

# CS241 Systems Programming Programs

Klara Nahrstedt

Lecture 2

1/23/06

---

# Overview

- What is a process?
- Programs : Importance to write correct programs
  - Layout of a Program Image
  - Programming Issues to Pay Attention to
- Summary

# Administrative

- This week
  - Read Chapter 2.1 (Tanenbaum)
  - Read Chapter 1.6, 2 and 3 (Robin&Robin)
  - Machine Problem 0 will be posted (January 25)
  - Deadline for MP0 is January 30
  - Read Newsgroup and Class Website

# Comment: Check out also R&R Appendix A

- UNIX Man Pages
- Compilation
- Header Files
- Linking and libraries
- Macros and conditional compilation
- Makefiles
- Debugging Aids – lint, debugger, truss
- Identifiers, Storage Classes and Linkage Classes

# OS: A Friendly Deception

- OS
  - Manages resources
  - Involves asynchronous and sometimes parallel activities
- Drawback? Maybe lots of extra work
- Benefit: makes life easier for user

# Users, Programs, Processes

- Users have accounts on the system
- Users launch programs
  - Many users may launch same program
  - One user may launch many instances of the same program
- Processes:
  - an executing program

# Analogy

- Program: steps for attending the lecture
  - Step1: walk to Siebel Center Building
  - Step2: enter 1404 Lecture Room
  - Step3: find a seat
  - Step4: listen and take notes
- Process: attending the lecture
  - Action
  - You are all in the middle of a process

# Processes

- A **process** is an abstraction for sequence of operations that implement a computation/program. A process may be manipulated, suspended, scheduled and terminated
- Process is a unit of work in a modern computer
- Process Types:
  - OS processes executing system code program
  - User processes executing user code program
- Processes are executed concurrently with CPU multiplexing among them

# Executing Programs (R&R 21-48)

- Importance of writing correct programs
  - Example: Buffer Overflow and Password Checking
- Program Layout
- Library Function Calls
- Function Return Values and Errors
- Argument Arrays
- Use of Static Variables and Functions

# Importance of Writing Correct Programs

## Buffer Overflow

```
char buf[80];
```

```
printf("Enter your first name");  
scanf("%s", buf);
```

**Problem:** if the user enters more than 79 bytes, the resulting string and the string terminator `\0` do not fit in the allocated variable!!

### Possible Fix of the Problem:

```
char buf[80];
```

```
printf("Enter your first name");  
scanf("%79s", buf);
```

.

# Function 'checkpass' susceptible to buffer overflow

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

int checkpass(void){
    int x;
    char a[9];
    x = 0;
    fprintf(stderr, "a at %p and\nx at %p\n", (void *)a, (void *)&x);
    printf("Enter a short word: ");
    scanf("%s", a);
    if (strcmp(a, "mypass") == 0)
        x = 1;
    return x;
}
```

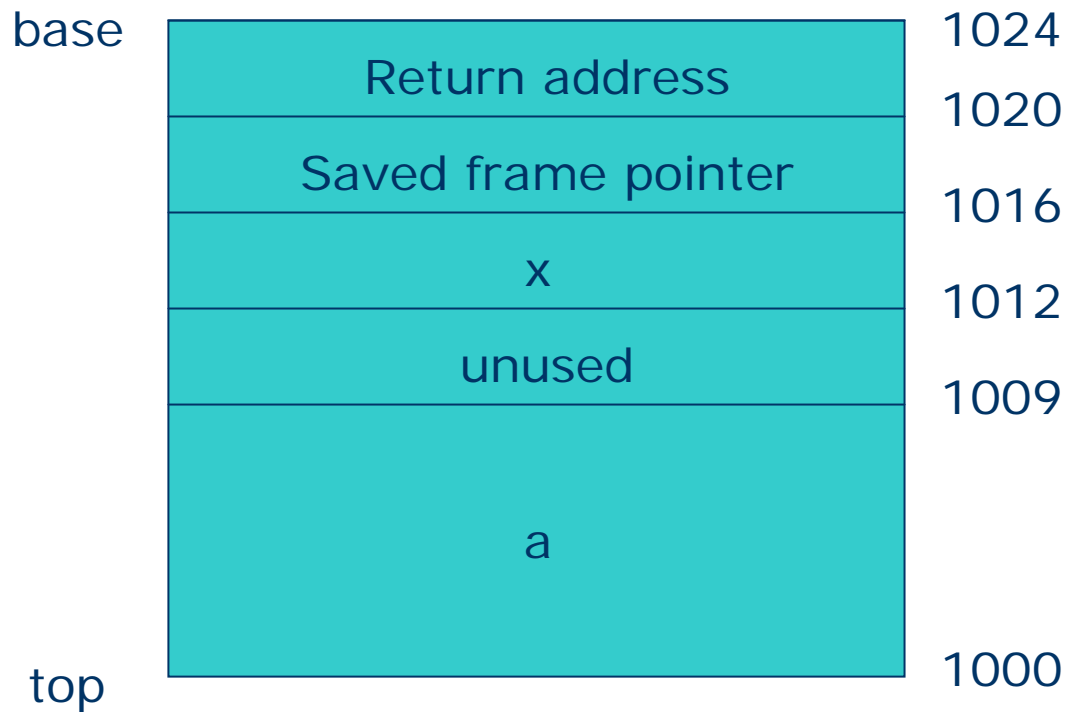
# Password check susceptible to buffer overflow

```
#include <stdio.h>
int checkpass(void);

int main(void) {
    int x;
    x = checkpass();
    fprintf(stderr, "x = %d\n", x);
    if (x)
        fprintf(stderr, "Password is correct!\n");
    else
        fprintf(stderr, "Password is not correct!\n");
    return 0;
}
```

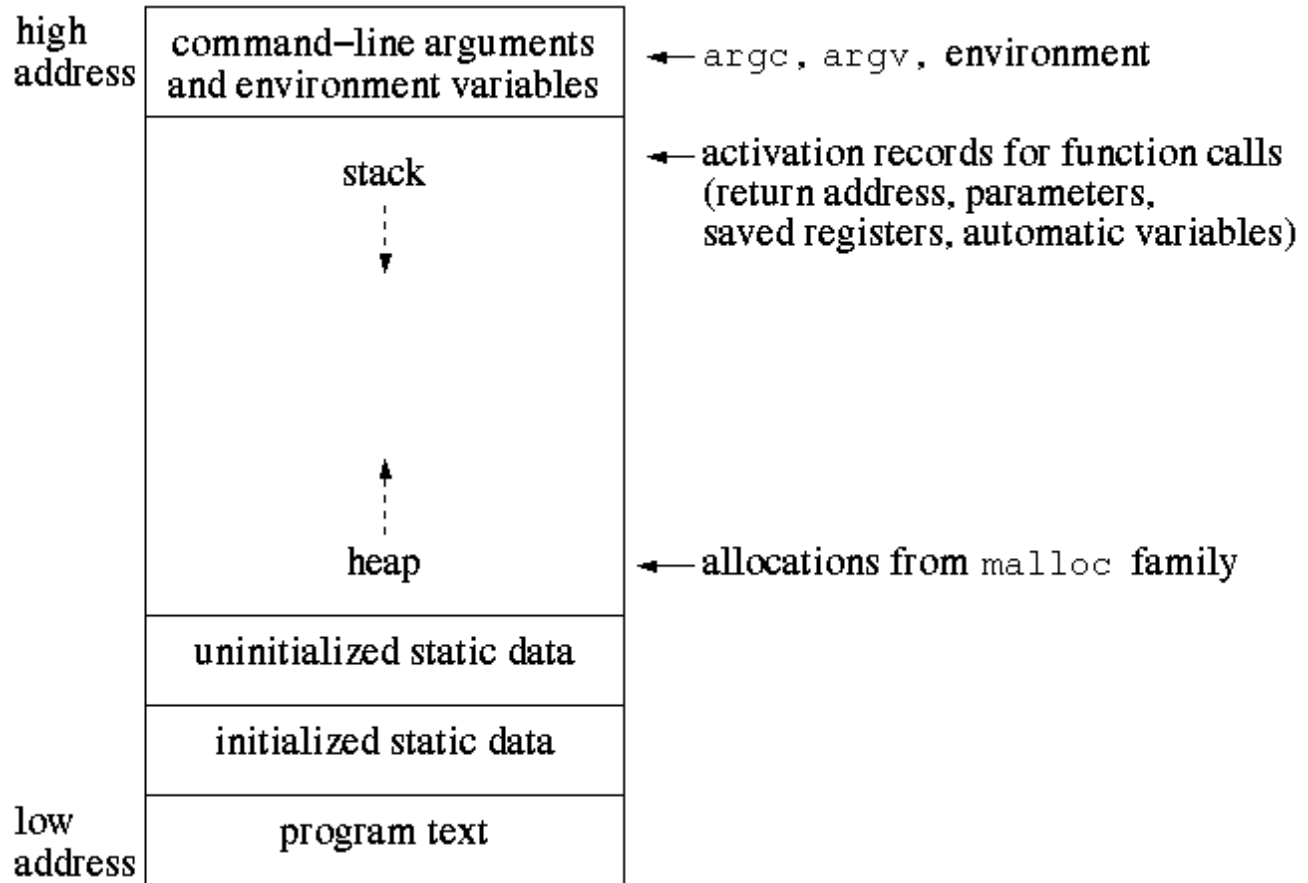
# Stack Layout for 'checkpass'

32-bit address space, i.e., integers and pointers are 4 bytes



**Buffer Overflow of 'a' can cause change of 'x' value  
DANGEROUS!!!**

# Program Layout



# Library Function Calls

- Traditional UNIX – returns 0 successful, -1 unsuccessful, sets errno  
POSIX Standard Committee decides that **no 'errno' be used**. Instead:
  - all new functions **return error code**
- Good coders handle ALL errors, not just mandatory (in the standard) ones.

# Error reporting

- *perror* function outputs to standard error a message corresponding to the current value of *errno*
  - No return values of errors are defined by *perror*

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

```
Void perror(const char * s);
```

-----

- *strerror* function returns a pointer to the system error message corresponding to the error code *errnum*
  - If successful, *strerror* returns a pointer to the error string

```
#include <string.h>
```

```
Char *strerror(int errnum);
```

# Many Library Calls abort if process is interrupted by signal

```
int error;  
int fildes;  
while (((error = close(fildes)) == -1) && (errno  
    == EINTR));  
if (error == -1)  
perror("Failed to close the file");
```

# Handling Errors

- Always handle all errors
- Either:
  - Print error message and exit program (only in main)
  - Return -1 or NULL and set an error indicator such as *errno*
  - Return an error code
- All functions should report errors to calling program
- Use conditional compilation to enclose debugging print statements
  - *cc -DDEBUG*

# Argument Arrays

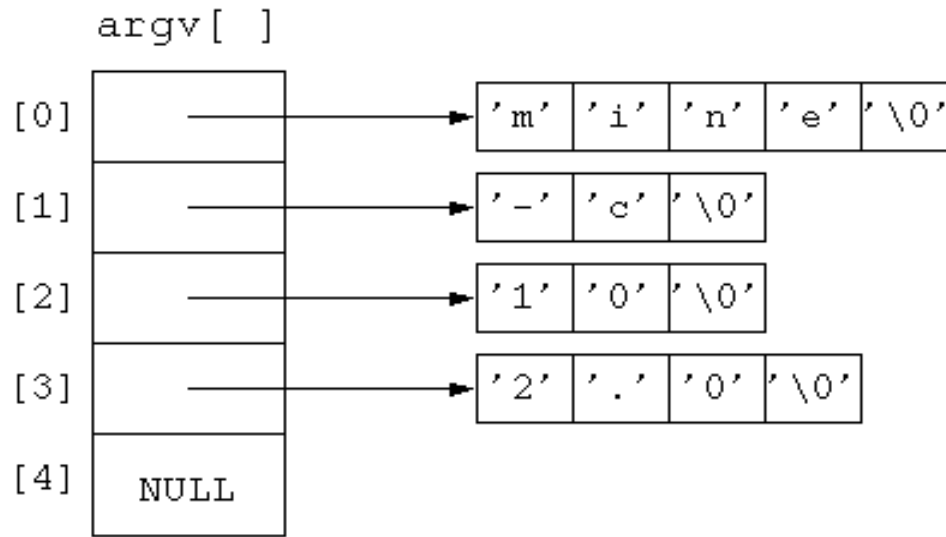
- Command line consists of tokens – arguments
  - `ls -l` (2 token)
  - `mine -c 10 2.0` (4 tokens)

*int main(int argc, char \* argv[])*

- *argc* is number of arguments,
- *argv* is an array of pointers to the tokens

# Argument Arrays

- An argument array is an array of pointers terminated by a NULL pointer.  
Each element of the array is of type `char *` and represents a string.



# Creating Argument Arrays

- Sometimes it is necessary to create a structure like this yourself from a string.  
The shell must do this when you execute a command.
- *argv[]* is an array of pointers to chars  
In C, this is the same as a pointer to a pointer to a char.
- One way to write a function to do this is:  
*char \*\*makeargv(char \*s)*
- If you want to return the number of tokens, you can pass a pointer to the *argv* array as in:  
*int makeargv(char \*s, char \*\*\*argvp)*
- The version we will use has an additional parameter that specifies a string of delimiters:  
*int makeargv(const char \*s, const char \*delimiters, char \*\*\*argvp)*  
The *const* for the first two parameters indicates that the strings should not be modified by the function.

# Identifiers, Storage and Linkage Classes (R&R, p812-814)

- Use static variables carefully, but they are also very useful
  - Example: static variable can hold internal state information between calls to a function
- Be clear about the keyword '*static*' since it is used in C programming in two different ways
  - If *static* is applied to a function, that function can be only called from the file in which it is defined
  - If *static* is applied to a variable then consider:
    - Variable definition outside of any block – i.e., variable exists for the duration of the program
    - Variable definition inside of a block, i.e., variable can only be accessed within the block

# Identifiers

- “**Identifier**” denotes an object: a function, a tag, member of a structure, etc. We mainly consider identifiers associated with functions and variables
- **Scope** is a region of program text where identifier is used
- Identifiers declared more than once may refer to the same object because of **linkage**
  - Each identifier has a **linkage class** of external, internal or none.
- An identifier representing an object has a linkage class related to **storage class** (storage duration).

# Linkage classes

- **Linkage class** determines whether variables can be accessed in files other than the one in which they are declared.
- **Internal linkage class** means they can only be accessed in the file in which they are declared.
- **External linkage class** means they can be accessed in other files.
- Variables declared outside any function and function name identifiers have external linkage by default.  
They can be given internal linkage with the key word *static*.
- Variables declared inside a function are only known inside that function and are said to **have no linkage**.

# Storage classes: static and automatic

- **Static storage class** refers to variables that, once allocated, persist throughout the execution of a program.
- **Automatic storage class** refers to variables which come into existence when the block in which they are declared is entered and are discarded when the defining block is exited.
- Variables declared inside a function have automatic storage class unless they are declared to be static. These are usually allocated on the program stack.
- Variables defined outside any functions have static storage class.
- The word *static* has two meanings in C. One is related to storage class and the other to linkage class.

# Variables

Where Declared	static Modifies	static Applied	Storage Class	Linkage Class
inside a function	storage class	yes	static	none
inside a function	storage class	no	automatic	none
outside any function	linkage class	yes	static	internal
outside any function	linkage class	no	static	external

# Functions

static Modifies	static Applied?	Linkage Class
linkage class	yes	internal
linkage class	no	external

# Summary

- Process Concept – executable program
- Correct Programs – needed
- Careful Consideration of Errors and Arguments when starting C Programming
  
- Read R&R : pp: 16-48, pp. 812-814
- Read T: Chapter 2.1