

# C Programs

- Preprocessor commands
  - #ifdef
  - #include
  - #define
- Type and macro definitions
- Variable and function declarations
- Exactly one main function
- Function calls and other executable statements
- Have a .c extension

# C Header Files

- Have a .h extension
- Usually contain only :
  - macro
  - type definitions
  - defined constants
  - function declarations
- Include header files with the `#include` preprocessor command

# Process

- Instance of a program in execution
- Each instance has its own address space and execution state
- Has a process ID and a state
- The OS manages memory allocated to the process
- Process has a flow of control called a thread
- Heavyweight process

# Variable Lifetimes

- Process variables that remain in effect for the lifetime of the process are allocated to static storage
- Process variables that are automatically created when execution enters a block and are destroyed when execution leaves the block are allocated to automatic storage

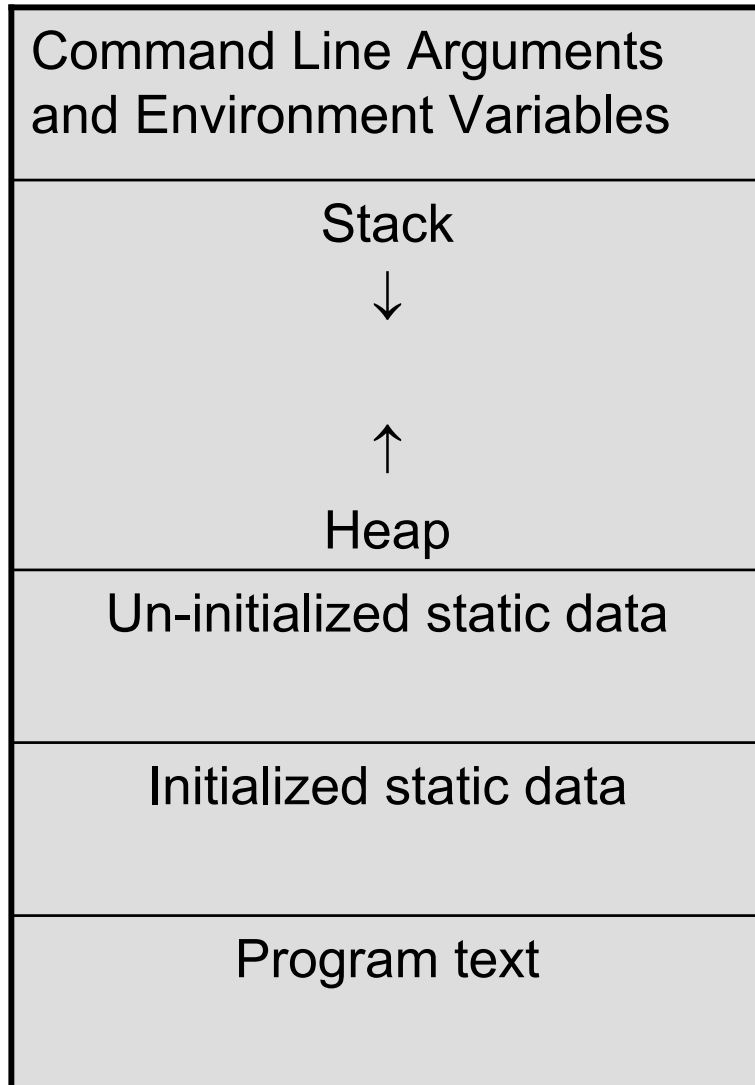
# Threads

- When a program executes, the pc determines which process instruction is executed next
- The resulting stream of instructions are called a thread of execution

# Multiple Threads

- A natural extension is to have multiple threads within the same process
- Each thread can be an independent task to complete the process executed in parallel
- Each thread is an abstract data type that has its own:
  - execution stack
  - pc value
  - register set
  - state
- Threads have low overhead and are frequently called lightweight processes.

# Program Layout



← argc, argv, environment

← Activation records for function calls  
(return address, parameters, saved registers, automatic variables)

← allocations from malloc family

# Activation Record

- Allocated at top of process stack
- Holds execution context (data) of a function call
- Removed from stack at end of function call
- Contains:
  - return address
  - stack management information
  - parameters
  - automatic variables

# Static Variable Example

- Version 1:

- `int myarray[5000] = {1,2,3,4};`
- `void main(int argc, char *argv[])`
- `{ myarray[0] = 3; }`

- Version 2:

- `int myarray[5000];`
- `void main(int argc, char *argv[])`
- `{ myarray[0] = 3;`

- Do an `ls -l` after compiling both versions. If integer is 4 bytes, version 1 executable is roughly 20,000 bytes larger.

# Static Variables and Thread Safe

- Static variables can make a program unsafe for threaded execution.
- Readdir uses a static variable to hold return values.
- This strategy is also used for client/server stubs when marshaling/un-marshaling arguments for remote procedure calls.
- Therefore, avoid static variables in a threaded environment wherever possible.

# Safe Functions

- Thread-Safe – Can be invoked concurrently or by multiple threads.
- Async-Signal-Safe – Can be called without restriction from a signal handler.

These terms replace the older notion of reentrant function.

# Function Errors

- Many functions return  $-1$  on error and set external parameter `errno` to appropriate error code
- Include `errno.h` in order to access the symbolic names associated with `errno`
- Use `errno` only immediately after an error is generated

# close Example

## SYNOPSIS

```
#include <unistd.h>  
int close(int fildes);
```

POSIX

Returns 0 if successful or  $-1$  if unsuccessful and sets `errno`

errno	cause
EBADF	fildes is not valid
EINTR	close was interrupted by a signal

# perror

- Outputs message string to standard error followed by the error message from the last system or library call that produced an error
- Place perror immediately after the occurrence of an error

## SYNOPSIS

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

```
void perror(const char *s)
```

POSIX: CX

# close Example with perror

```
#include <unistd.h>
```

```
int filedes;
```

```
if (close(filedes) == -1)
```

```
    perror ("Failed to close the file");
```

# perror Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int fd;

void main(void)
{
int fd;

if ((fd = open("my.file", O_RDONLY)) == -1)
    perror("Unsuccessful open of my.file");
}
```

# errno

- Test errno only if a function call returns an error
- errno can be set to various values depending on the function it is used with
- For example, when used with the function open, the value EAGAIN indicates the file is locked

# perror/errno Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
int fd;

void main(void)
{
int fd;

while (((fd = open("my.file", O_RDONLY)) == -1)
&& (errno == EAGAIN))
;
if (fd == -1)
    perror("Unsuccessful open of my.file");
}
```

# strerror

- Use `strerror` instead of `perror` to format a message that contains variable values
- The message output depends on the value of `errno`

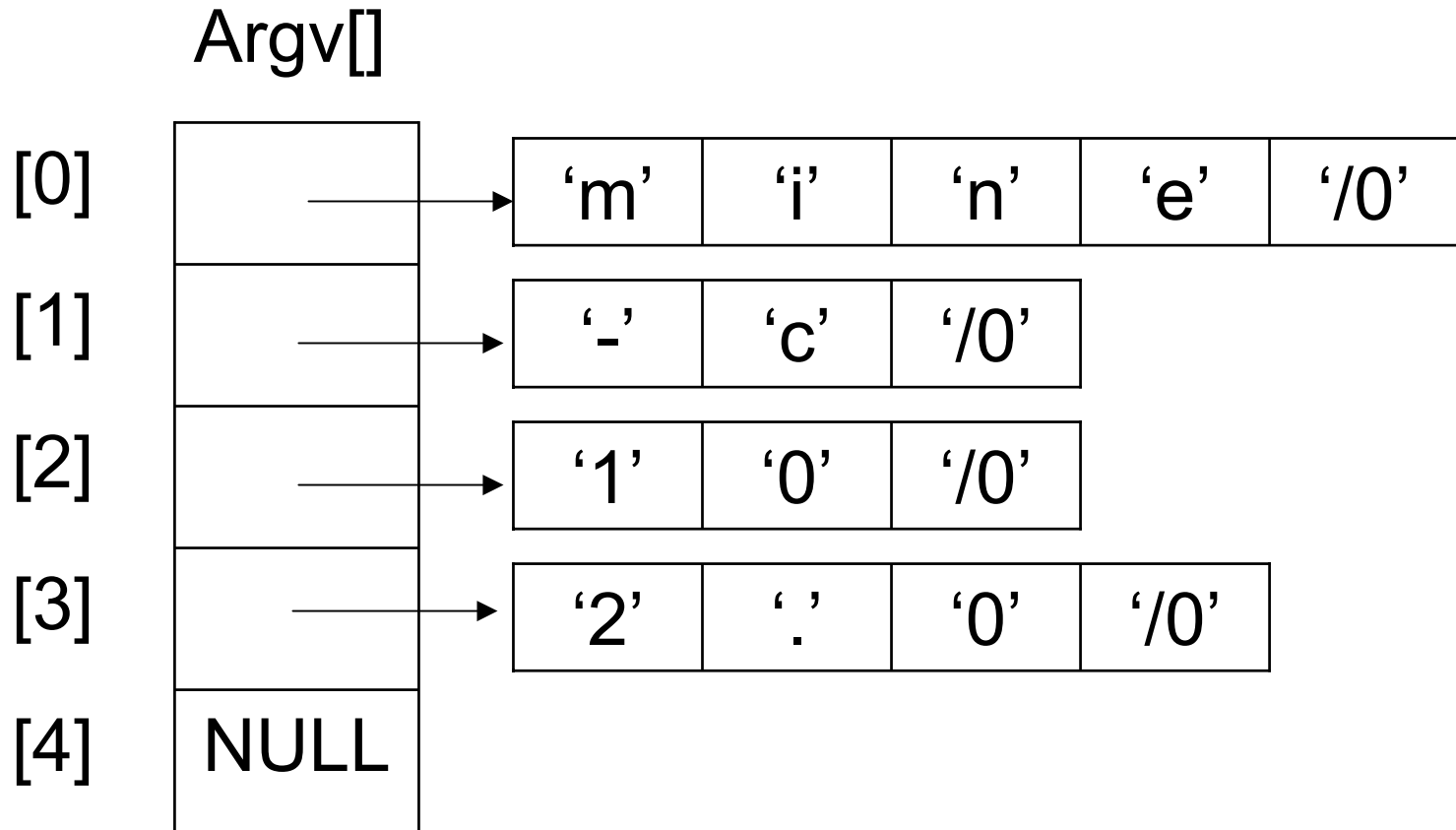
# strerror Example

```
#include ...  
void main(int argc, char *argv[])  
{  
    int fd;  
  
    if ((fd = open(argv[1], O_RDONLY)) == -1)  
        fprintf(stderr, "Could not open file %s: %s\n",  
                argv[1], strerror(errno));  
  
}
```

# Suggestions

- Make use of return values
- Do not exit early from functions – use error value from function instead
- Make functions general but usable (conflict?)
- Do not assume buffer sizes
- Use standard system defined limits rather than arbitrary constants
- Don't re-invent the wheel
- Don't modify input parameters unless necessary
- Don't use static (global) variables if automatic (local) variables will do just as well
- Be sure to free memory allocated by malloc
- Consider recursive calls to functions
- Analyze consequences of interrupts
- Careful plan the termination of each program component

# argv array for mine -c 10 2.0



# makeargv, first half

```
#include ...
/*
 * Make argv array (*arvp) for tokens in s which are separated by
 * delimiters. Return -1 on error or the number of tokens otherwise.
 */
int makeargv(char *s, char *delimiters, char ***arvp)
{ char *t;
  char *snew;
  int numtokens;
  int i;
  /* snew is real start of string after skipping leading delimiters */
  snew = s + strspn(s, delimiters);
  /* create space for a copy of snew in t */
  if ((t = calloc(strlen(snew) + 1, sizeof(char))) == NULL) {
    *arvp = NULL;
    numtokens = -1; }
}
```

# makeargv, second half

```
} else {          /* count the number of tokens in snw */
    strcpy(t, snw);
    if (strtok(t, delimiters) == NULL)
        numtokens = 0;
    else
        for (numtokens = 1; strtok(NULL, delimiters) != NULL;
            numtokens++);
        /* create an argument array to contain ptrs to tokens */
    if ((*argvp = calloc(numtokens + 1, sizeof(char *))) == NULL) {
        free(t);
        numtokens = -1;
    } else {      /* insert pointers to tokens into the array */
        if (numtokens > 0) {
            strcpy(t, snw);
            **argvp = strtok(t, delimiters);
            for (i = 1; i < numtokens + 1; i++)
                *((*argvp) + i) = strtok(NULL, delimiters);
        } else {
            **argvp = NULL;
            free(t); } } }
return numtokens
```

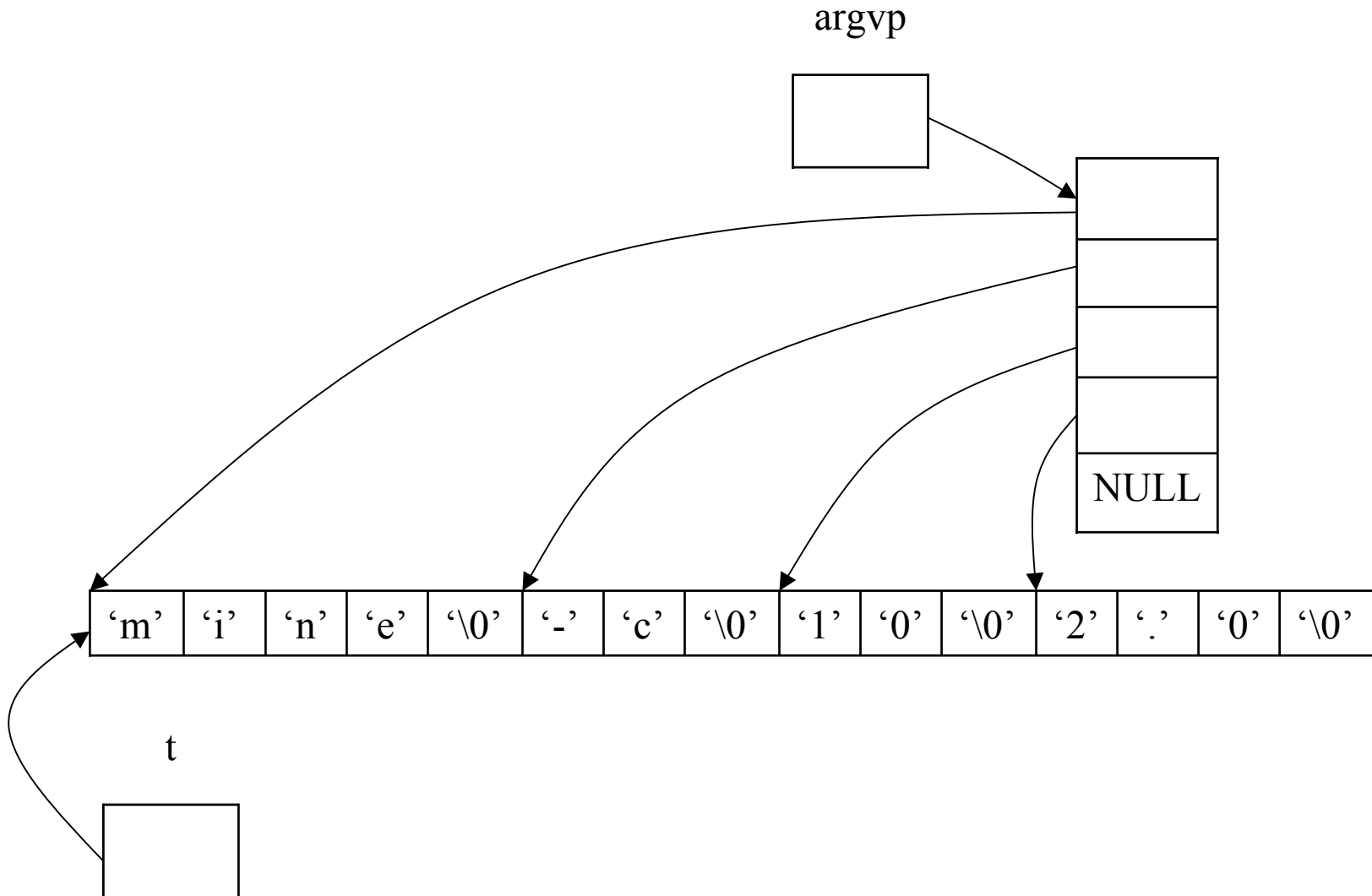
# argtest

```
/* Program 1.1 */
#include <stdio.h>
#include <stdlib.h>
int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv[])
{
    char **myargv;
    char delim[] = " \t";
    int i;
    int numtokens;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s string\n", argv[0]);
        exit(1);
    }
    if ((numtokens = makeargv(argv[1], delim, &myargv)) < 0) {
        fprintf(stderr,
            "Could not construct argument array for %s\n", argv[1]);
        exit(1);
    } else {
        printf("The argument array contains:\n");
        for (i = 0; i < numtokens; i++)
            printf("[%d]:%s\n", i, myargv[i]);
    }
    exit(0);
}
```

# makeargv data structures



# strtok

strtok is not thread safe:

```
strtok(NULL, delimiters);
```

strtok\_r IS thread safe. It has a user-provided parameter that stores the position of the next call to strtok\_r.

# Thread Safe Functions

A function is “thread safe” if it can be safely invoked by multiple threads

# Show History

- listlib.h
- listlib.c
- keeplog.c
- keeploglib.c

# Async-Signal Safe Functions

A function is async-signal safe if that function can be called without restriction from a signal handler

# read

- Not reentrant because error information is returned in external variable `errno`.
- If `read` returns `-1`, `errno` is set to the appropriate error.

# Solutions to read Problem

- Have the read function return the error value as a function value
- Have the read function return the error value as a parameter.
- Implement errno as local to each thread (instead of as a global variable).
- Implement errno as a macro that invokes a function to get errno for the currently running thread.
- Change read so that it invokes a signal on error.

# Process Termination

Normal or abnormal.

- Cancel pending timers and signals
- Release virtual memory resources
- Release other process-held system resources such as locks
- Close open files

# Zombies

- If a parent process is not waiting for a child when it finishes, the child cannot be terminated by its parent. We call these child processes zombies.
- Orphaned child processes become zombies when they terminate.
- System init process (process whose ID is 1) gets rid of orphaned zombies.

# Normal Termination

- Return from main.
- Call to C function `exit` or `atexit`
- Call to `_exit` or `_Exit` system call.

(note that C function `exit` calls user-defined exit handlers that usually provides extra cleanup before calling on `_exit` or `_Exit`).

# exit and \_exit

- Take an integer parameter *status* that indicates the status of the program.
- 0 normal termination.
- Programmer defined non-zero indicates error.
- At exit C function installs user-defined exit handler. Last-installed, first executed.

# exit, \_Exit, \_exit Synopsis

## SYNOPSIS

```
#include <stdlib.h>
```

```
void exit (int status);
```

```
void _Exit (int status);
```

ISO C

## SYNOPSIS

```
#include <unistd.h>
```

```
void _exit (int status);
```

POSIX

# atexit Synopsis

## SYNOPSIS

```
#include <stdlib.h.
```

```
int atexit (void (*func)(void));
```

ISO C

atexit installs a user-defined exit handler. If successful, atexit returns 0 and executes the handler function. If unsuccessful, atexit returns a non-zero value

# Abnormal Termination

- Call abort.
- Process a signal that causes termination.