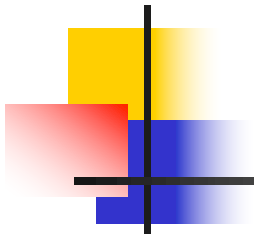
Chapter 7.
Unix Process Environment

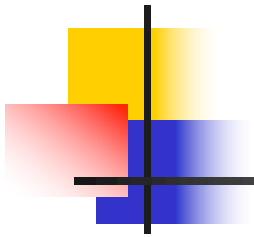
朱金辉

华南理工大学软件学院



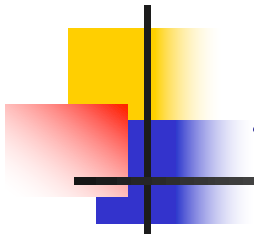
1. Introduction

- How is main() called?
- How are arguments passed?
- Memory layout?
- Memory allocation?
- Environment variables
- Process termination



2. main Function

- int main(int argc, char *argv[]);
- arc = #arguments
- argv[] = arguments
- Kernel executes a special START-UP routine before main()
- Start-up routine sets things up before main() is called: stack, heap, etc.

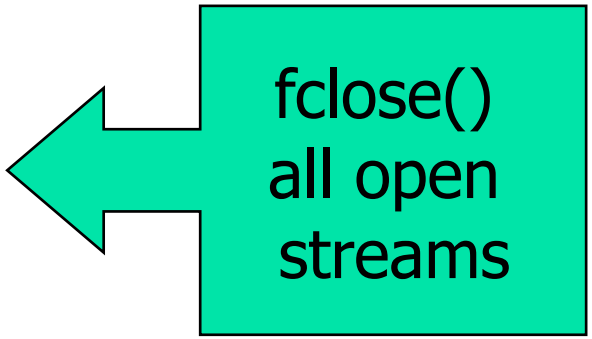


3. Process Termination

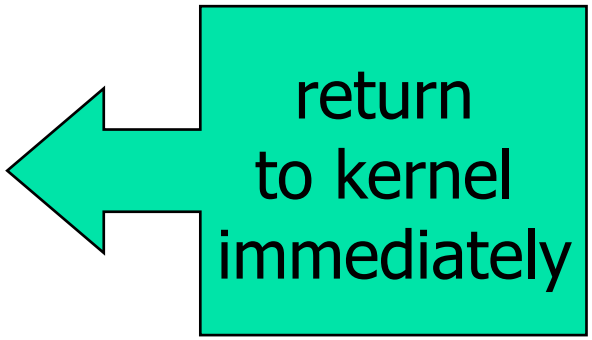
- Normal termination:
  - return from main()
  - calling exit()
  - calling _exit()
  - Return of the last thread from its start routine (Section 11.5)
  - Calling pthread_exit (Section 11.5) from the last thread
- Abnormal termination
  - calling abort() (Section 11.7)
  - terminated by a signal (Section 10.2)
  - Response of the last thread to a cancellation request (Sections 11.5 and 12.7)

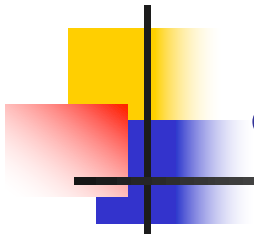
exit() and _exit()

- #include <stdlib.h>
- void exit(int *status*);



- #include <unistd.h>
- void _exit(int *status*);

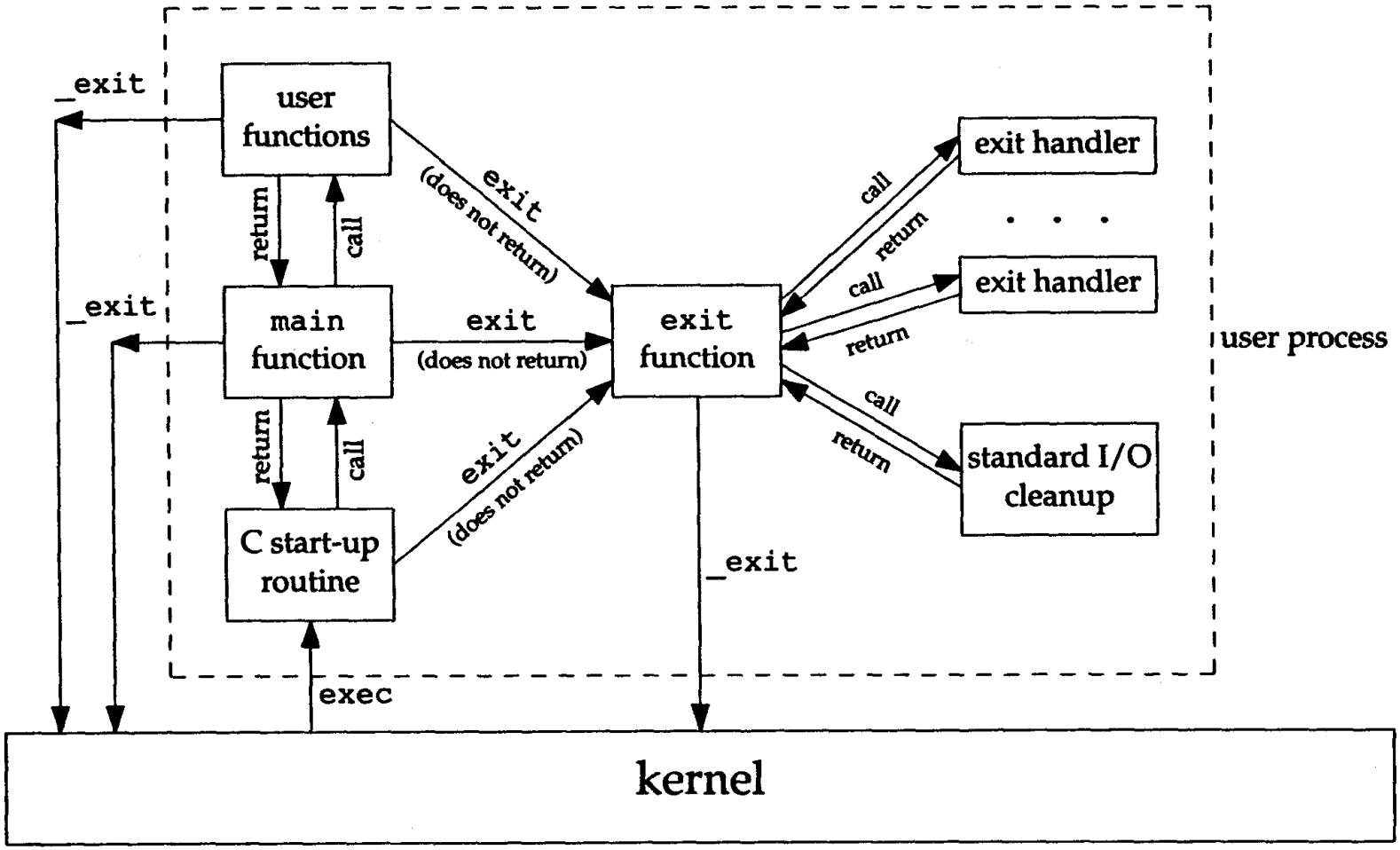




atexit(): Exit Handler

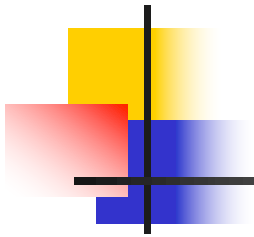
- #include <stdlib.h>
- int atexit(void (*_func_) (void));
- Returns: 0 if OK, nonzero on error
- *func* is an exit handler
- exit() calls these exit handler functions in the reverse order of registration
- #times called = #times registered

Program Start & Termination



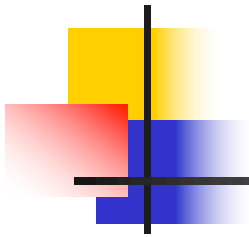
Program 7.3: Exit Handlers

```
#include        "apue.h"
static void     my_exit1(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: results

- **$ a.out**
- main is done
- first exit handler
- first exit handler
- second exit handler

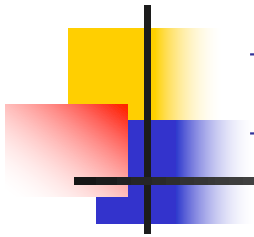


4. Command-Line Arguments

- exec() can pass command-line arguments to a new program
- Part of normal operation of Unix shells
- argv[argc] is NULL (ANSI, POSIX.1)

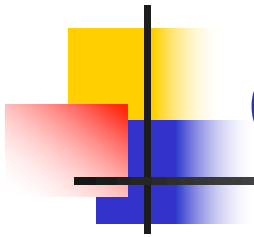
Program 7.4: echo()

```
#include        "apue.h"

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

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



Program 7.4: results

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



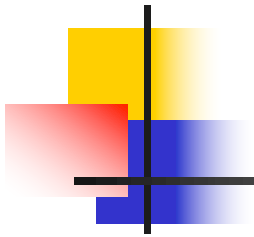
getopt

- #include <unistd.h>
- int getopt(int *argc*, char * const *argv*[],
  const char **options*);
- extern int optind, opterr, optopt;
- extern char *optarg;
- Returns: the next option character, or
  −1 when all options have been processed

- Example:
  - command [-i] [-u username] [-z] filename
  - pass "iu:z" as the *options*

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int flags, opt;
    int nsecs, tfnd;

    nsecs = 0;
    tfnd = 0;
    flags = 0;
    while ((opt = getopt(argc, argv, "nt:")) != -1)
    {
        switch (opt)
        {
        case 'n':
            flags = 1;
            break;
        case 't':
            nsecs = atoi(optarg);
            tfnd = 1;
            break;
        default: /* '?' */
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }
    printf("name argument = %s\n", argv[optind]);
    /* Other code omitted */
    exit(EXIT_SUCCESS);
}
```

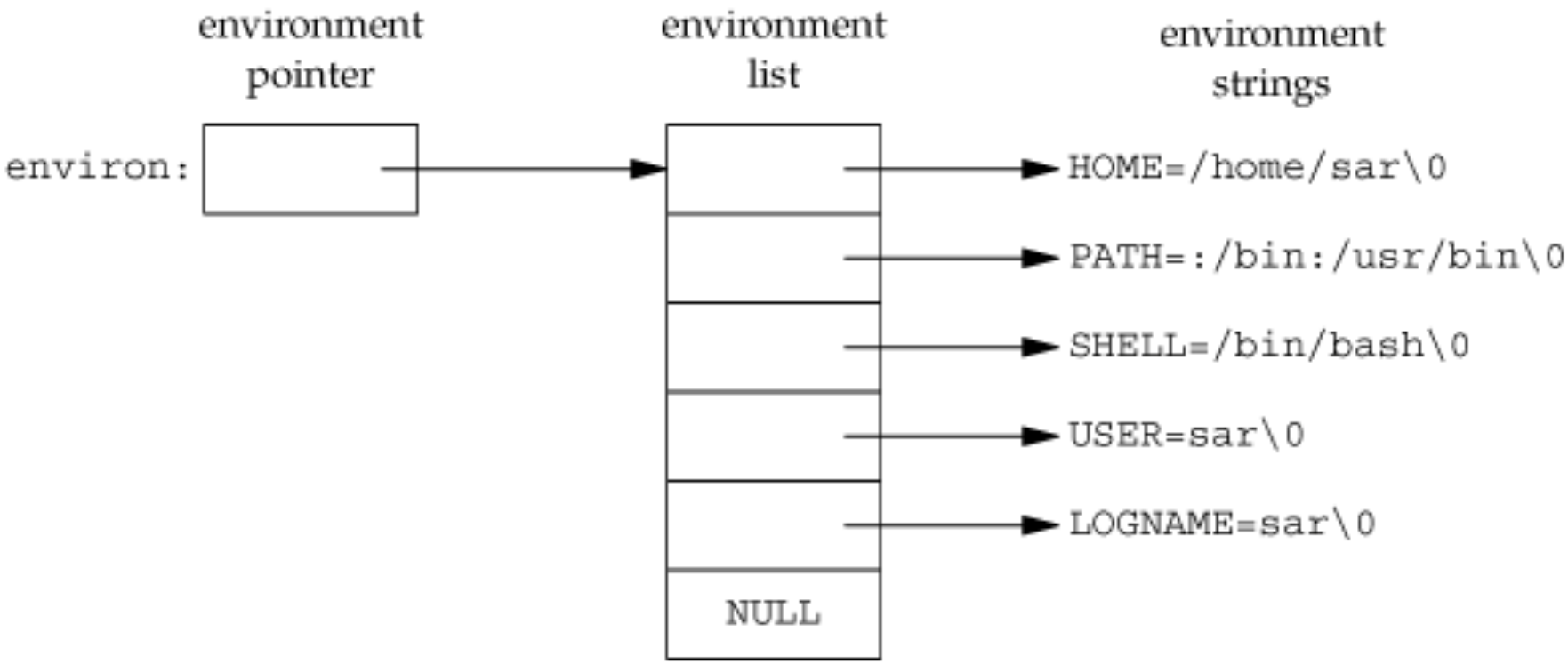
- Optarg: If an option takes an argument, getopt sets optarg to point to the option's argument string when an option is processed.
- Optind: The index in the argv array of the next string to be processed. It starts at 1 and is incremented for each argument processed by getopt.

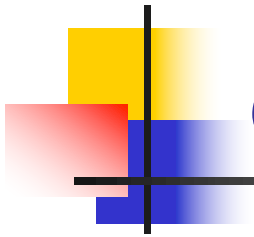


5. Environment List

- An array of character pointers (null-terminated C strings)
- Array address is in global variable environ
- extern char **environ;
- getenv(): get an environment string
- putenv(): set an environment string

Environment List (Fig. 7.5)

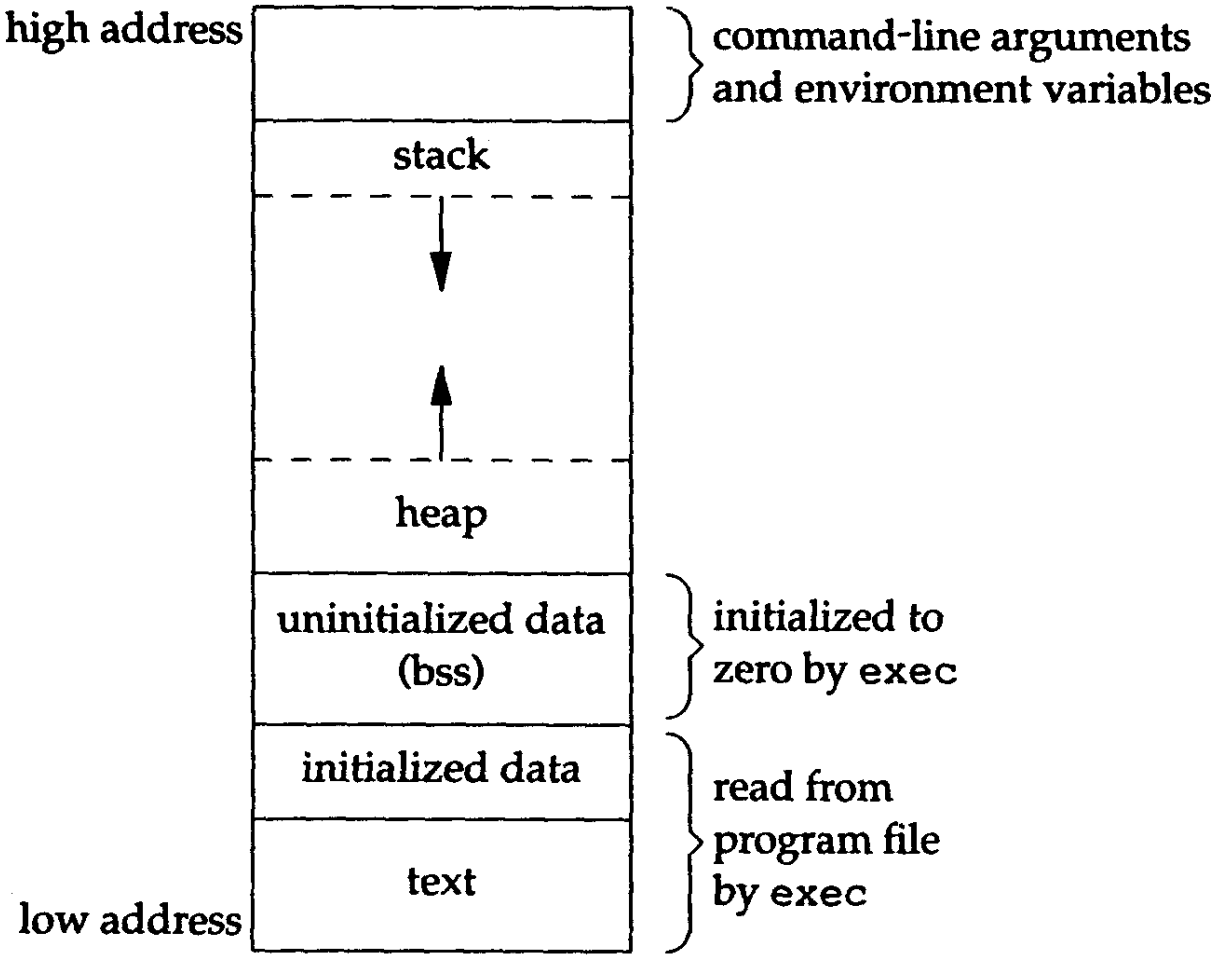


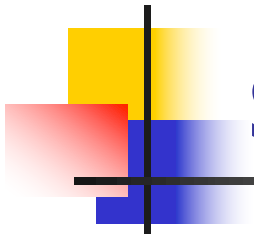


6. Memory Layout of a C Program

- **Text segment**: Machine instructions (read-only, sharable)
- **Initialized data segment**: e.g. int maxcount = 99; (initialized!)
- **Uninitialized data segment**: (bss: block started by symbol) e.g. long sum[1000];
- **Stack**: automatic variables, function calling information, context-switch information, (recursive functions)
- **Heap**: dynamic memory allocation

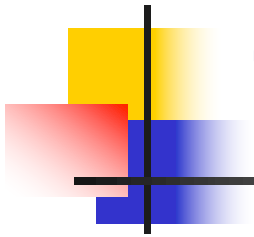
Memory Layout (Fig. 7.6)





size

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



7. Shared Libraries

- Common library routines removed from executable files
- Single copy of common library routines in memory is maintained
- No need to re-link edit every program if a library is updated or changed
- Size is smaller, some run-time overhead

Shared Libraries

Without Shared Libraries

$ ls –l a.out

-rwxrwxr-x 1 stevens 104859 Aug 2 14:25 a.out

$ size a.out

text data bss dec hex

49152 49152 0 98304 18000

With Shared Libraries

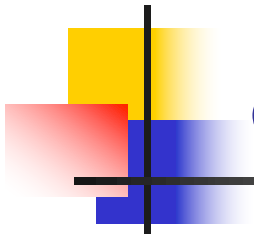
$ ls –l a.out

-rwxrwxr-x 1 stevens 24576 Aug 2 14:26 a.out

$ size a.out

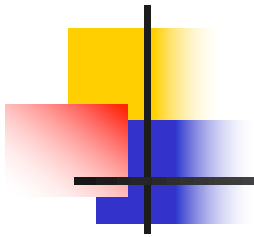
text data bss dec hex

8192 8192 0 16384 4000



8. Memory Allocation

- malloc():
  - allocates specified #bytes,
  - initial value of memory is indeterminate
- calloc():
  - allocates specified #objects of specified size,
  - initialized to all 0 bits
- realloc():
  - changes size of previously allocated memory,
  - initial value of new area is indeterminate

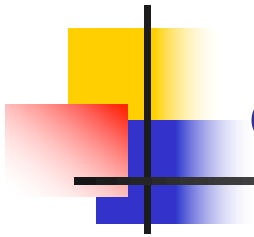


Memory Allocation

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

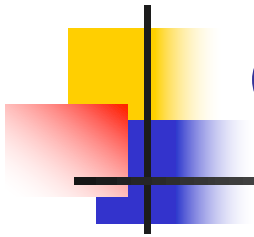
Return: nonnull pointer if OK,
NULL on error

```
void free(void *ptr);
```



alloca()

- Allocates memory from stack, instead of heap
- Advantage: No need to free space, automatically freed after function returns
- Disadvantage: Some systems do not support alloca()



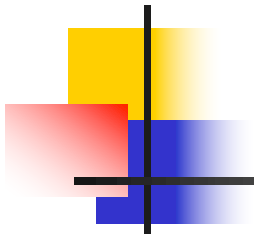
9. Environment Variables

```
#include <stdlib.h>

char *getenv(const char *name);
```

- Returns: pointer to value associated with name, NULL if not found
- Some environment variables are set automatically by shell upon login
- E.g.: HOME, USER, etc.

Environment Variables (Fig. 7.7)

Variable	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Description
COLUMNS	•	•	•	•	•	terminal width
DATEMSK	XSI		•		•	getdate(3) template file pathname
HOME	•	•	•	•	•	home directory
LANG	•	•	•	•	•	name of locale
LC_ALL	•	•	•	•	•	name of locale
LC_COLLATE	•	•	•	•	•	name of locale for collation
LC_CTYPE	•	•	•	•	•	name of locale for character classification
LC_MESSAGES	•	•	•	•	•	name of locale for messages
LC_MONETARY	•	•	•	•	•	name of locale for monetary editing
LC_NUMERIC	•	•	•	•	•	name of locale for numeric editing
LC_TIME	•	•	•	•	•	name of locale for date/time formatting
LINES	•	•	•	•	•	terminal height
LOGNAME	•	•	•	•	•	login name
MSGVERB	XSI	•			•	fmtmsg(3) message components to process
NLSPATH	XSI	•	•	•	•	sequence of templates for message catalogs
PATH	•	•	•	•	•	list of path prefixes to search for executable file
PWD	•	•	•	•	•	absolute pathname of current working directory
SHELL	•	•	•	•	•	name of user's preferred shell
TERM	•	•	•	•	•	terminal type
TMPDIR	•	•	•	•	•	pathname of directory for creating temporary files
TZ	•	•	•	•	•	time zone information

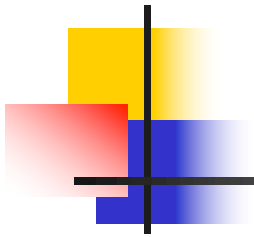


Setting an environment variable

```
#include <stdlib.h>

int putenv(const char *str);

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

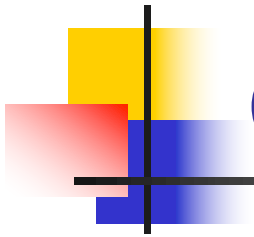
- Return: 0 if OK, nonzero on error

```
void unsetenv(const char *name);
```



10. setjmp(), longjmp() Functions

- In C, we cannot goto a label in another function
- setjmp() and longjmp() must be used
- See Program 7.9 (a skeleton) for command processing
  - read commands,
  - determine commands
  - call functions to process each command



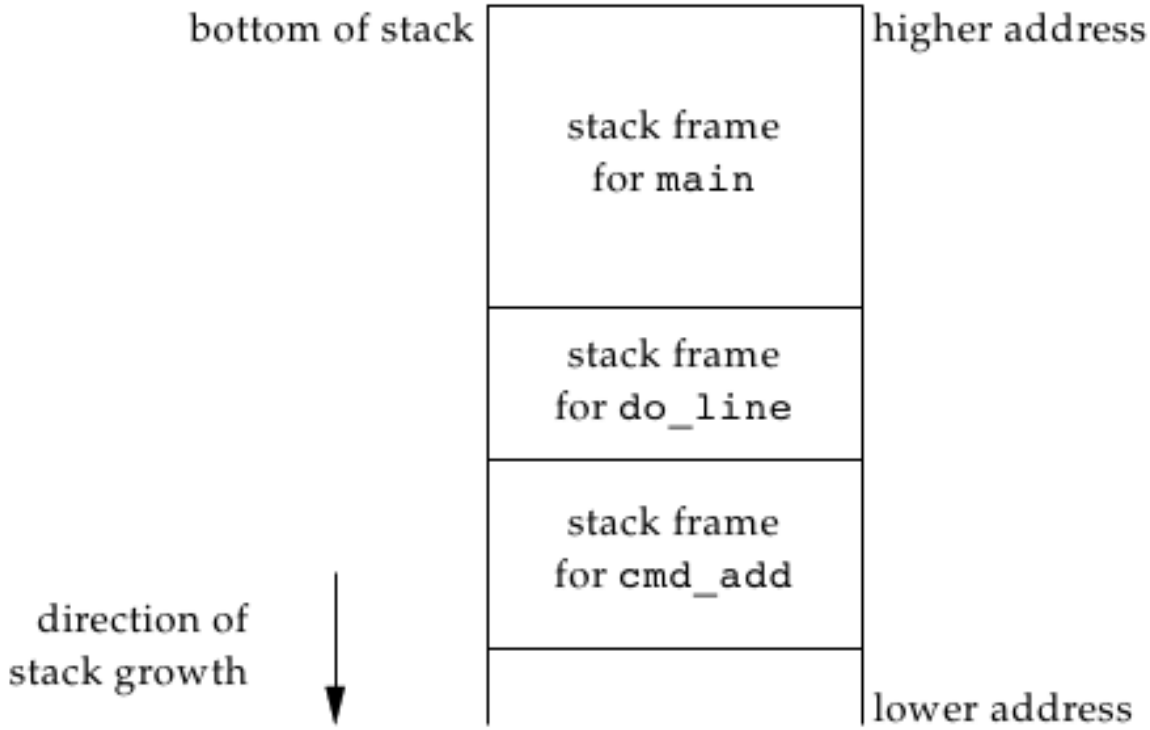
cmd1.c

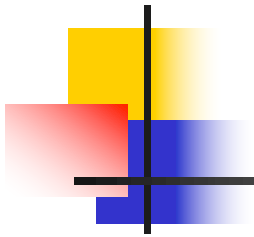
```
#define TOK_ADD 5
void do_line(char *);
void cmd_add (void);
int get_token(void);

int main(void) {
  char line[MAXLINE];
  while (fgets(line,MAXLINE,stdin )!=NULL)
        do_line(line);
  exit(0);
}

char *tok_ptr;
```

```
do_line(char *ptr) {
  int cmd;
  tok_ptr=ptr;
  while ((cmd=get_token())>0) {
    switch (cmd) {
    case TOK_ADD:
      cmd_add();
      break;
    }
  }
}
```

```
void cmd_add(void) {
  int token;
  token=get_token();
  /* rest code */
}
int get_token(void) {
   /*fetch next token */
}
```

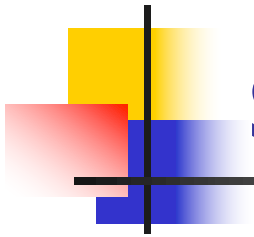
After cmd_add(): stack frame





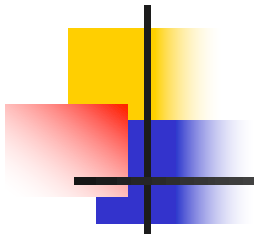
setjmp() and longjmp()

- Often we are deeply nested,
- An error occurs,
- We want to print an error, ignore rest of input, and return to main()
- Large # of levels → handle return at each level for each error
- Direct nonlocal goto: setjmp, longjmp



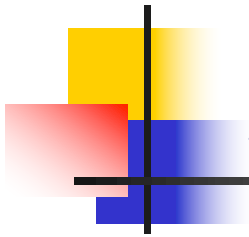
setjmp() and longjmp()

- #include <setjmp.h>
- int setjmp(jmp_buf *env*);
- Returns: 0 if called directly, nonzero if returning from a call to longjmp
- void longjmp(jmp_buf *env*, int *val*);



Program 7.11

- setjmp(jmpbuffer) stores current state of main at the start of program exec
- longjmp(jmpbuffer, 1) unwounds the stacks of do_line()and cmd_add()
- and causes setjmp() to return 1



Automatic, Register, Volatile Variables

- After longjmp(), what are the values of the automatic and register variables?
  - Rolled back
  - Left alone
- Standards: indeterminate
- Volatile variables: don’t rollback values
- Global, static variables: leave alone

Program 7.13: longjmp() …

```
#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. */
```

Program 7.13: longjmp() …

```
globval = 95; autoval = 96; regival = 97; volaval = 98; statval = 99;
    f1(autoval, regival, volaval, statval); /* never returns */
    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);
}
```

Program 7.13: results

compile without any optimization

```
$ 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
```

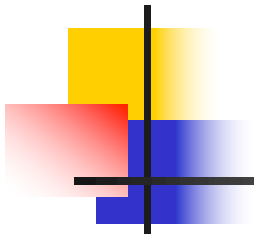
compile with full optimization

```
$ 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
```

Program 7.14: Incorrect usage of automatic variables

```
#include   <stdio.h>

#define    DATAFILE          "datafile"

FILE *
open_data(void)
{
    FILE  *fp;
    char   databuf[BUFSIZ];
                    /* setvbuf makes this the stdio buffer */

    if ( (fp = fopen(DATAFILE, "r")) == NULL)
            return(NULL);

    if (setvbuf(fp, databuf, BUFSIZ, _IOLBF) != 0)
            return(NULL);

    return(fp);                     /* error */
}
```

local automatic variable!

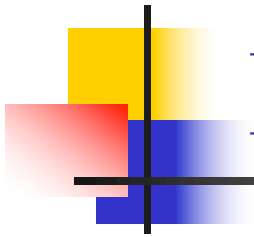


11. getrlimit(), setrlimit()

- Every process has resource limits

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

Return: 0 if OK, nonzero on error

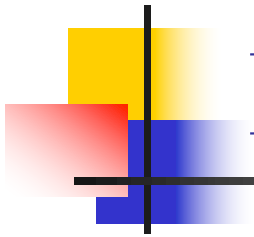


Resource Limits

- struct rlimit {
  - rlim_t rlim_cur; /* soft limit: curr limit */
  - rlim_t rlim_max; /* hard limit: max */

};

- Soft limit: can be changed by any process to <=hard limit
- Hard limit: can be changed by any process to >= soft limit (irreversible!)
  - can be raised only by superuser process



Resource Limits Example

- RLIMIT_AS
  - The maximum size in bytes of a process’ s total available memory. This affects the sbrk function (Section 1.11) and the mmap function (Section 14.8).
- RLIMIT_NPROC
  - The maximum number of child processes per real user ID.

Program 7.16: resource limits

```
#include  <sys/types.h>
#include  <sys/time.h>
#include  <sys/resource.h>
#include  "apue.h"

#define   doit(name)          pr_limits(#name, name)

static void         pr_limits(char *, int);

int main(void){
    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);
#ifdef    RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif
#ifdef    RLIMIT_NOFILE       /* SVR4 name */
    doit(RLIMIT_NOFILE);
#endif
#ifdef    RLIMIT_OFILE        /* 44BSD name */
    doit(RLIMIT_OFILE);
#endif
#ifdef    RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif
```

```
#ifdef RLIMIT_RSS
   doit(RLIMIT_RSS);
#endif
   doit(RLIMIT_STACK);
#ifdef RLIMIT_VMEM
   doit(RLIMIT_VMEM);
#endif
   exit(0);
}
static void pr_limits(char *name, int resource) {
   struct rlimit    limit;

   if (getrlimit(resource, &limit) < 0)
       err_sys("getrlimit error for %s", name);
   printf("%-14s  ", name);
   if (limit.rlim_cur == RLIM_INFINITY)
       printf("(infinite)  ");
   else
       printf("%10ld  ", limit.rlim_cur);
   if (limit.rlim_max == RLIM_INFINITY)
       printf("(infinite)\n");
   else
       printf("%10ld\n", limit.rlim_max);
}
```

Program 7.16: 4.3 BSD results

- $ a.out
- RLIMIT_CORE (infinite) (infinite)
- RLIMIT_CPU (infinite) (infinite)
- RLIMIT_DATA 8388608 16777216
- RLIMIT_FSIZE (infinite) (infinite)
- RLIMIT_MEMLOCK (infinite) (infinite)
- RLIMIT_OFILE 64 (infinite)
- RLIMIT_NPROC 40 (infinite)
- RLIMIT_RSS 27070464 27070464
- RLIMIT_STACK 524288 16777216