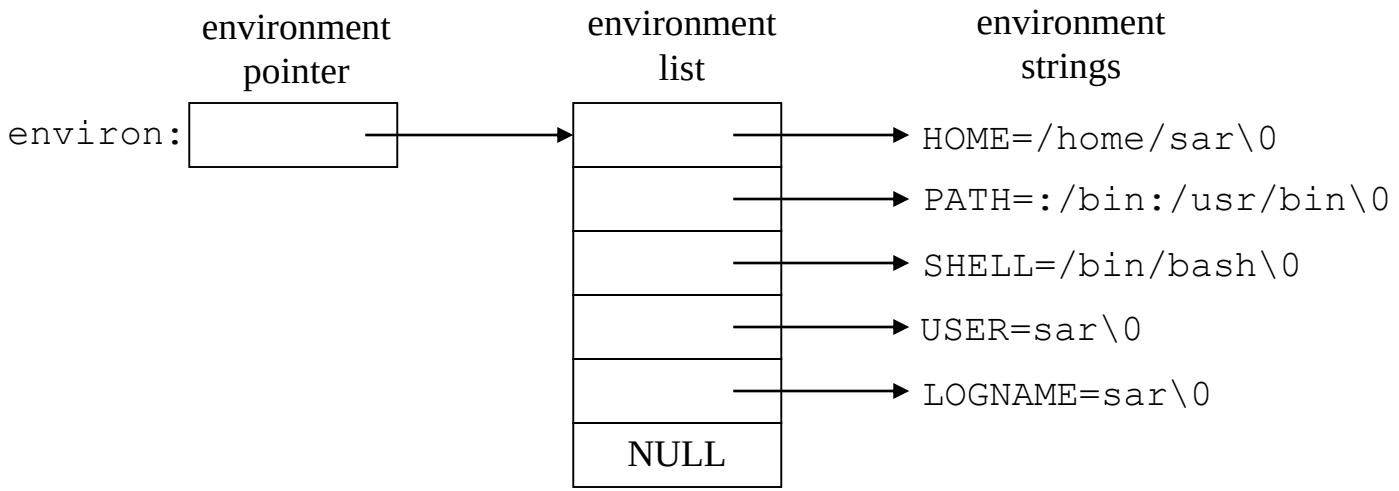
I. Command-Line Arguments

```
#include “apue.h”
int
main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < argc; i++)
        printf(“argv[%d]: %s\n”, i, argv[i]);
    exit(0);
}
```

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

I. Environment List

- Each program is passed an environment list, an array of char pointers.
- A global variable: `extern char **environ;`
- `getenv/putenv` functions



I. Memory Layout of a C Program

- **Text segment: machine instructions**
  - Sharable (a single copy for frequently executed programs such as text editors, C compiler, shells, etc.)
  - Often read-only
- **(Initialized) data segment: variables specifically initialized in the program**

```
int maxcount = 99;
```

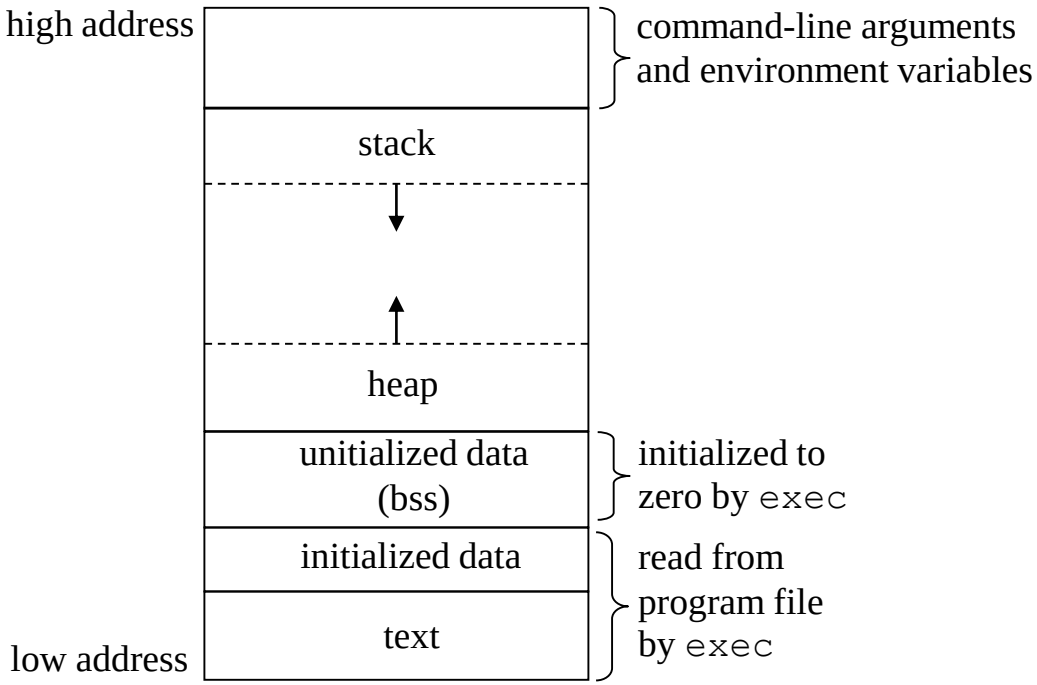
- **Uninitialized data segment (bss segment): initialized by the kernel to arithmetic 0 or null pointers**

```
long sum[1000];
```

- **Stack: return addr & certain info about the caller's environ (such as some of the machine registers), auto variables**
- **Heap: dynamic memory allocation**

I. Memory Layout of a C Program

```
$ size /bin/cc /bin/sh
text    data    bss     dec     hex
81920   16384   664     98968   18298   /bin/cc
90112   16384   0       106496  1a000   /bin/sh
```



I. Shared Libraries

- Static library (*.a) vs. dynamic library (*.so)
- A single copy of the library routine somewhere in memory
- Reduces executable size at the cost of run-time overhead (the first time shared libraries are called)
- Can be replaced with new version without re-linking

```
$ ldd tcsh
libsocket.so.1 =>      /usr/lib/libsocket.so.1
libnsl.so.1 =>    /usr/lib/libnsl.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
libmp.so.2 =>     /usr/lib/libmp.so.2
```

- LD_LIBRARY_PATH

I. Memory Allocation

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
void free(void *ptr);
```

- **malloc**
  - allocates a specified number of bytes of memory
  - initial value is indeterminate.
- **calloc**
  - allocates space for a specified number of objects of a specified size.
  - initialized to all 0 bits.
- **realloc**
  - changes the size of a previously allocated area.
  - The initial value of increased space is *indeterminate*.

❑ Additional space for record keeping

- Blk size, a ptr to the next blk, and the like. Overwriting this record-keeping information often leads to a catastrophe.
- Freeing an already freed block or a block not obtained from the allocation functions – can also be fatal.

❑ Memory leakage

❑ `void *alloca(size_t size)`

- It allocates size bytes of space in ***the stack frame***.
- No need to free the space.
  - automatically freed when the function from which alloca() is called returns.

I. Environment Variables

❑ Environment variables

- *name=value*

❑ **Defined environment variables**

- HOME, LANG, LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, LOGNAME, NLSPATH, SHELL, PATH, TERM, TZ

❑ **csh built-in commands: env, setenv, unsetenv**

```
#include <stdlib.h>

char *getenv(const char *name); /* ISO C */
```

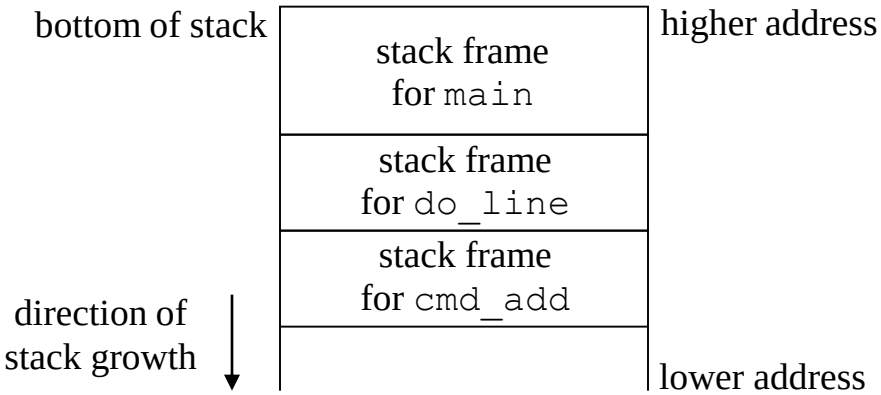
I. Environment Variables

```
#include <stdlib.h>
int putenv(const char *str);
int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
```

- **`putenv` takes a string of the form *name*=*value*. If *name* already exists, its old definition is first removed.**
- **`setenv` sets *name* to *value*.**
  - If *name* already exists, then
    - If *rewrite* is nonzero, the existing definition is first removed.
    - Otherwise, no effect.
- **`unsetenv` removes any definition of *name*.**

I. setjmp and longjmp Functions

- **Useful for handling error conditions in a deeply nested function call.**
- **Program 7.9**



- **Nonlocal goto to handle nonfatal errors**
  - If the `cmd_add` encounters an error, it might want to ignore the rest of the line, and return to the `main` to read the next input line.
  - A chain of return values?

I. setjmp and longjmp Functions

```
#include "apue.h"
#define TOK_ADD 5
void do_line(char *);
void cmd_add(void);
int get_token(void);
int main(void) {
    char line[MAXLINE];
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}
char *tok_ptr; /* global pointer for get_token() */
void do_line(char *ptr) /* process one line of input */
{
    int cmd;
    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) { /* one case for each command */
            case TOK_ADD:
                cmd_add();
                break;
        }
    }
}
```

```
void cmd_add(void)
{
    int token;
    token = get_token();
    /* rest of processing for this command */
}

int get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

Program 7.9

I. setjmp and longjmp Functions

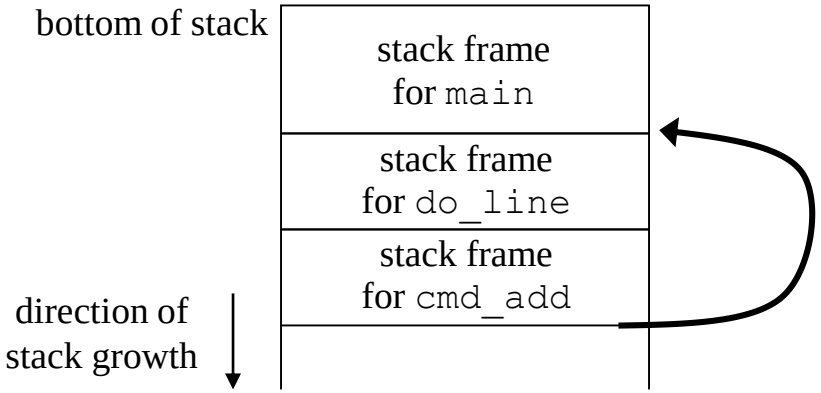
- **Nonlocal goto (setjmp/longjmp) as opposed to C goto statement within a function.**

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

- **setjmp returns 0 if called directly, nonzero if returning from a call to longjmp.**
- **jmp_buf is some form of array holding all the information required to restore the status of the stack.**
- **env is a global variable.**

I. setjmp and longjmp Functions

- **longjmp(env, val)**
  - The same `env` used in a call to `setjmp`.
  - A nonzero value `val` becomes the return value from `setjmp`.
- **A longjmp call in cmd_add**



I. setjmp and longjmp Functions

```
#include “apue.h”
#include <setjmp.h>

jmp_buf   jmpbuffer;

int main(void) {
        char   line[MAXLINE];

        if (setjmp(jmpbuffer) != 0)
                printf(“error”);
        while (fgets(line, MAXLINE, stdin) != NULL)
                do_line(line);
        exit(0);
}

…

void cmd_add(void) {
        int   token;

        token = get_token();
        if (token < 0)                    /* an error has occurred */
                longjmp(jmpbuffer, 1);
        /* rest of processing for this command */
}
```

I. `setjmp` and `longjmp` Functions

- **What are the states of the automatic vars and register vars in the `main` function?**
  - Rolled back or left alone? The answer is “indeterminate.”
  - If you don’t want your automatic var rolled back, define it with the `volatile` attribute. If you want it “left alone”, declare it global or static.

- **Program 7.13 on a system with vars in registers rolled back and vars in memory left alone.**

```
$ cc testjmp.c; ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ cc –O testjmp.c; ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

I. setjmp and longjmp Functions

```
#include "apue.h"
#include <setjmp.h>
static void f1(int, int, int, int);
static void f2(void);
static jmp_buf jmpbuffer;
static int globval;
int main(void)
{
    int autoval;
    register int regival;
    volatile int volaval;
    static int statval;
    globval = 1; autoval = 2; regival = 3; volaval = 4;
    statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d,
                regival = %d, volaval = %d,
                statval = %d\n", globval, autoval,
                regival, volaval, statval);
        exit(0);
    }
```

```
/* Change variables after setjmp, but before longjmp.*/
    globval = 95; autoval = 96; regival = 97;
    volaval = 98; statval = 99;

    /* never returns */
    f1(autoval, regival, volaval, statval);
    exit(0);
}

static void f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,
            volaval = %d, statval = %d\n",
            globval, i, j, k, l);
    f2();
}
static void f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

I. getrlimit and setrlimit

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);

struct rlimit {
 rlim_t rlim_cur; /* soft limit: current limit */
 rlim_t rlim_max; /* hard limit: max value for rlim_cur */
}
```

- *resource*: **RLIMIT_AS, RLIMIT_CORE, RLIMIT_CPU, RLIMIT_DATA, RLIMIT_FSIZE, RLIMIT_LOCKS, RLIMIT_MEMLOCK, RLIMIT_NOFILE, RLIMIT_NPROC, RLIMIT_RSS, RLIMIT_SBSIZE, RLIMIT_STACK, RLIMIT_VMEM**
- **The resource limits for a process are established by process 0 on boot and then inherited by each successive process.**

Q and A