Topics: Programs, command-line, storage class and Processes (SGG 3.1-3.2; USP 2, 3)

CS 3733 Operating Systems

Instructor: Dr. Turgay Korkmaz

Department Computer Science

The University of Texas at San Antonio

Office:	NPB 3.330
Phone:	(210) 458-7346
Fax:	(210) 458-4437
e-mail:	korkmaz@cs.utsa.edu
web:	www.cs.utsa.edu/~korkmaz

These slides are prepared based on the materials provided by Drs. Robbins, Zhu, and Liu. Thanks to all.

Objectives

- Introduce process concept -- **program in execution**, which forms the basis of all computation
  - Understand: Program vs. Process vs. Threads
- Learn Various Aspects of Processes: Creation, State Transitions and Termination
- Learn the presentation of processes in OS: PCB (process control block)
- Understand Memory Layout of Program Image
  - Understand Argument Array and Function Safety
  - Understand Storage and Linkage Classes
- Learn process usage in Unix

Outline

- Programs, Processes and Threads
- Process creation and its components
- States of a process and transitions
- PCB: Process Control Block
- Process (program image) in memory
  - Pointers
  - Argument Arrays
  - Making Functions Safe
  - Storage and Linkage Classes

(SGG 3.1, 3.2, USP 2)

- Process Creation in UNIX (USP 3)

Programs vs. Processes

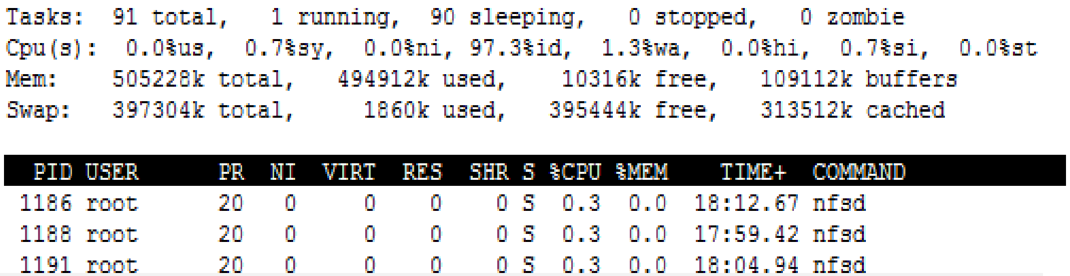
- Program: a sequence of instructions/functions
  - To accomplish a defined task
  - **Passive** entity, stored as files on disk

- Process: a **program in execution**
  - Unit of work in a system
  - **Active/Dynamic** concept: running of a program
- How does a program becomes a process?
  - Run/execute a program? But how?

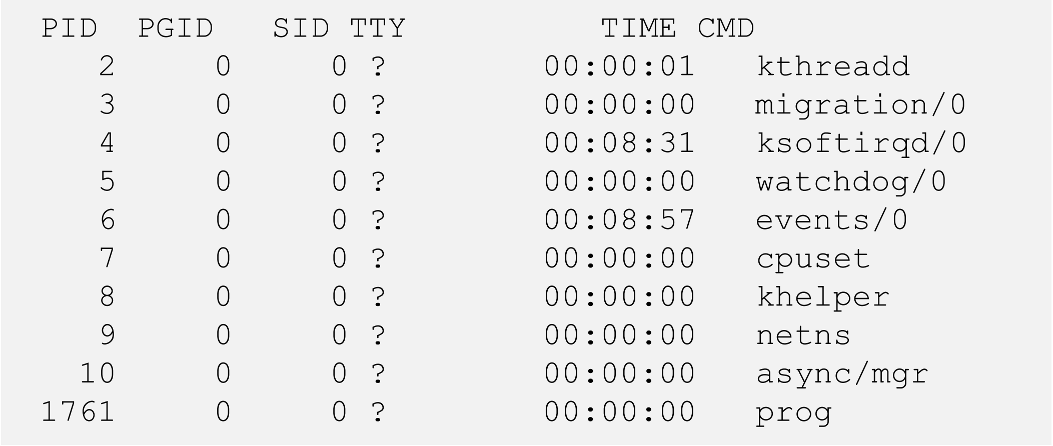
How does a program becomes a process?

```
> vi prog.c
> gcc prog.c -o prog
>./prog &
> top
> ps -ejH
```

← what is happening here?



```
Tasks:  91 total,   1 running,  90 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.7%sy,  0.0%ni, 97.3%id,  1.3%wa,  0.0%hi,  0.7%si,  0.0%st
Mem:    505228k total,  494912k used,   10316k free,  109112k buffers
Swap:   397304k total,     1860k used,  395444k free,  313512k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1186 root      20   0     0    0    0 S  0.3  0.0  18:12.67 nfsd
 1188 root      20   0     0    0    0 S  0.3  0.0  17:59.42 nfsd
 1191 root      20   0     0    0    0 S  0.3  0.0  18:04.94 nfsd
```

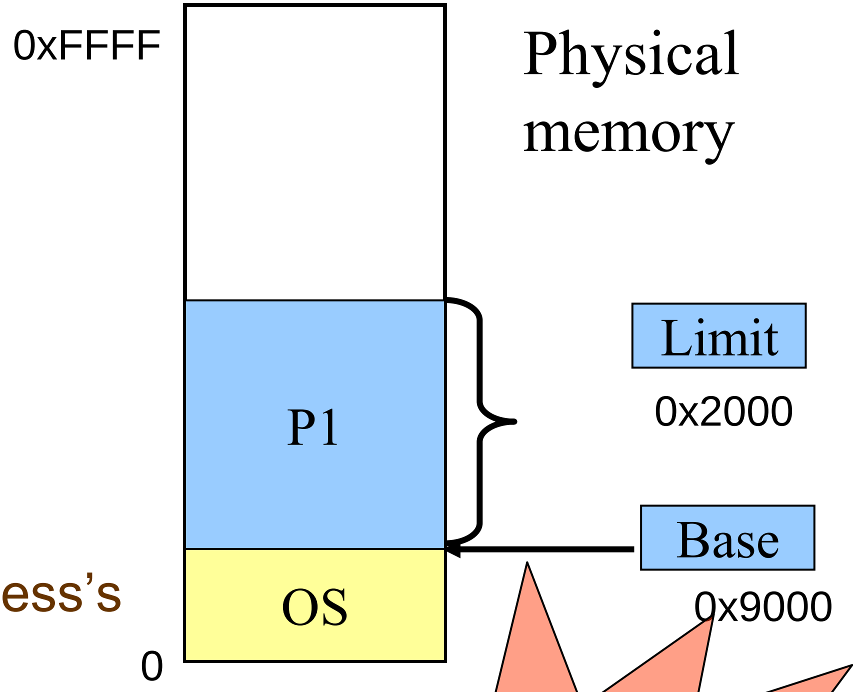


```
  PID  PGID   SID TTY          TIME CMD
    2     0     0 ?        00:00:01   kthreadd
    3     0     0 ?        00:00:00   migration/0
    4     0     0 ?        00:08:31   ksoftirqd/0
    5     0     0 ?        00:00:00   watchdog/0
    6     0     0 ?        00:08:57   events/0
    7     0     0 ?        00:00:00   cpuset
    8     0     0 ?        00:00:00   khelper
    9     0     0 ?        00:00:00   netns
   10     0     0 ?        00:00:00   async/mgr
 1761     0     0 ?        00:00:00   prog
```

Can multiple processes be from a single program?

Load prog to Memory and Set Program Counter (PC)
Process execution progresses in **sequential** fashion

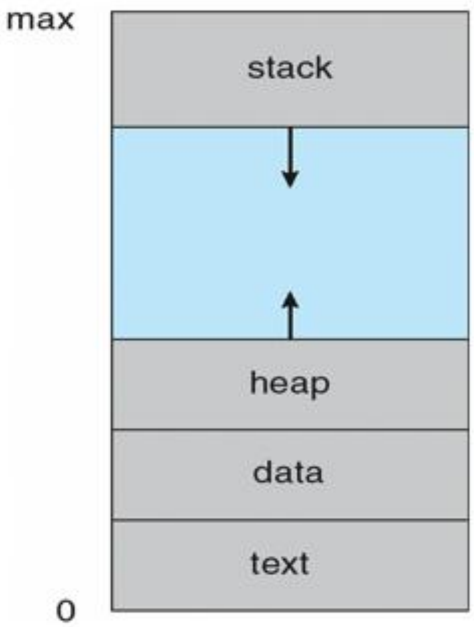
- Special CPU registers
  - **Base register**: start of the process’s memory partition
  - **Limit register**: length of the process’s memory partition
  - Access limited to system mode
- Address *translation*
  - **Logical address**: location from the process’s point of view
  - **Physical address**: location in actual memory
  - Physical = base + logical address
    - ✓ Logical address: 0x1204
      Physical address:0x1204+0x9000 = 0xa204
  - Logical address larger than limit → error



More about Memory management in SGG Ch 8 & 9

Process: Address Space

- A process includes
  - Memory image
    - Program code: **text** segment
    - Global variables: **data** section
    - Temporary data (local variables, function parameters, and return address etc.): **Stack**
    - Dynamically allocated data (malloc): **Heap**
  - CPU image
    - Registers, Program counter (PC),
    - Stack Ptr (SP), Proc Status Word(PSW)
- OS makes CPU switch from one process to another
- Why/How?



Switching from one process to another

Example 2.2 from USP Book

- Suppose
  - process 1 executed statements 245, 246 and 247 in a loop
  - process 2 executes the statements 10, 11, 12, 13, ...
  - CPU starts executing process 1 for 5 instructions; then process 1 loses CPU;
  - CPU then executes 4 instructions of process 2 before losing the CPU;
- the executed sequence of instructions:

  245_1, 246_1, 247_1, 245_1, 246_1, 10_2, 11_2, 12_2, 13_2, 247_1, 245_1, 246_1, ...;

  subscript indicates which process
- Two threads of execution (each from a process)
- To switch from one process to another, OS needs to
  - store the state of the old one to memory and
  - restore the state of the new one from memory

Context - Switching

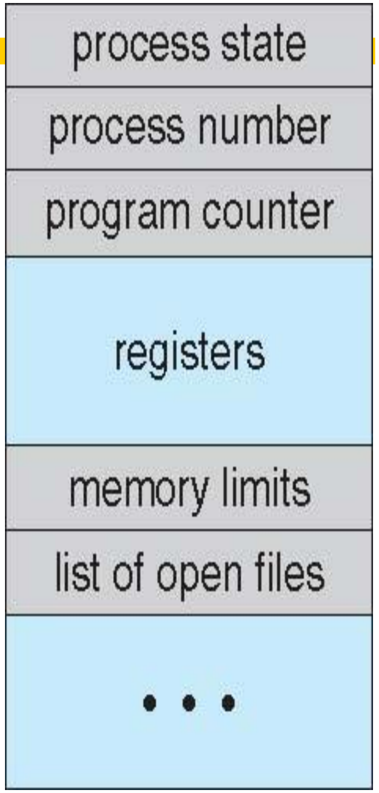
Process Control Block (PCB):

Everything about a process is stored in PCB

CPU image

- **Registers**: in addition to general registers
  - **Program Counter (PC):** contains the memory address of the next instruction to be fetched.
  - **Stack Pointer (SP**): points to the top of the current *stack* in memory. The stack contains one frame for each procedure that has been entered but not yet exited.
  - **Program Status Word (PSW**): contains the condition code bits and various other control bits
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
- Thread synchronization and communication resource: semaphores and sockets

Double linked list to maintain PCBs

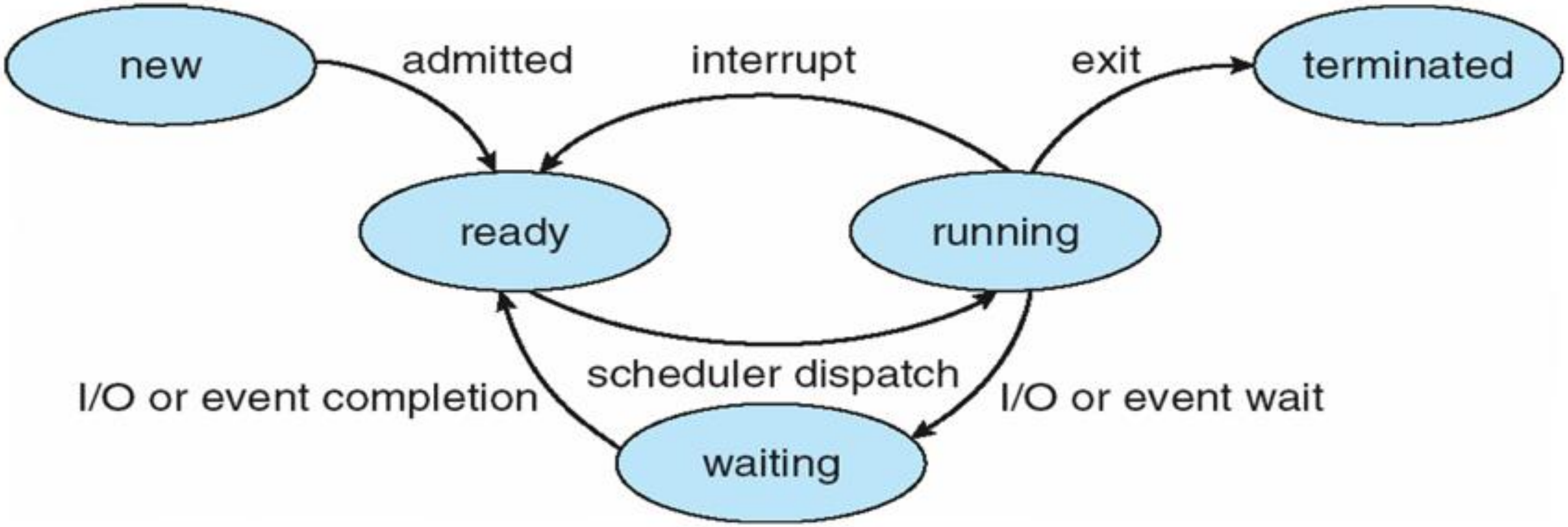

process state
process number
program counter
registers
memory limits
list of open files
• • •

/linux-3.6.5/include/linux/sched.h

```
struct task_struct {
        volatile long state;
        void *stack;
        atomic_t usage;
        unsigned int flags;
        unsigned int ptrace;
        /* … ~1.7K
         360 lines */
}
```

Process States and State Transitions

- During the lifetime of a process, it changes its ***state***
  - **New**: being created and starting up
  - **Running**: instructions being executed
  - **Waiting**: waiting for some event (such as I/O) to occur
  - **Ready**: waiting to be assigned to a processor
  - **Terminated/Halted**: finished execution

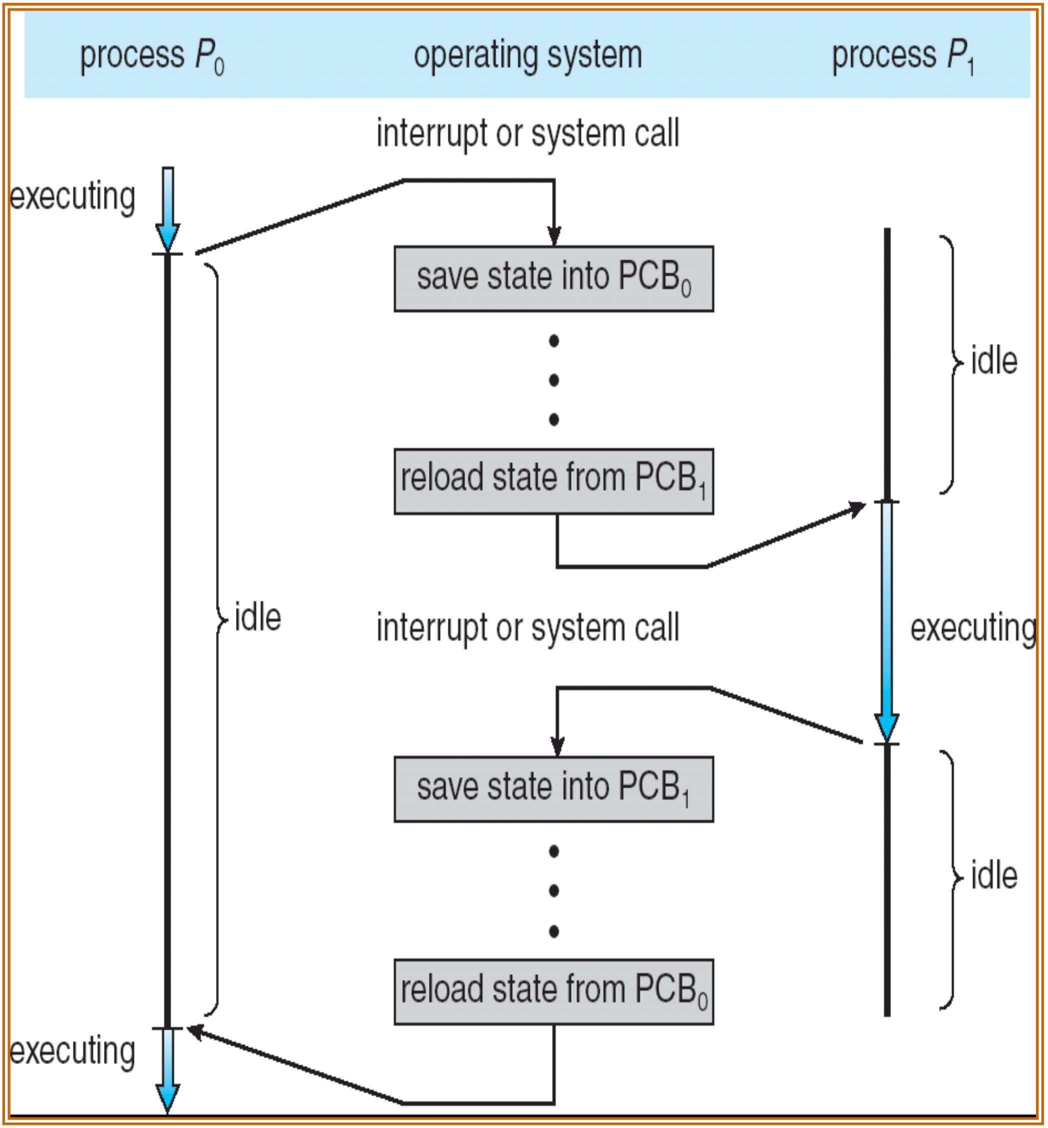


Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead
  - 1 ~ 1000 ms
- Hardware support
  - Multiple set of registers
- Other performance issues/problems
  - Cache content: locality is lost
  - TLB content: may need to flush

Which process to run?

Scheduling!!!



Let’s try to see context switching in action?

```
> vi prog.c
  for(i=0; i<1000; i++) {
     // …
     fprintf(stderr, “%s”, argv[1]);
  }
> prog A &
> prog A &; prog B &; prog C &
```

Process vs. Threads

- A process stars with a single ***flow of control***
- The flow executes a sequence of instructions: ***thread of execution***
- Thread of execution: logically related sequence of instruction address from PC during execution
- **Thread**: represents a thread of execution of a process → basic unit of CPU utilization
  - Thread ID
  - PC
  - Register set
  - Stack

Threads will be discussed later!

Focus on single thread processes!

Outline

- Programs, Processes and Threads
- Process creation and its components
- States of a process and transitions
- PCB: Process Control Block
- Process (program image) in memory
  - Pointers
  - Argument Arrays
  - Making Functions Safe
  - Storage and Linkage Classes

(SGG 3.1, 3.2, USP 2)

- Process Creation in UNIX (USP 3)

Process (program image) in memory

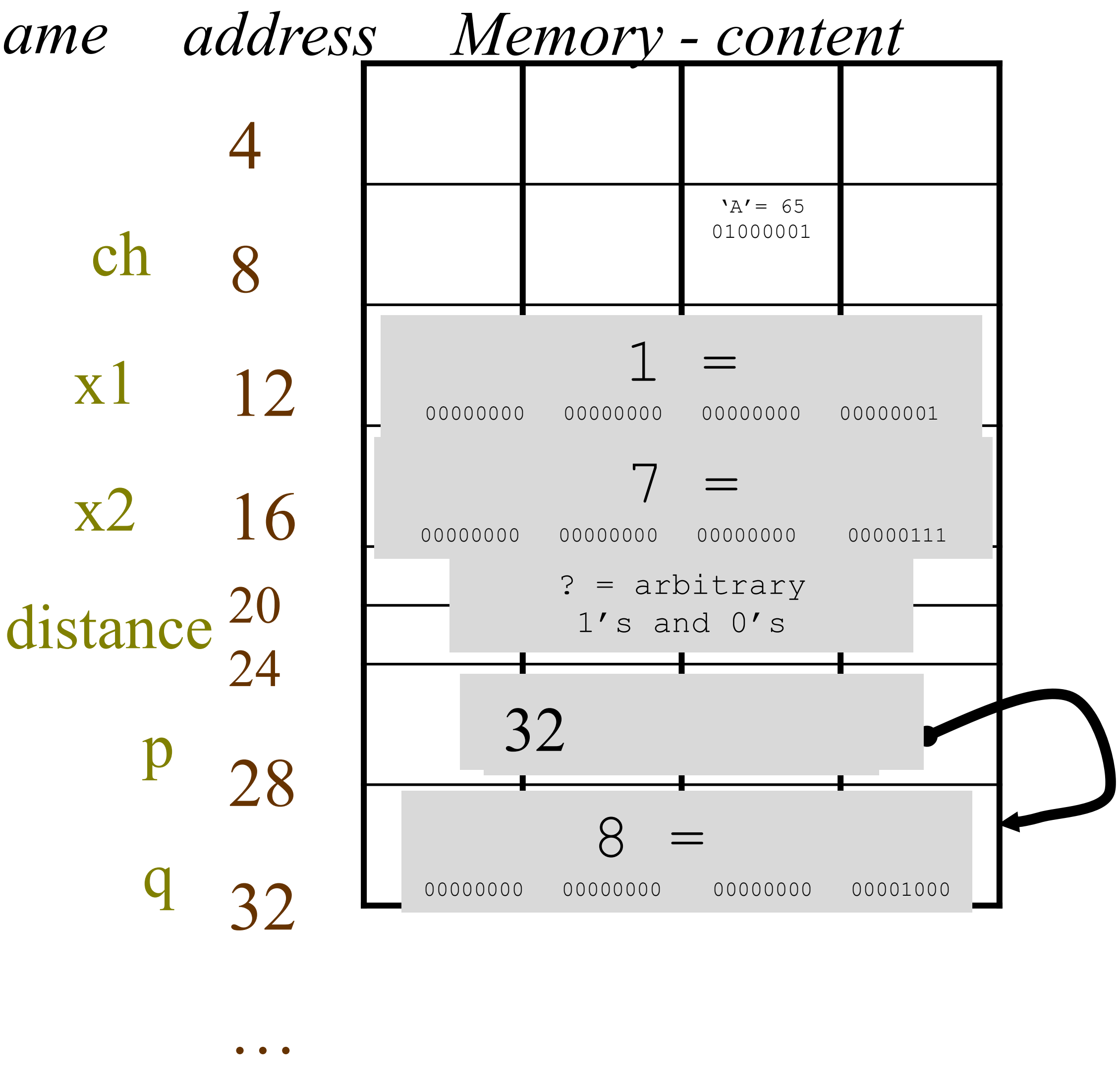
- Pointers
- Command–line arguments
- Making Functions Safe
- Storage and Linkage Classes

Pointers review and quiz 1

- A pointer is a variable that contains the *address* of another variable (memory location).
- Using pointers, we can indirectly access memory and/or update the content at this address.
- Why do we need pointers?
  - To refer to a large data structure in a **compact way**
  - Facilitate **data sharing** between different parts of the program
  - Make it possible to reserve **new memory** during program execution
  - Allow to represent **record relationships** among data items (data structures: link list, tree etc…)

Pointers review: Addresses and Pointers

```
char ch='A';

int x1=1, x2=7;

double distance;

int *p;

int q=8;

p = &q;
```



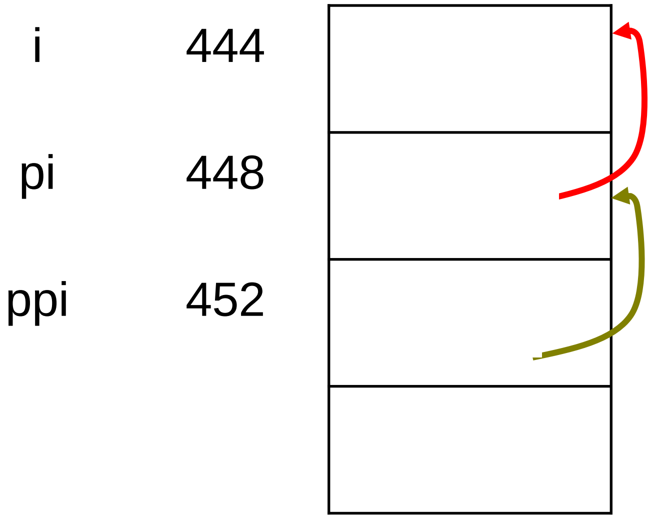
name	*address*	*Memory - content*
	4	
ch	8	'A'= 65 01000001
x1	12	1 = 00000000 00000000 00000000 00000001
x2	16	7 = 00000000 00000000 00000000 00000111
distance	20 24	? = arbitrary 1's and 0's
p	28	32
q	32	8 = 00000000 00000000 00000000 00001000
	…	

How about

```
char *pc;
double *pd;
```

Pointers review: Pointers to Pointers

```
int     i;
int     *pi;
int     **ppi;
i=5;
ppi=&pi;
*ppi = &i;
```



i	444	
pi	448	
ppi	452	

Can we have
int ***p;

What will happen now i=10;
*pi=20;
**ppi = 30;

```
main()
{
  int *pi;
  …
  f(&pi);
}

int f(int **p)
{
 *pp=New(int);
  …
}
```

*

Pointers review: Pointers and Structures

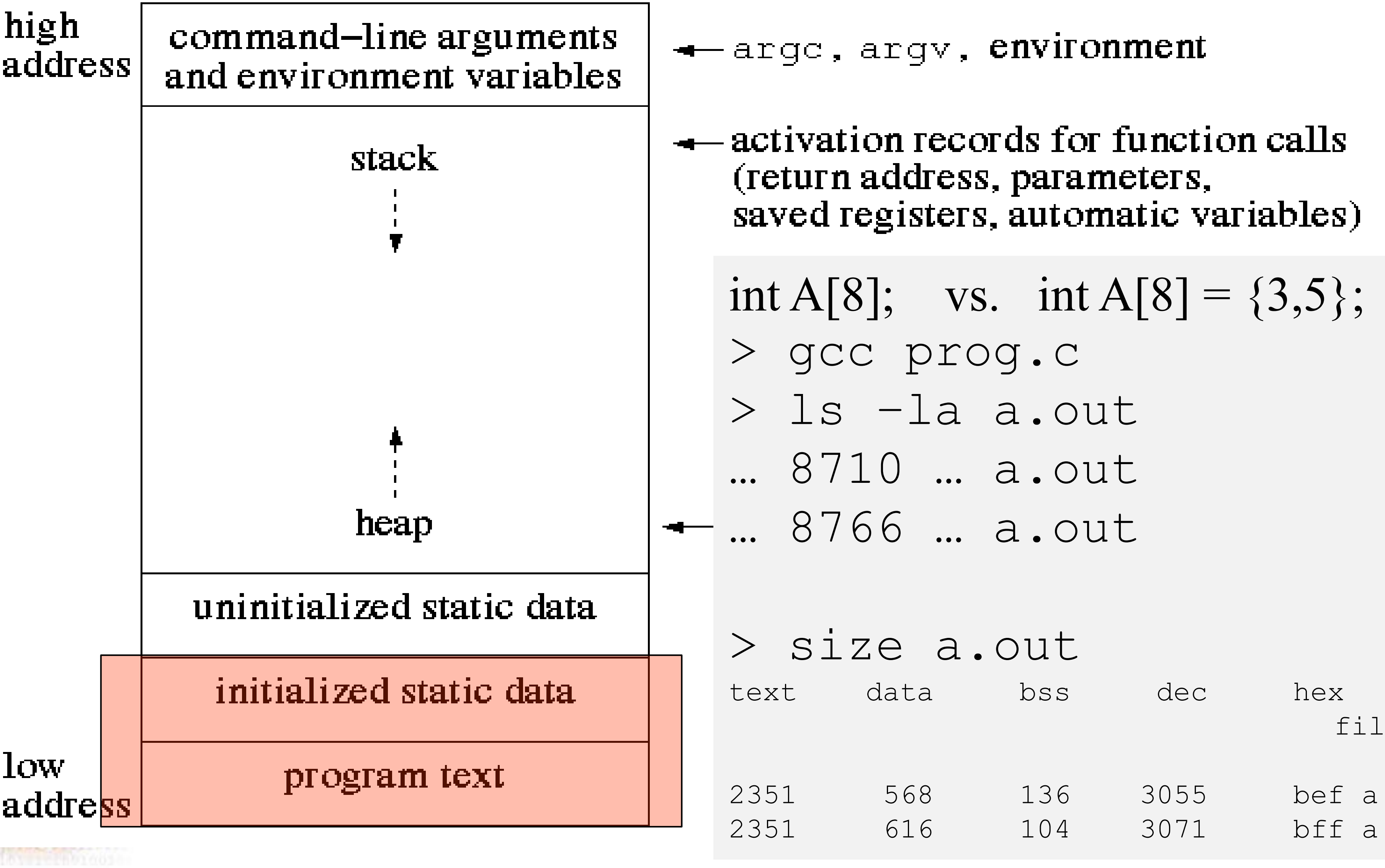
```
typedef struct point {
  int x;
  int y;
} pointT;
typedef struct triangle {
  pointT corner[3];
  char   color;
  char   *name;
  struct triangle *next;
} triangleT;
triangleT p, *q;
```

```
/* fix each statement  */
p->corner[2]->x = 6;
p->name = “Equilateral”;
q.corner->y = 6;
q.next.color = ‘r’;
```

After your fixes, there will be no compiler errors in the above code. But still some statements will cause an error during execution time. Identify and explain how to fix them.

Solve quiz 1

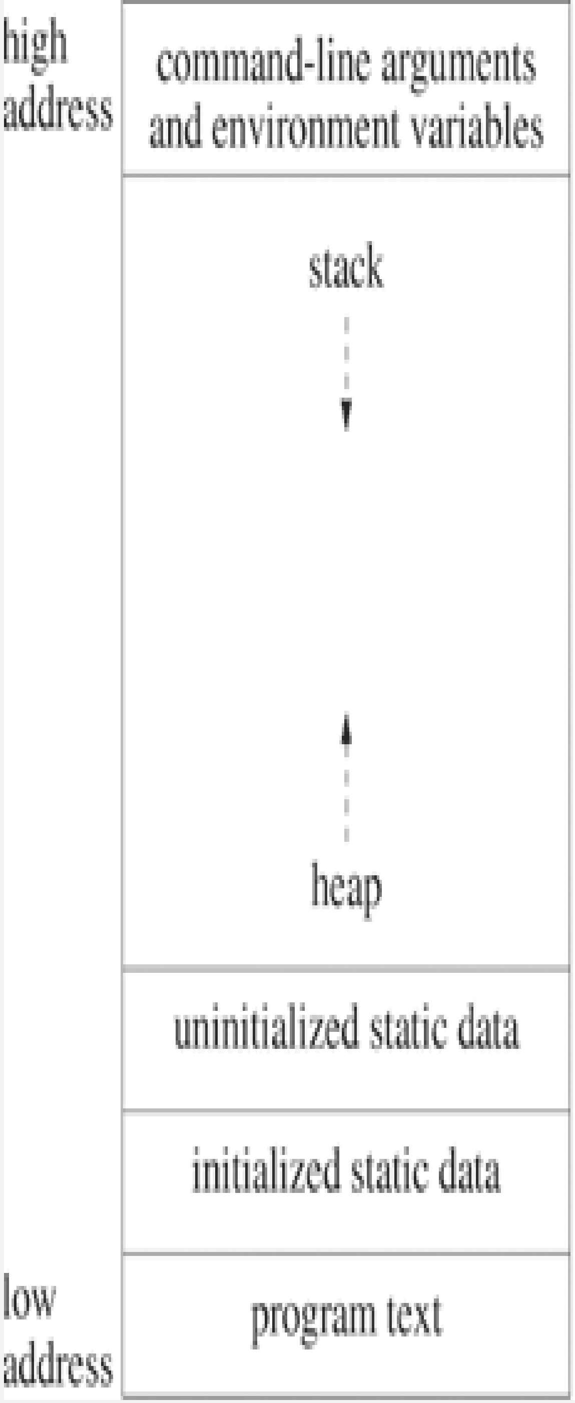
Program Image in Memory (Fig. 2.1 USP)

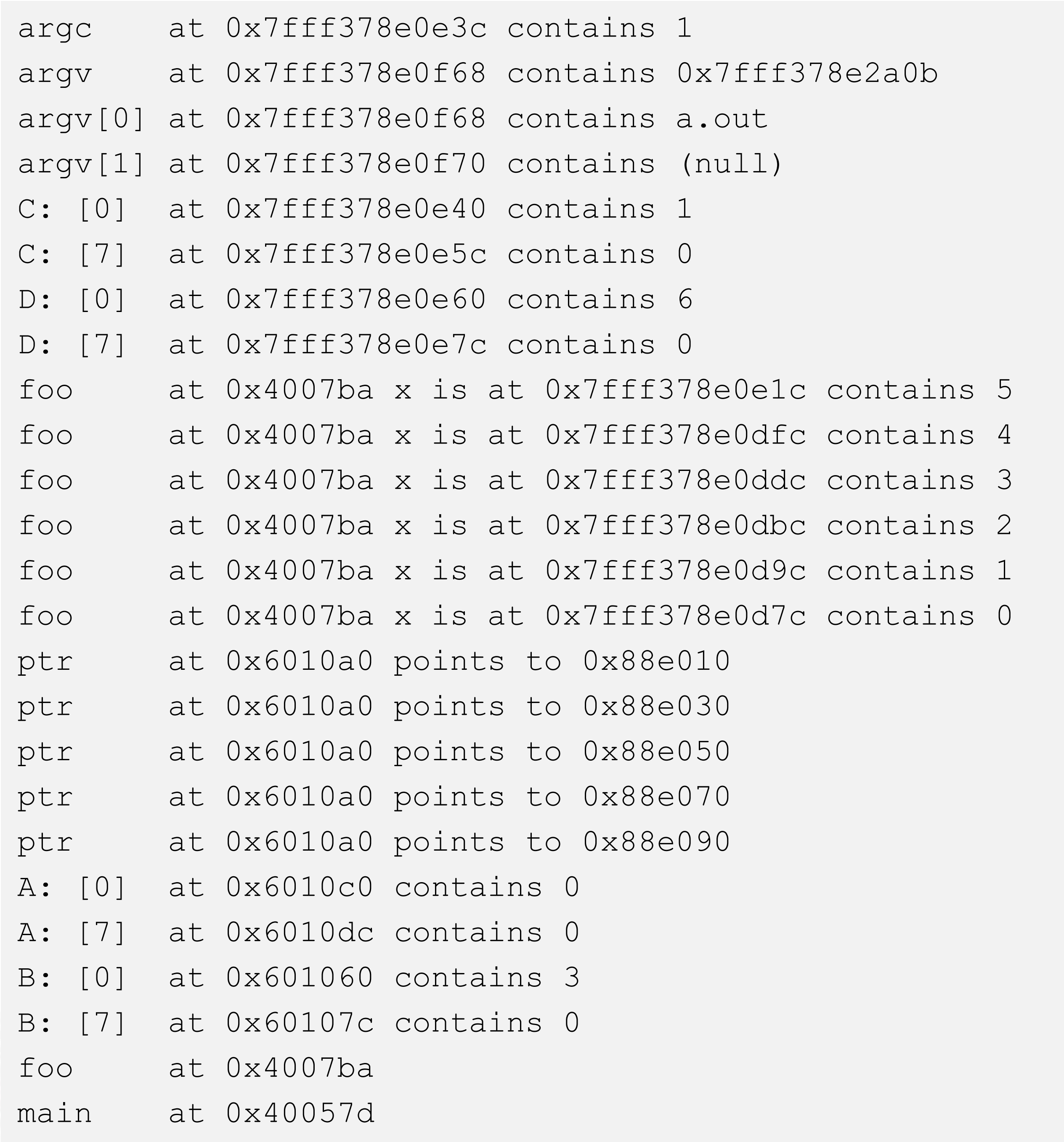

high address
command-line arguments and environment variables
stack
heap
uninitialized static data
initialized static data
program text
low address
argc, argv, environment
activation records for function calls (return address, parameters, saved registers, automatic variables)

int A[8]; vs. int A[8] = {3,5};

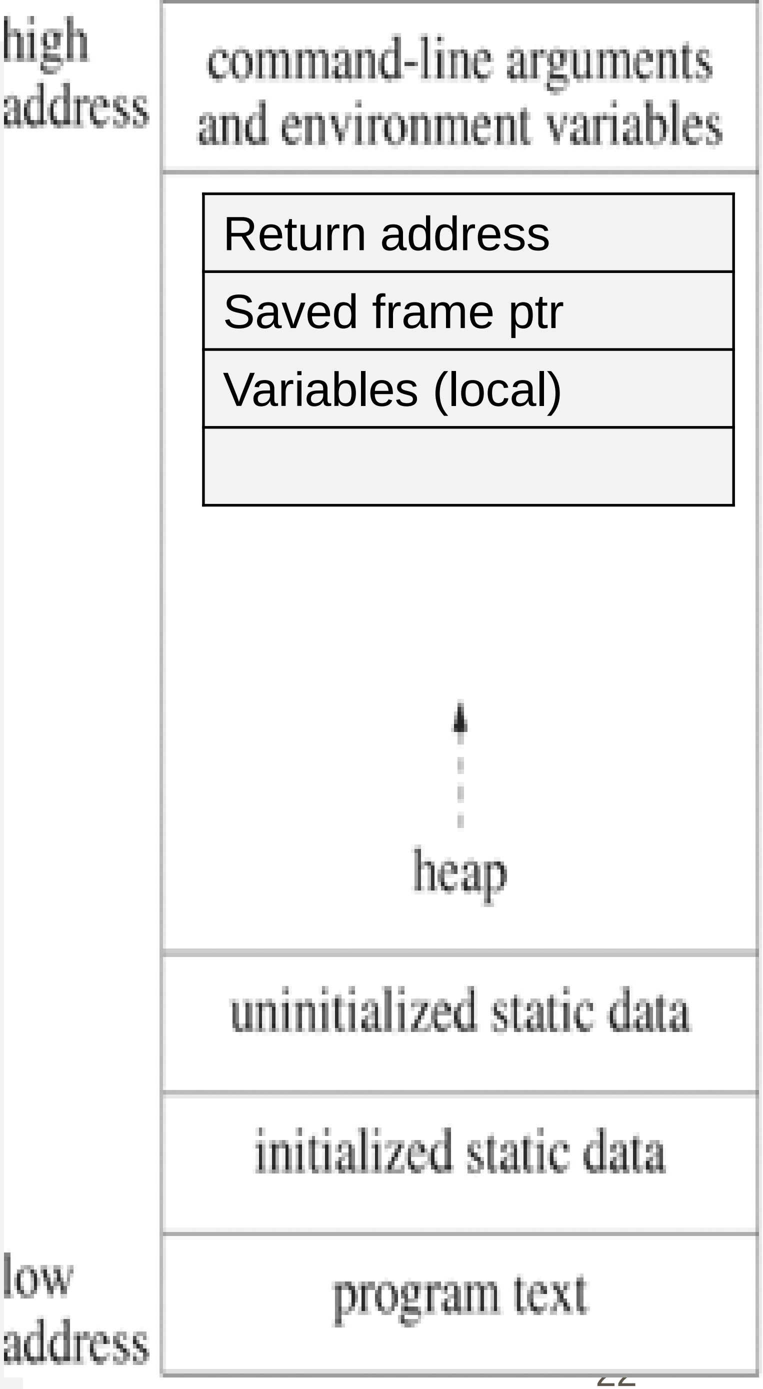
```
> gcc prog.c
> ls -la a.out
… 8710 … a.out
… 8766 … a.out


> size a.out
text    data     bss     dec     hex filename
2351     568     136    3055     bef a.out
2351     616     104    3071     bff a.out
```

```
int A[8];        int B[8]={3};     int i, *ptr;
int main(int argc, char *argv[]) {
   int C[8];  int D[8] ={6};
   printf("argc    at %p contains %d \n", &argc, argc);
   printf("argv    at %p contains %p\n", argv, *argv);
   printf("argv[0] at %p contains %s\n", &argv[0], argv[0]);
   printf("argv[%d] at %p contains %s\n",argc,  &argv[argc] , argv

   printf("C: [0]  at %p contains %d\n", &C[0], C[0]);
   printf("C: [7]  at %p contains %d\n", &C[7], C[7]);
   printf("D: [0]  at %p contains %d\n", &D[0], D[0]);
   printf("D: [7]  at %p contains %d\n", &D[7], D[7]);
   foo(5);
   for(i=0; i<5; i++) {
      ptr = (int *) malloc(sizeof(int));
      printf("ptr     at %p points to %p\n", &ptr, ptr);
   }
   printf("A: [0]  at %p contains %d\n", &A[0], A[0]);
   printf("A: [7]  at %p contains %d\n", &A[7], A[7]);
   printf("B: [0]  at %p contains %d\n", &B[0], B[0]);
   printf("B: [7]  at %p contains %d\n", &B[7], B[7]);
   printf("foo     at %p\n", foo);
   printf("main    at %p\n", main);
   return 0;
}
int foo(int x){
   printf("foo     at %p x is at %p contains %d\n", foo, &x, x);
   if (x > 0) foo(x-1);
   return x;
}
```


high address
command-line arguments and environment variables
stack
heap
uninitialized static data
initialized static data
program text
low address



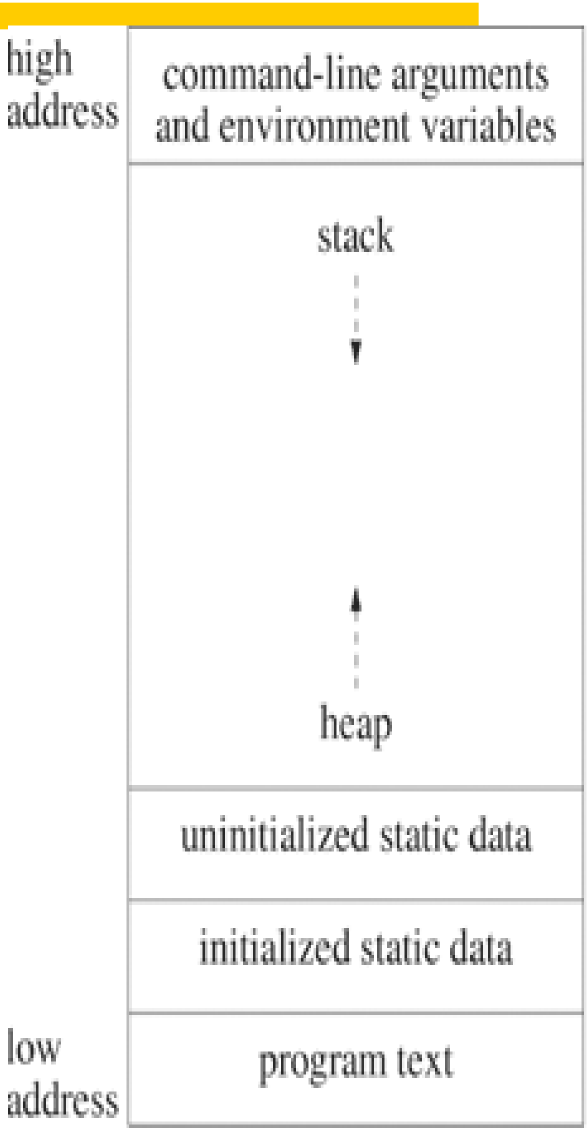
```
argc     at 0x7fff378e0e3c contains 1
argv     at 0x7fff378e0f68 contains 0x7fff378e2a0b
argv[0] at 0x7fff378e0f68 contains a.out
argv[1] at 0x7fff378e0f70 contains (null)
C: [0]   at 0x7fff378e0e40 contains 1
C: [7]   at 0x7fff378e0e5c contains 0
D: [0]   at 0x7fff378e0e60 contains 6
D: [7]   at 0x7fff378e0e7c contains 0
foo      at 0x4007ba x is at 0x7fff378e0e1c contains 5
foo      at 0x4007ba x is at 0x7fff378e0dfc contains 4
foo      at 0x4007ba x is at 0x7fff378e0ddc contains 3
foo      at 0x4007ba x is at 0x7fff378e0dbc contains 2
foo      at 0x4007ba x is at 0x7fff378e0d9c contains 1
foo      at 0x4007ba x is at 0x7fff378e0d7c contains 0
ptr      at 0x6010a0 points to 0x88e010
ptr      at 0x6010a0 points to 0x88e030
ptr      at 0x6010a0 points to 0x88e050
ptr      at 0x6010a0 points to 0x88e070
ptr      at 0x6010a0 points to 0x88e090
A: [0]   at 0x6010c0 contains 0
A: [7]   at 0x6010dc contains 0
B: [0]   at 0x601060 contains 3
B: [7]   at 0x60107c contains 0
foo      at 0x4007ba
main     at 0x40057d
```



Another Example Program

ptr: stack ? heap
x: stack ? heap

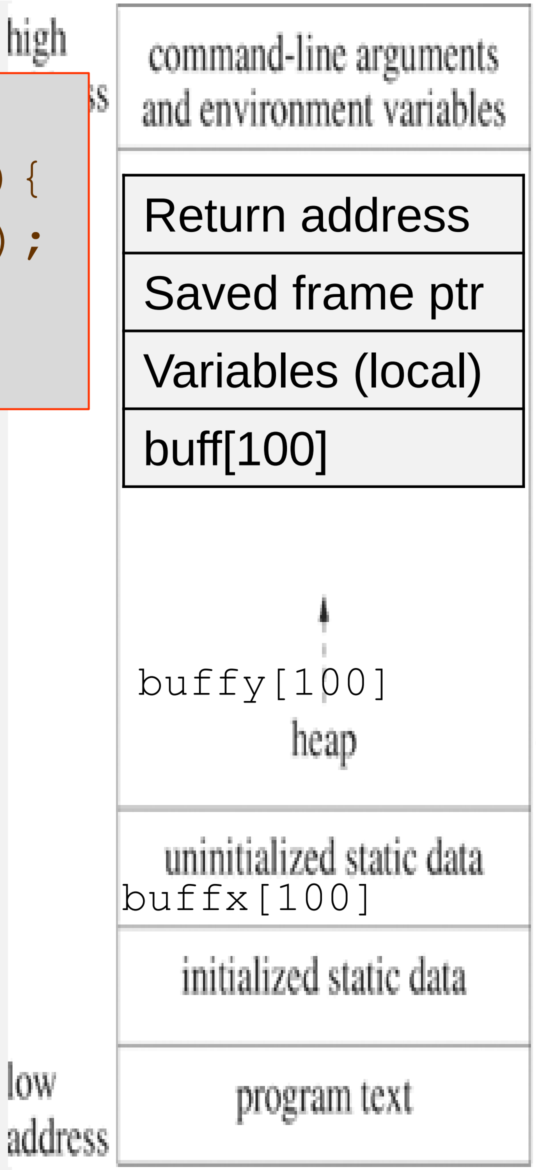
```
int foo(int x){
    return x;
} // foo is popped off the call stack here

int main()
{
    int *ptr = (int *)
               malloc(sizeof(int));

    foo(5); // foo is pushed on the stack

   return 0;
}
```


high address
command-line arguments and environment variables
stack
heap
uninitialized static data
initialized static data
program text
low address

What might go wrong here? How can we fix?

```
char *get_me_a_name(){
   char buff[100];
   scanf(“%s”, buff);
   return buff;
}
char *get_me_a_name(){
   static char buffx[100];
   scanf(“%s”, buffx);
   return buffx;
}
char *get_me_a_name(){
   char *buffy;
   buffy = malloc(100); // if NULL ?
   scanf(“%s”, buffy);
   return buffy;
}
```

```
char buffx[100];
char *get_me_a_name(){
   scanf(“%s”, buffx);
   return buffx;
}
```



high

ss

command-line arguments and environment variables

Return address

Saved frame ptr

Variables (local)

buff[100]

buffy[100]

heap

uninitialized static data

buffx[100]

initialized static data

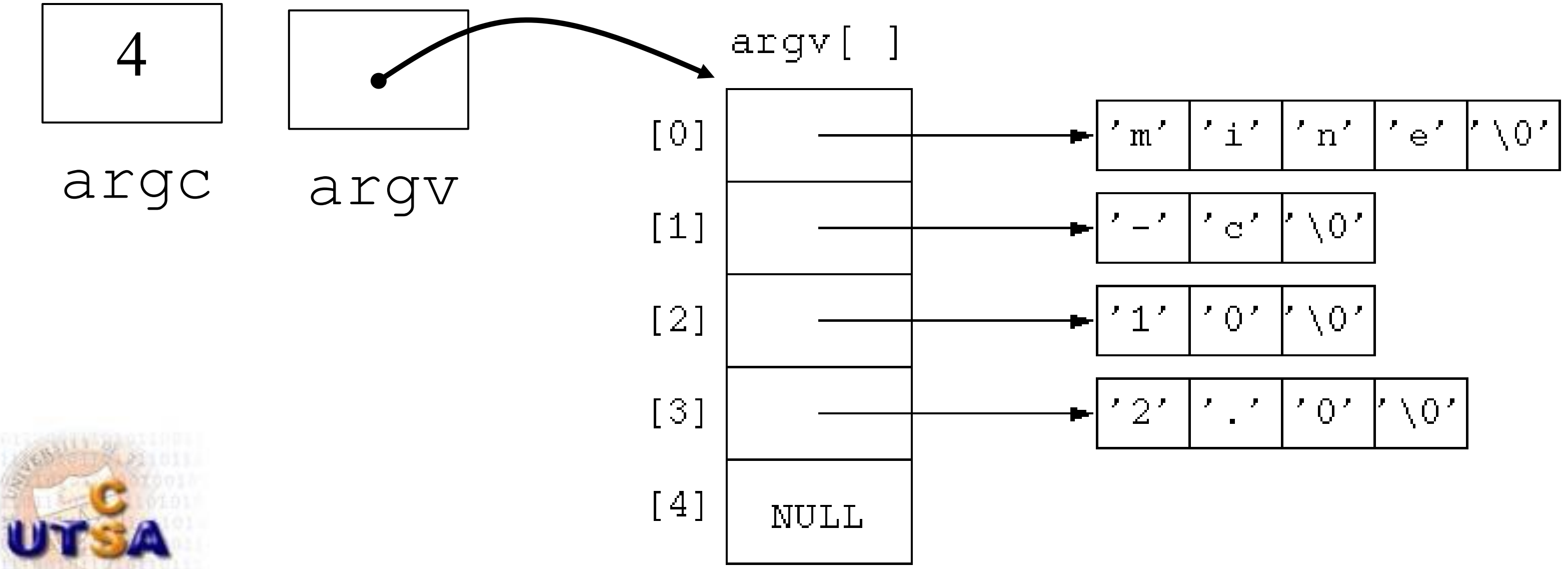
low address

program text

Argument Arrays and quiz 2

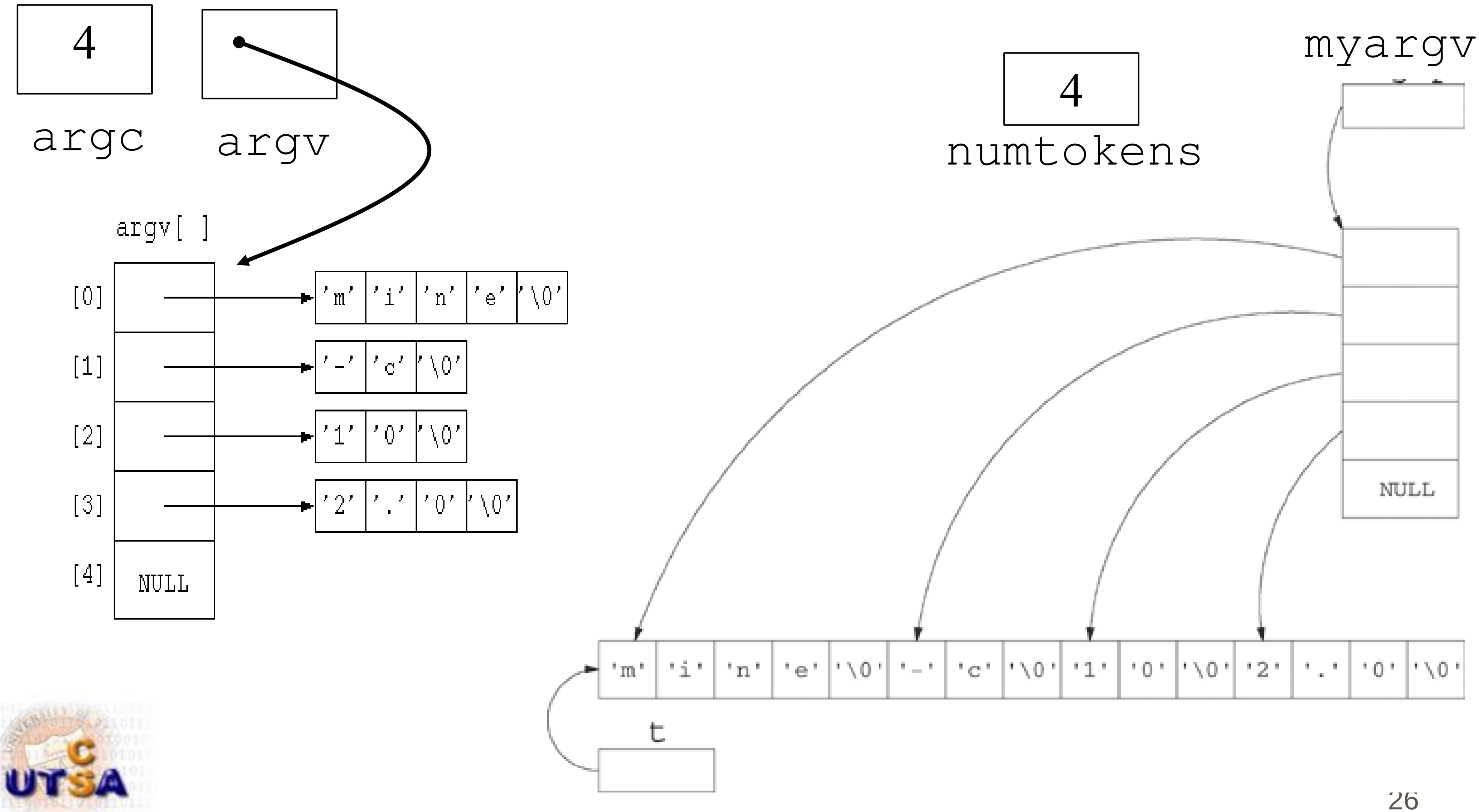
```
> mine -c 10 2.0
```

- What information do you get in your program mine?
- Argument array
  - an array of pointers terminated by a NULL pointer
  - Each element is of type char * and represents a string.



Self-study

Create Your Own Argument Arrays from a string



Self-study

Create Your Own Argument Arrays

- How would you declare **`myargv`** and **`numtokens`**?
- How would you write a function (return type, parameters etc.)?

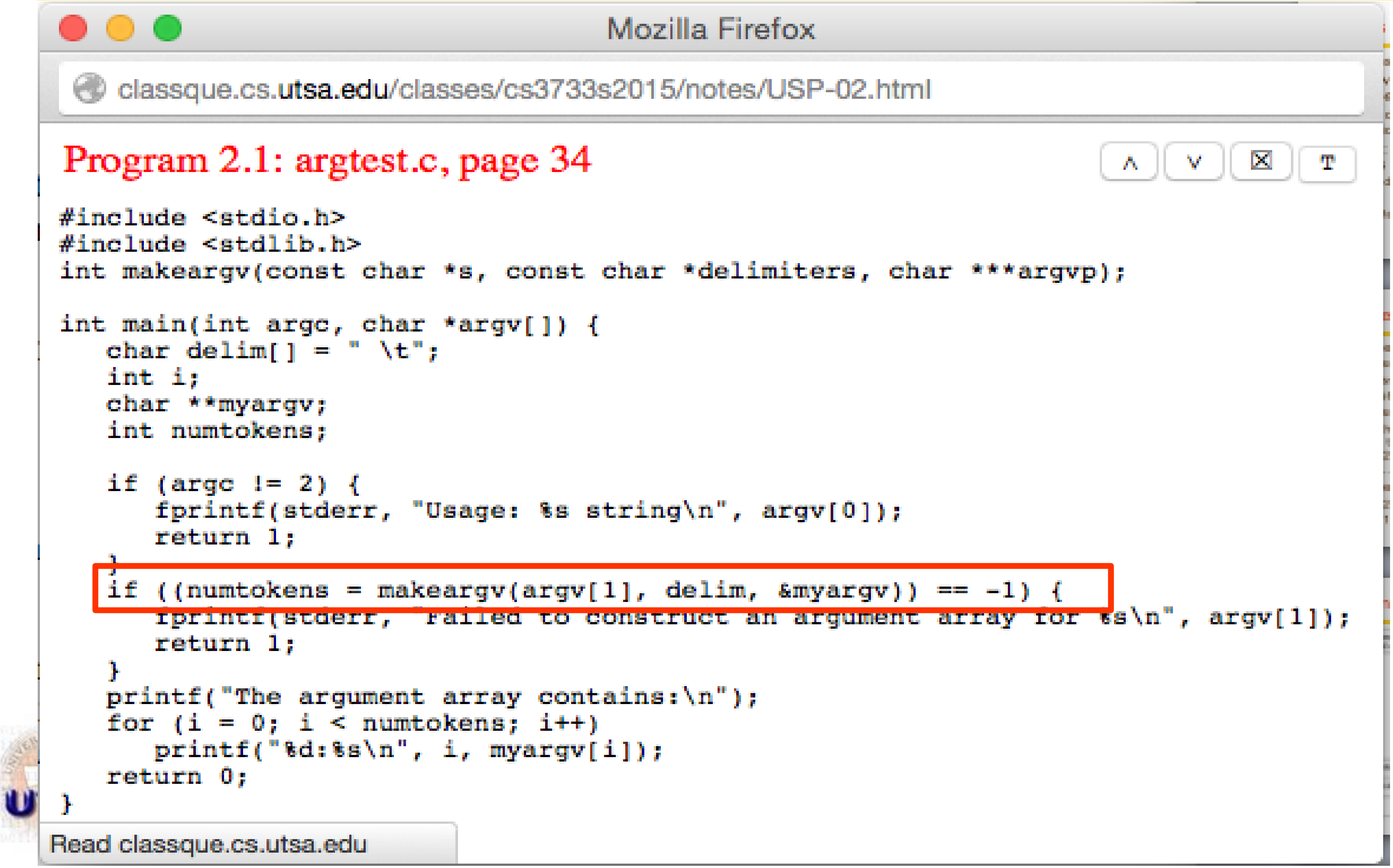
```
char **makeargv(char *s);
char **makeargv(char *s,int *ptrnum);
int     makeargv(char *s,char ***argvp);
```

- 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.

An example to use makeargv ()


Mozilla Firefox
classque.cs.utsa.edu/classes/cs3733s2015/notes/USP-02.html
^
v
☒
T
Read classque.cs.utsa.edu

Program 2.1: argtest.c, page 34

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

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

   if (argc != 2) {
      fprintf(stderr, "Usage: %s string\n", argv[0]);
      return 1;
   }
   if ((numtokens = makeargv(argv[1], delim, &myargv)) == -1) {
      fprintf(stderr, "Failed to construct an argument array for %s\n", argv[1]);
      return 1;
   }
   printf("The argument array contains:\n");
   for (i = 0; i < numtokens; i++)
      printf("%d:%s\n", i, myargv[i]);
   return 0;
}
```

Self-study

How to Write makeargv ()

See https://en.wikipedia.org/wiki/Restrict for restrict

- □ Use the function strtok():

  #include <string.h>

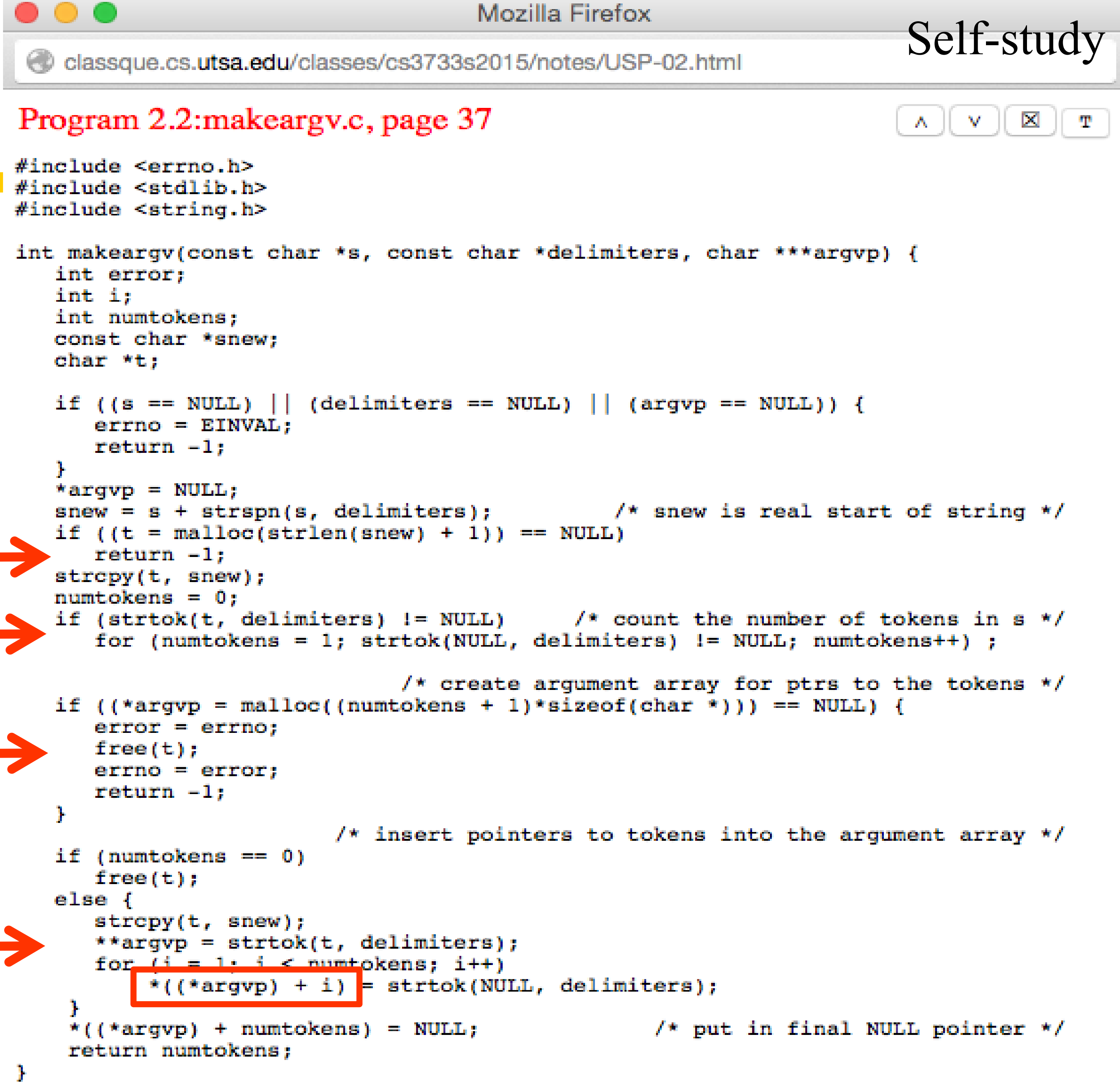
  char *strtok(char *restrict s1, const char *restrict delimit);

  - ➢ s1 is the string to parse; Note that, s1 will be modified;
  - ➢ delimit is a string of delimiters
  - ➢ returns a pointer to the next token or NULL if none left.

- □ To NOT modify the string passed to makeargv()
  - ➢ Get a copy of the string;
  - ➢ Make a pass with strtok to count the tokens
  - ➢ Use the count to allocate the myargv array
  - ➢ Second pass with strtok to set pointers in myargv array

The Function: makeargv()

Self-study



Program 2.2:makeargv.c, page 37

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

int makeargv(const char *s, const char *delimiters, char ***argvp) {
   int error;
   int i;
   int numtokens;
   const char *snew;
   char *t;

   if ((s == NULL) || (delimiters == NULL) || (argvp == NULL)) {
      errno = EINVAL;
      return -1;
   }
   *argvp = NULL;
   snew = s + strspn(s, delimiters);         /* snew is real start of string */
   if ((t = malloc(strlen(snew) + 1)) == NULL)
      return -1;
   strcpy(t, snew);
   numtokens = 0;
   if (strtok(t, delimiters) != NULL)     /* count the number of tokens in s */
      for (numtokens = 1; strtok(NULL, delimiters) != NULL; numtokens++) ;

                             /* create argument array for ptrs to the tokens */
   if ((*argvp = malloc((numtokens + 1)*sizeof(char *))) == NULL) {
      error = errno;
      free(t);
      errno = error;
      return -1;
   }
                        /* insert pointers to tokens into the argument array */
   if (numtokens == 0)
      free(t);
   else {
      strcpy(t, snew);
      **argvp = strtok(t, delimiters);
      for (i = 1; i < numtokens; i++)
          *((*argvp) + i) = strtok(NULL, delimiters);
    }
    *((*argvp) + numtokens) = NULL;             /* put in final NULL pointer */
    return numtokens;
}
```

copy of the string

Count tokens

Allocate argvp

Set pointers in argvp

Re-write this using array notation

Result Array of The Example

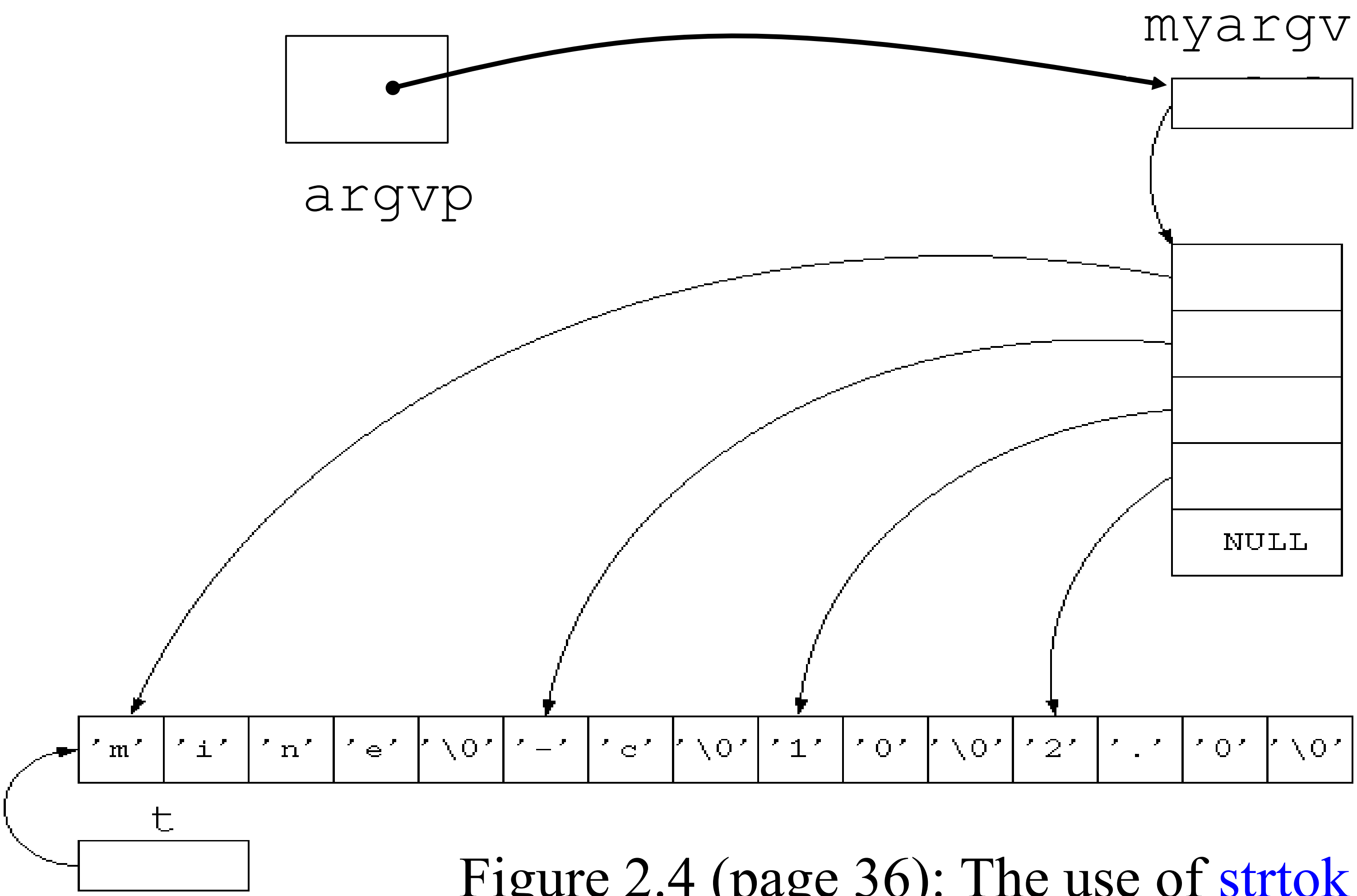
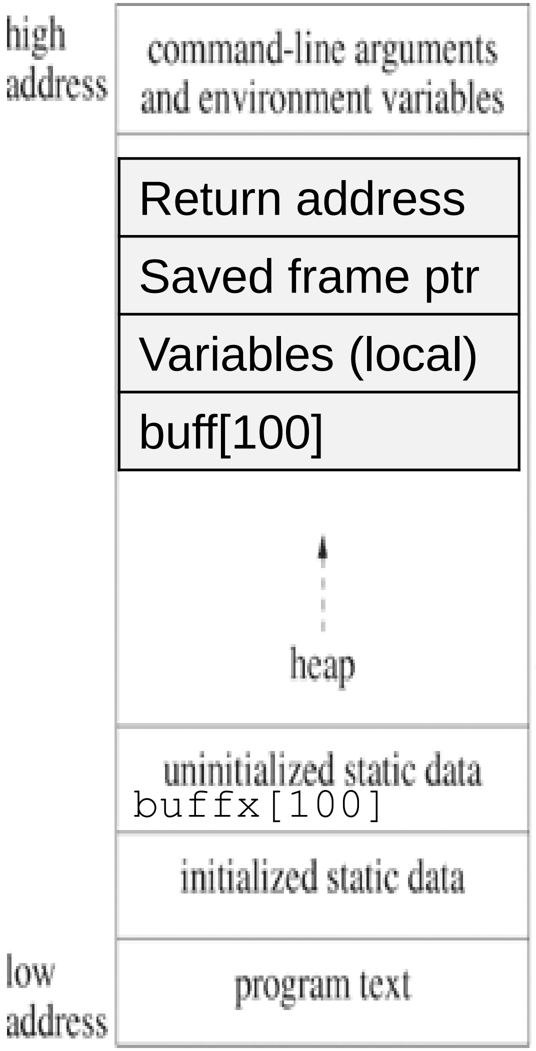


Figure 2.4 (page 36): The use of strtok to allocate strings in place for makeargv.

Safety Issues: What was wrong here?

```
char *get_me_a_name(){
     char buff[100];
     scanf(“%s”, buff);
     return buff;
}
char *get_me_a_name(){
     static char buffx[100];
     scanf(“%s”, buffx);
     return buffx;
}
```



high address

command-line arguments and environment variables

Return address

Saved frame ptr

Variables (local)

buff[100]

heap

uninitialized static data
buffx[100]

initialized static data

low address

program text

Safety Issues of strtok ()

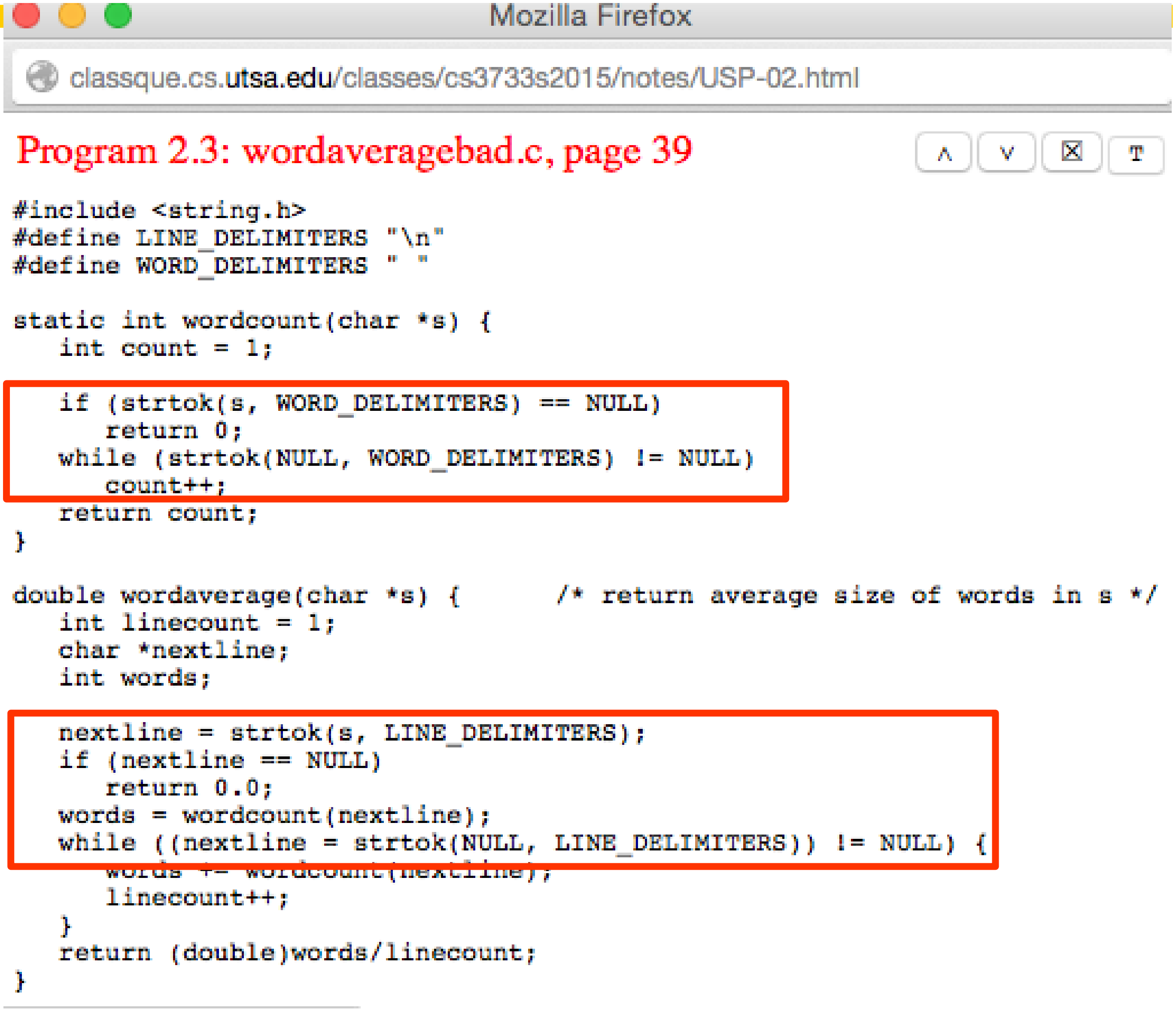
- strtok() is not safe to use with threads
  - it remembers the previous state (i.e., contains a static pointer)

- Programs using strtok can fail even w/o threads
- Suppose to write a program that calculates the average number of words per line in a text file
  - double wordaverage(char *s);
  - Parse input string into lines using strtok;
  - then call a function: int wordcount(char *s);
  - If wordcount() also uses strtok(), program fails!

Self-study

The Fail Version of wordaverage ()

Both functions use strtok()



Program 2.3: wordaveragebad.c, page 39

```
#include <string.h>
#define LINE_DELIMITERS "\n"
#define WORD_DELIMITERS " "

static int wordcount(char *s) {
   int count = 1;

   if (strtok(s, WORD_DELIMITERS) == NULL)
      return 0;
   while (strtok(NULL, WORD_DELIMITERS) != NULL)
      count++;
   return count;
}

double wordaverage(char *s) {       /* return average size of words in s */
   int linecount = 1;
   char *nextline;
   int words;

   nextline = strtok(s, LINE_DELIMITERS);
   if (nextline == NULL)
      return 0.0;
   words = wordcount(nextline);
   while ((nextline = strtok(NULL, LINE_DELIMITERS)) != NULL) {
      words += wordcount(nextline);
      linecount++;
   }
   return (double)words/linecount;
}
```

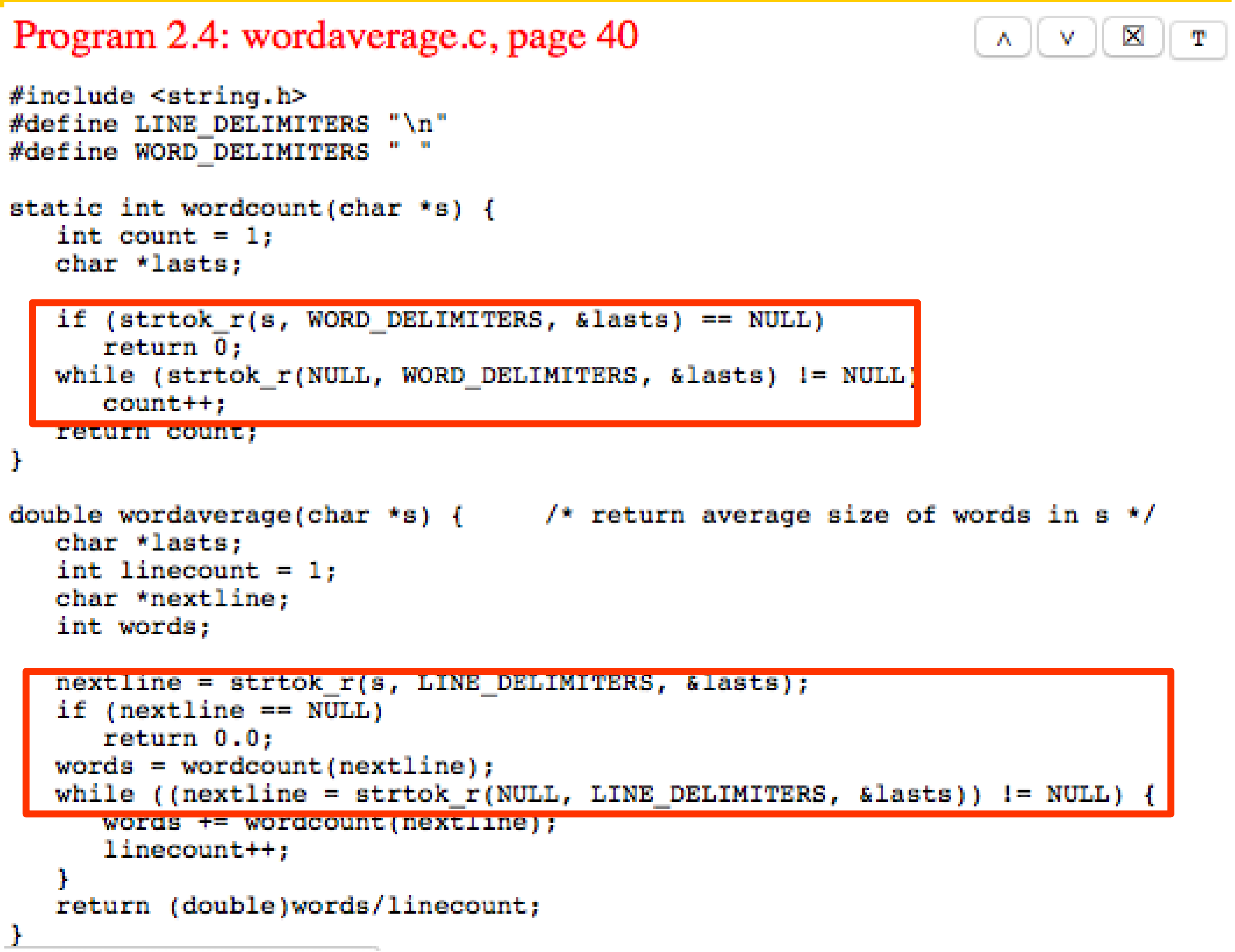
Making Functions Safe

- A safe version of strtok: strtok_r()
  - #include <string.h>
  - char *strtok_**r**(char *restrict s1, const char *restrict s2, char **restrict lasts);
  - A char pointer to hold the current position

- Use strtok_r() in either wordaverage() or wordcount() will solve above problem!

Self-study

The Safe Version of wordaverage ()

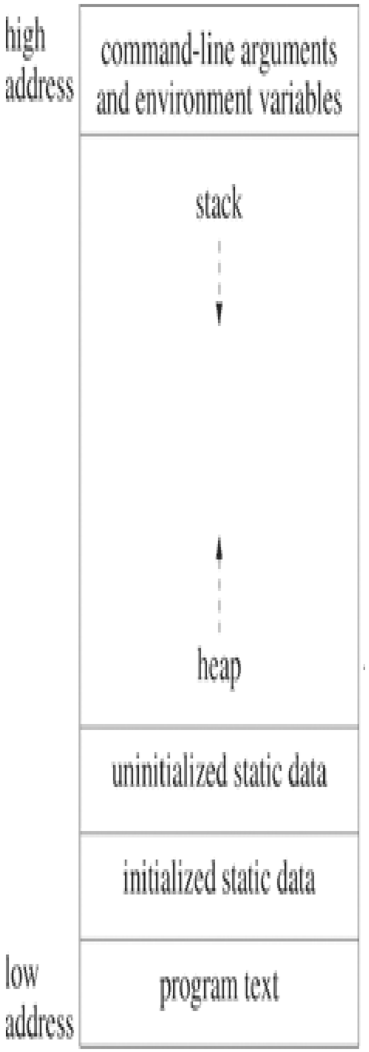


Program 2.4: wordaverage.c, page 40

```
#include <string.h>
#define LINE_DELIMITERS "\n"
#define WORD_DELIMITERS " "

static int wordcount(char *s) {
   int count = 1;
   char *lasts;

   if (strtok_r(s, WORD_DELIMITERS, &lasts) == NULL)
      return 0;
   while (strtok_r(NULL, WORD_DELIMITERS, &lasts) != NULL)
      count++;
   return count;
}

double wordaverage(char *s) {      /* return average size of words in s */
   char *lasts;
   int linecount = 1;
   char *nextline;
   int words;

   nextline = strtok_r(s, LINE_DELIMITERS, &lasts);
   if (nextline == NULL)
      return 0.0;
   words = wordcount(nextline);
   while ((nextline = strtok_r(NULL, LINE_DELIMITERS, &lasts)) != NULL) {
      words += wordcount(nextline);
      linecount++;
   }
   return (double)words/linecount;
}
```

Storage Classes (USP Appendix A.5)

- **Static** vs. **automatic**
  - Static storage class: **variables** that, once allocated, **persist throughout the execution of a program**
  - Automatic storage class: variables which come into existence when block in which they are declared is entered; **discarded** when the defining block is exited
- Variables
  - Variables defined **outside any functions** have **static** storage class.
  - Declared **inside a function** have **automatic storage class** (unless they are declared to be **static**),
  - Automaic ones are usually allocated on the stack



high address

command-line arguments and environment variables

stack

heap

uninitialized static data

initialized static data

program text

low address

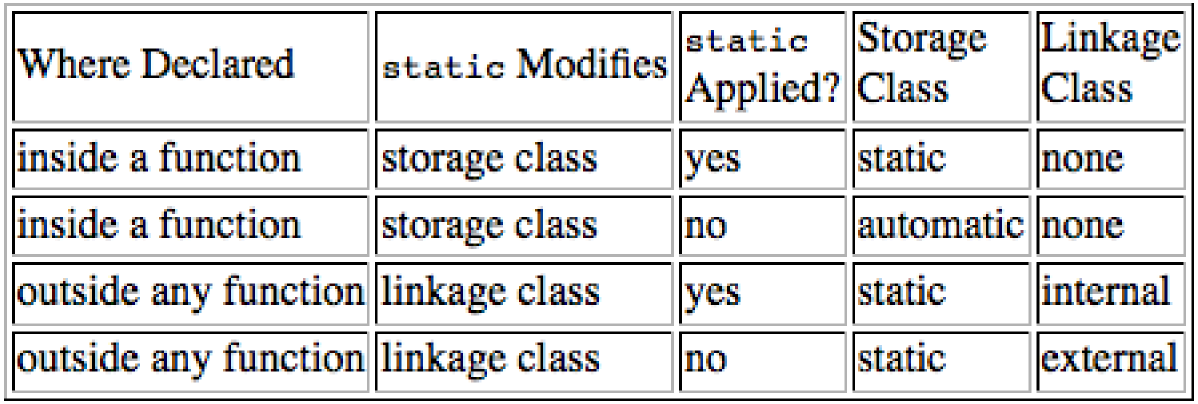
Linkage Classes

- *static* has two meanings is C
  - One related to storage class
  - The other to linkage class
- Linkage classes: determines whether variables can be accessed in files other than the one in which they are declared
  - **Internal** linkage class: can only be accessed in the file in which they are declared
  - **External** linkage class: can be accessed in other files.

Linkage Classes (cont.)

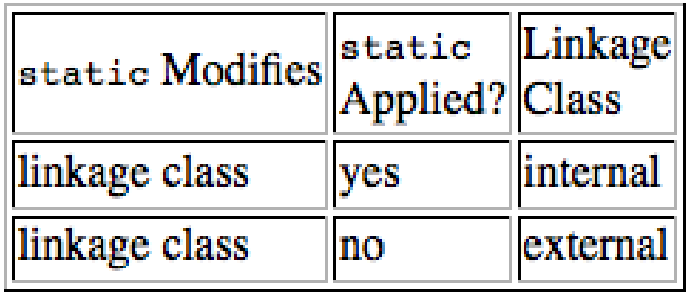
- Variables
  - Declared **outside any function** and **function name identifiers** have **external** linkage by default; however, they can be given internal linkage with the key word *static*
  - Declared **inside a function** are only known inside that function and are said to **have no linkage**

Variables



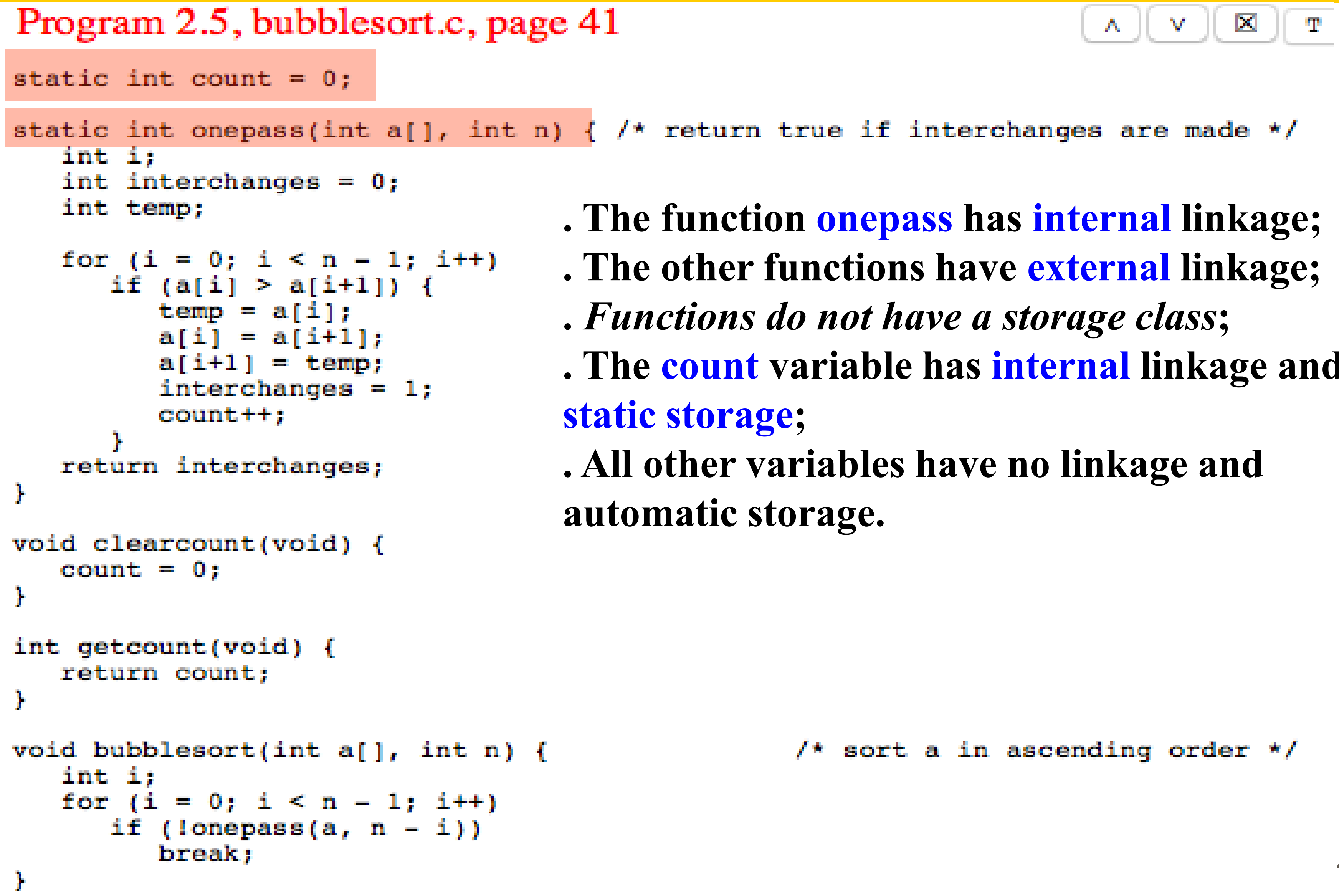
Where Declared	static Modifies	static Applied?	Storage Class	Linkage Class
inside a function	storage class	yes	static	none
inside a function	storage class	no	automatic	none
outside any function	linkage class	yes	static	internal
outside any function	linkage class	no	static	external

Functions



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

Example Program: bubblesort.c



Program 2.5, bubblesort.c, page 41

```
static int count = 0;

static int onepass(int a[], int n) { /* return true if interchanges are made */
   int i;
   int interchanges = 0;
   int temp;

   for (i = 0; i < n - 1; i++)
      if (a[i] > a[i+1]) {
         temp = a[i];
         a[i] = a[i+1];
         a[i+1] = temp;
         interchanges = 1;
         count++;
      }
   return interchanges;
}

void clearcount(void) {
   count = 0;
}

int getcount(void) {
   return count;
}

void bubblesort(int a[], int n) {               /* sort a in ascending order */
   int i;
   for (i = 0; i < n - 1; i++)
      if (!onepass(a, n - i))
         break;
}
```

. The function onepass has internal linkage;

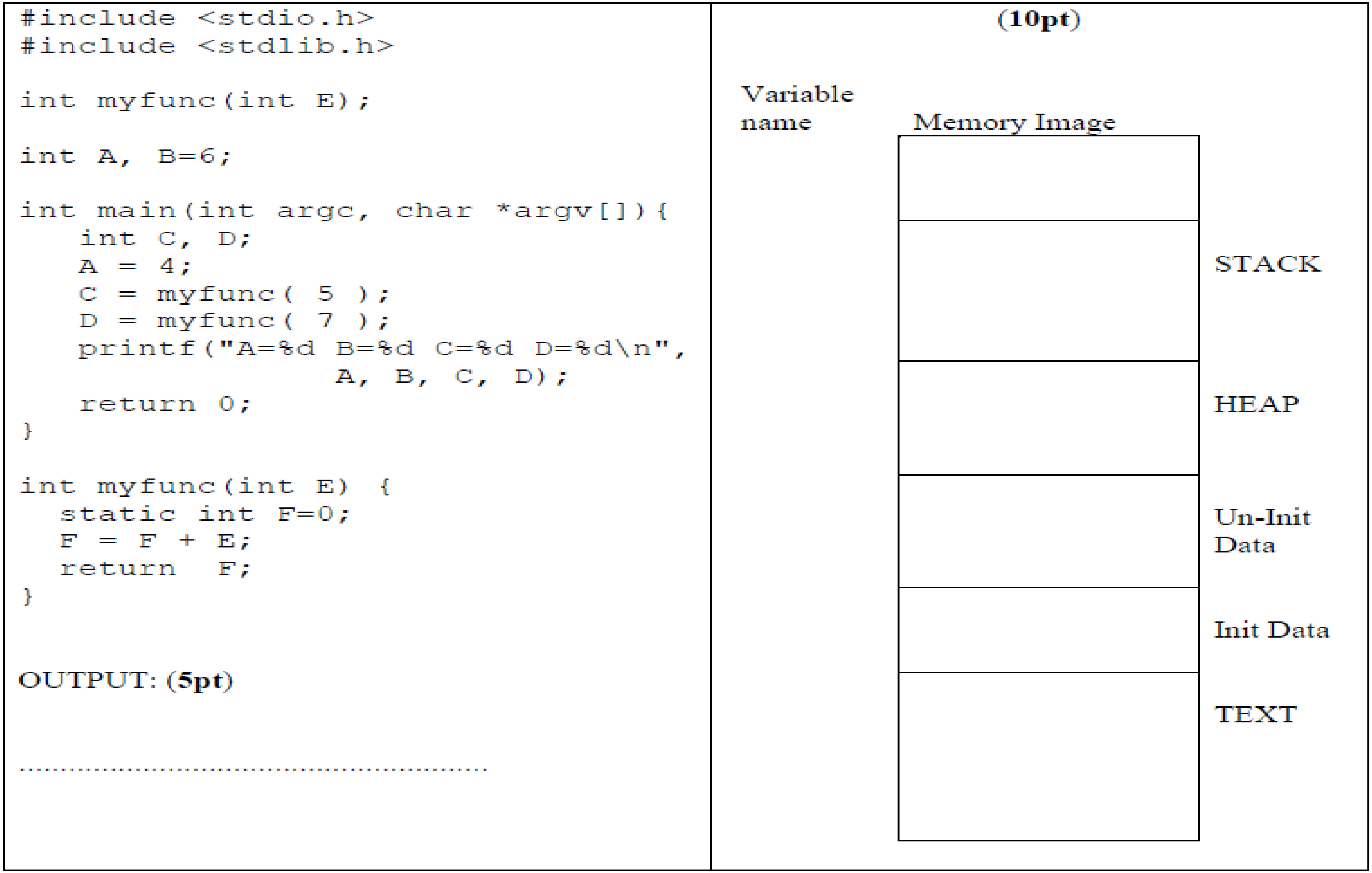
. The other functions have external linkage;

. *Functions do not have a storage class*;

. The count variable has internal linkage and static storage;

. All other variables have no linkage and automatic storage.

Example: Show the memory image of the following program. Specifically show where the variables reside, and how their values change. Also give the output.



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

int myfunc(int E);

int A, B=6;

int main(int argc, char *argv[]){
   int C, D;
   A = 4;
   C = myfunc( 5 );
   D = myfunc( 7 );
   printf("A=%d B=%d C=%d D=%d\n",
                 A, B, C, D);
   return 0;
}

int myfunc(int E) {
  static int F=0;
  F = F + E;
  return  F;
}
```

OUTPUT: **(5pt)**

....

(10pt)

Variable name

Memory Image

STACK

HEAP

Un-Init Data

Init Data

TEXT

Example cont’d

- In the previous example, suppose we remove the "**static**" from
  - `"static int F=0;"` and
  - we just have `"int F=0;"` in myfunc().
- Explain how this may affect the program and what might be the new output.

Outline

- Programs, Processes and Threads
- Process creation and its components
- States of a process and transitions
- PCB: Process Control Block
- Process (program image) in memory
  - Pointers
  - Argument Arrays
  - Making Functions Safe
  - Storage and Linkage Classes

(SGG 3.1, 3.2, USP 2)

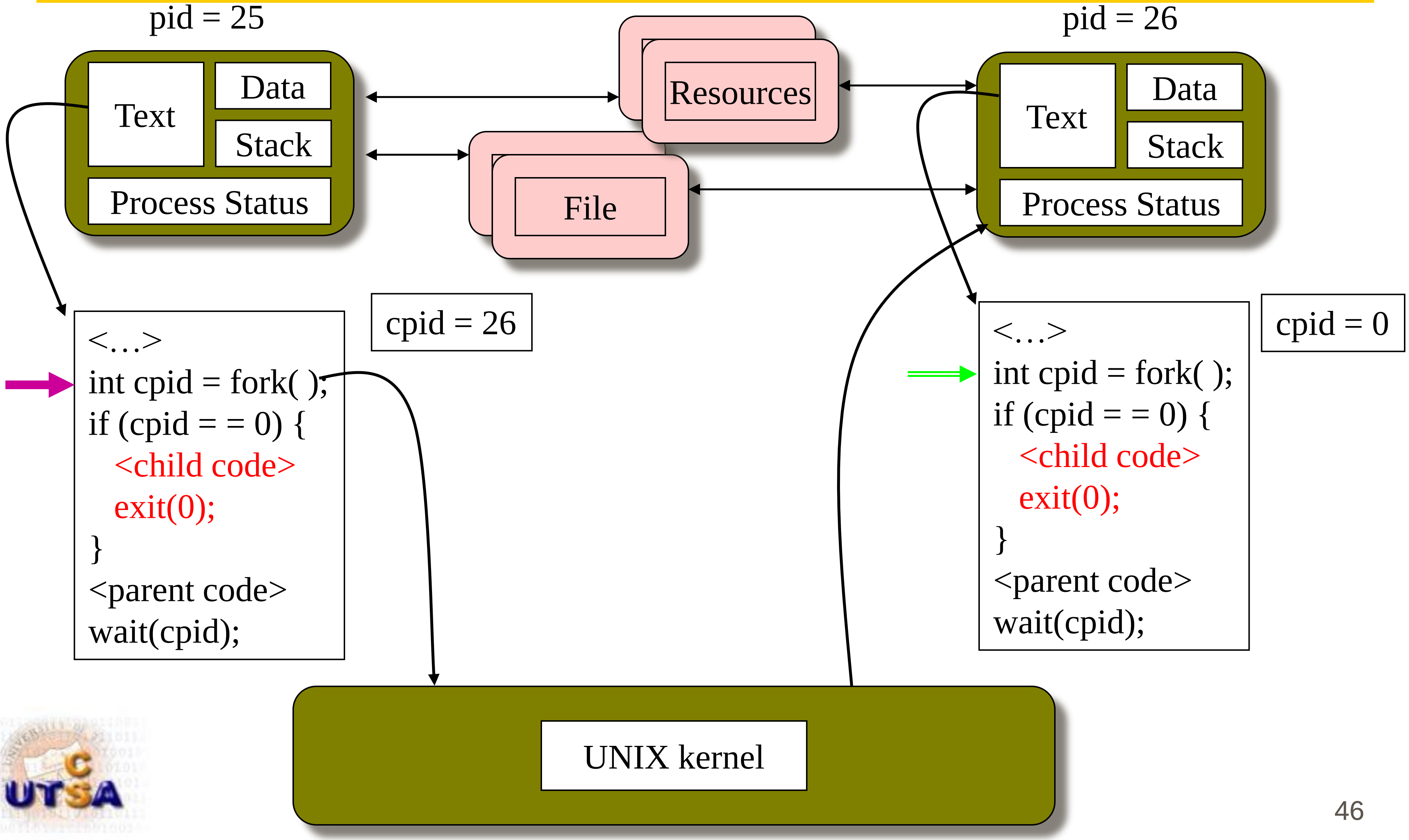
- Process Creation in UNIX (USP 3)

Process Creation in UNIX

Process Creation in UNIX

- Process has a *process identifier (pid)*
- A process (parent) creates another process (child) using the system call ***fork***
- The new child process has a ***separate*** *copy* of the parent process’s address space (code, data etc.).
- Both parent and child processes continue execution at the instruction right after the ***fork*** system call
  - Return value of 0 → new (child) process continues
  - Otherwise, return non-zero pid of child process → parent process continues

An Example: `fork( )` In UNIX



```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

```
<…>
int cpid = fork( );
if (cpid = = 0) {
   <child code>
   exit(0);
}
<parent code>
wait(cpid);
```

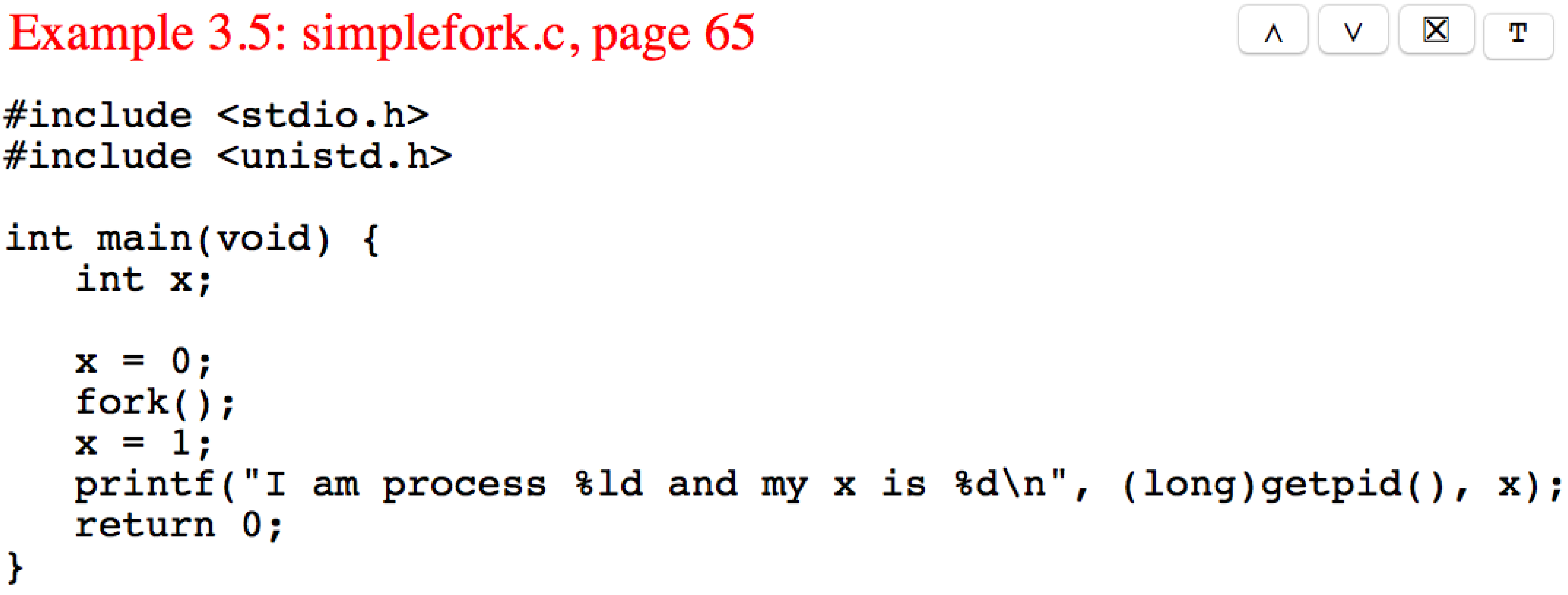
Create Process with `fork( )`

- Usage of `fork( )`

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

- Return values of fork()
  - Error → -1
  - Return to child process → 0
  - Return to parent process → child’s pid

Example 3.5 (p65): simplefork.c

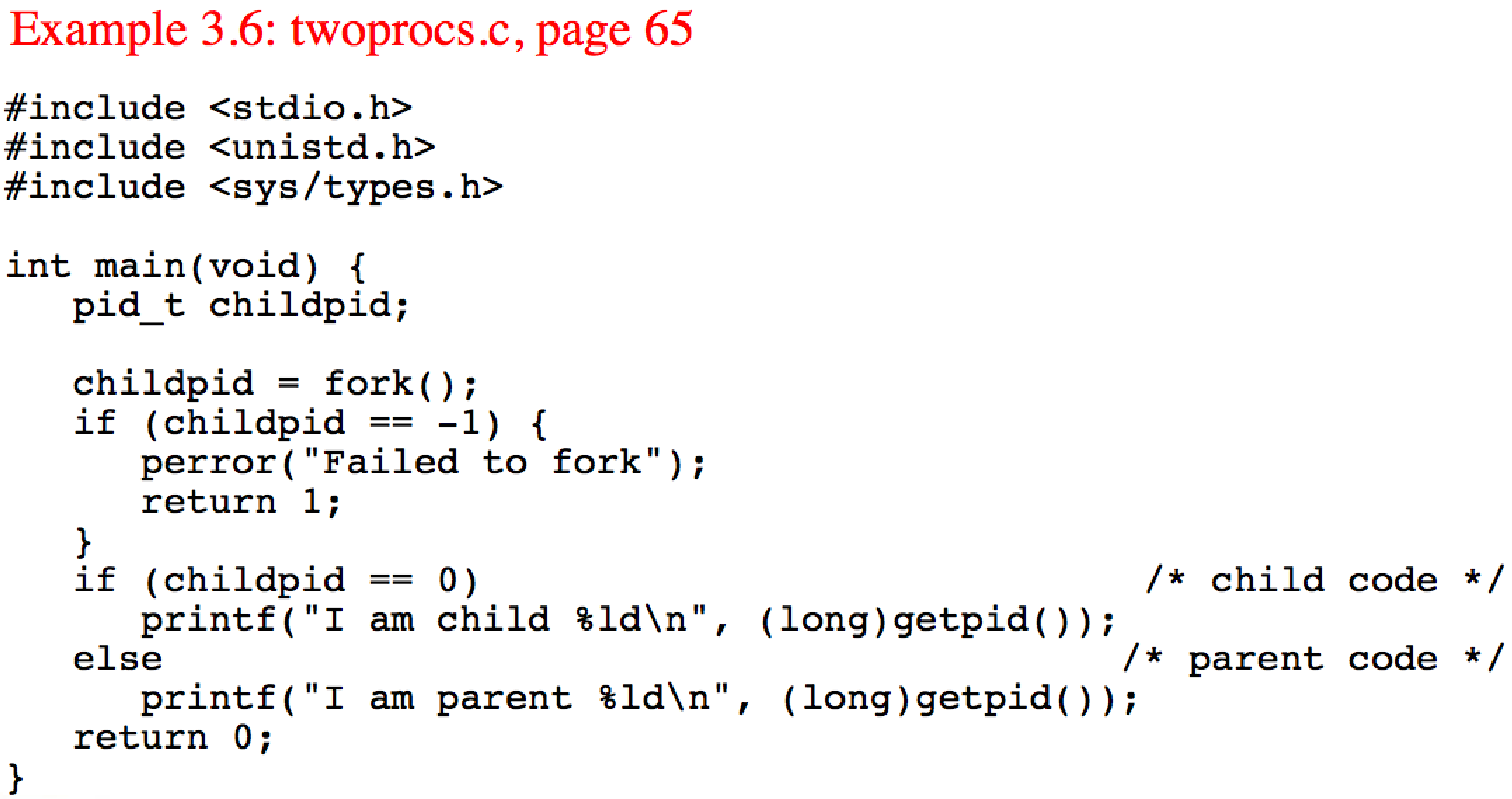


Example 3.5: simplefork.c, page 65

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

int main(void) {
   int x;

   x = 0;
   fork();
   x = 1;
   printf("I am process %ld and my x is %d\n", (long)getpid(), x);
   return 0;
}
```

What is the value of x for each process?

Example 3.6 (p65): twoprocs.c



Example 3.6: twoprocs.c, page 65

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
   pid_t childpid;

   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                             /* child code */
      printf("I am child %ld\n", (long)getpid());
   else                                          /* parent code */
      printf("I am parent %ld\n", (long)getpid());
   return 0;
}
```

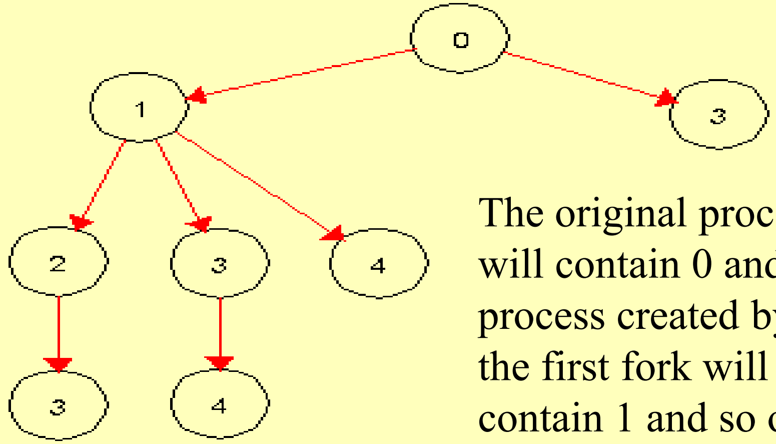
Parent and child print different messages

Graph of Process Relation

❑Each process will be represented by a small circle containing a label representing which fork created the process, what was the output, values of some variables etc...

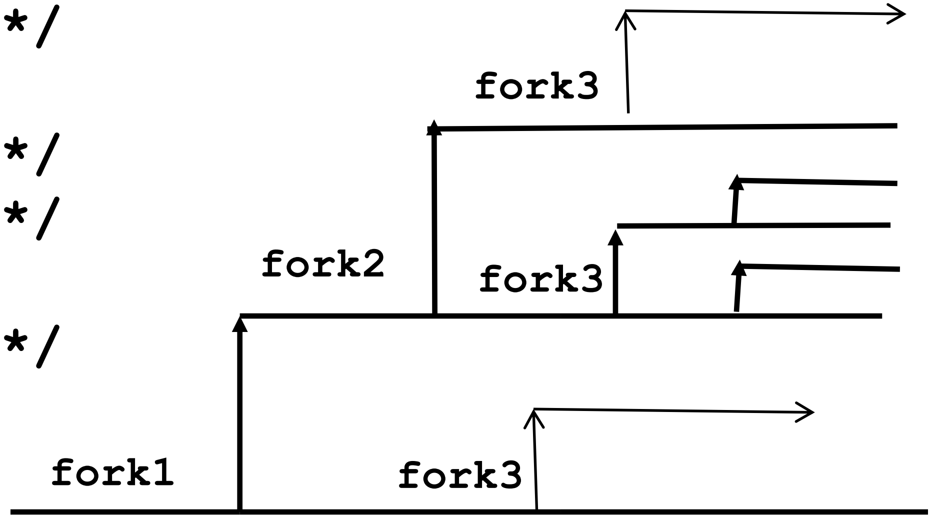
❑There will be arrows from each parent to all of its children. Each arrow should point in a downward direction.

```
//Given program segment
c2 = 0;
c1 = fork();  /* fork number 1 */
if (c1 == 0)
  c2 = fork();/* fork number 2 */
fork();       /* fork number 3 */
if (c2 > 0)
   fork();    /* fork number 4 */
```



The original process will contain 0 and the process created by the first fork will contain 1 and so on.

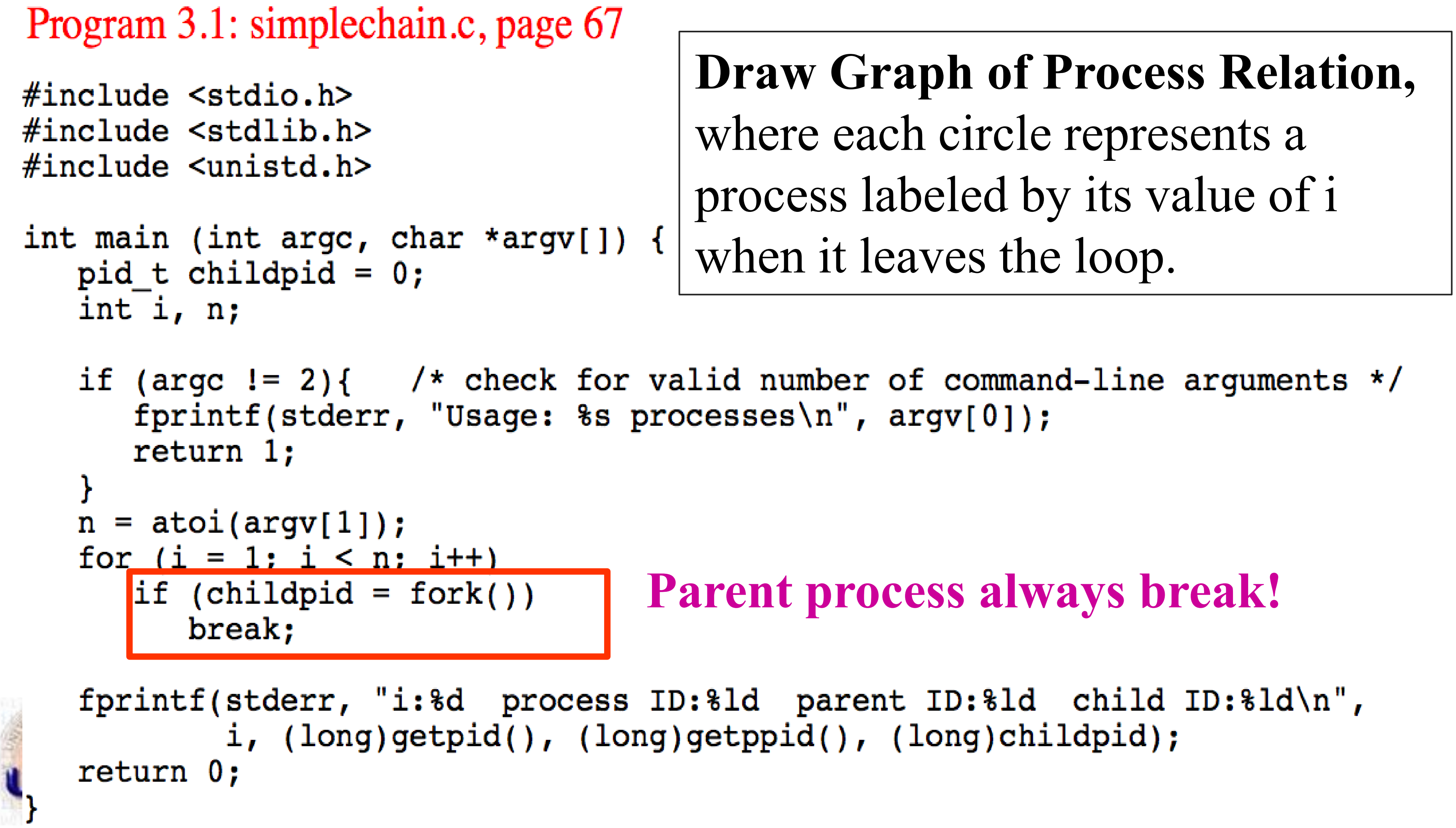
Label: fork number



How many process will be created?
What is the relationship among them?

Program 3.1 (p67): simplechain.c when n is 4

- A chain of processes, parent always breaks

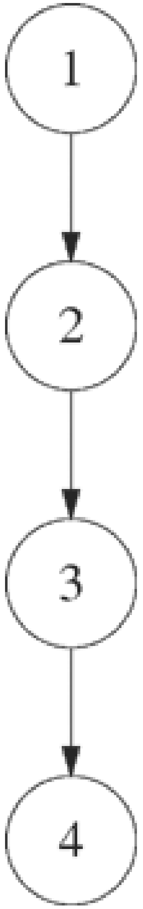


Program 3.1: simplechain.c, page 67

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

int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){   /* check for valid number of command-line arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork())
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Draw Graph of Process Relation, where each circle represents a process labeled by its value of i when it leaves the loop.

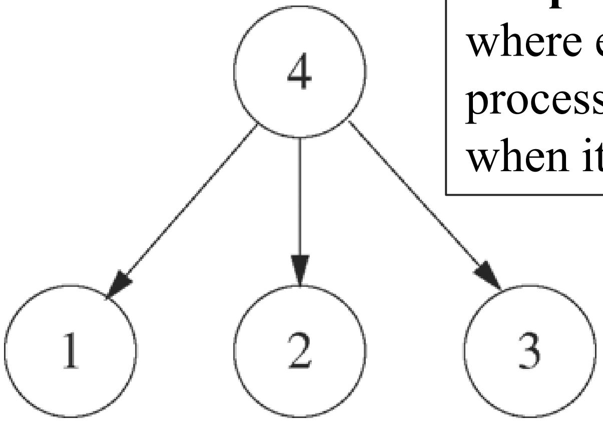
Parent process always break!



Program 3.2 (p68): simplefan.c when n is 4

- A fan of processes, the child breaks

Write the code for a given Graph of Process Relation, where each circle represents a process labeled by its value of i when it leaves the loop.



Program 3.2: simplefan.c, page 68

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

int main (int argc, char *argv[])
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){   /* check for valid number of command-line arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if ((childpid = fork()) <= 0)
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Child process always break!

REF

Create Process w. exec: different program

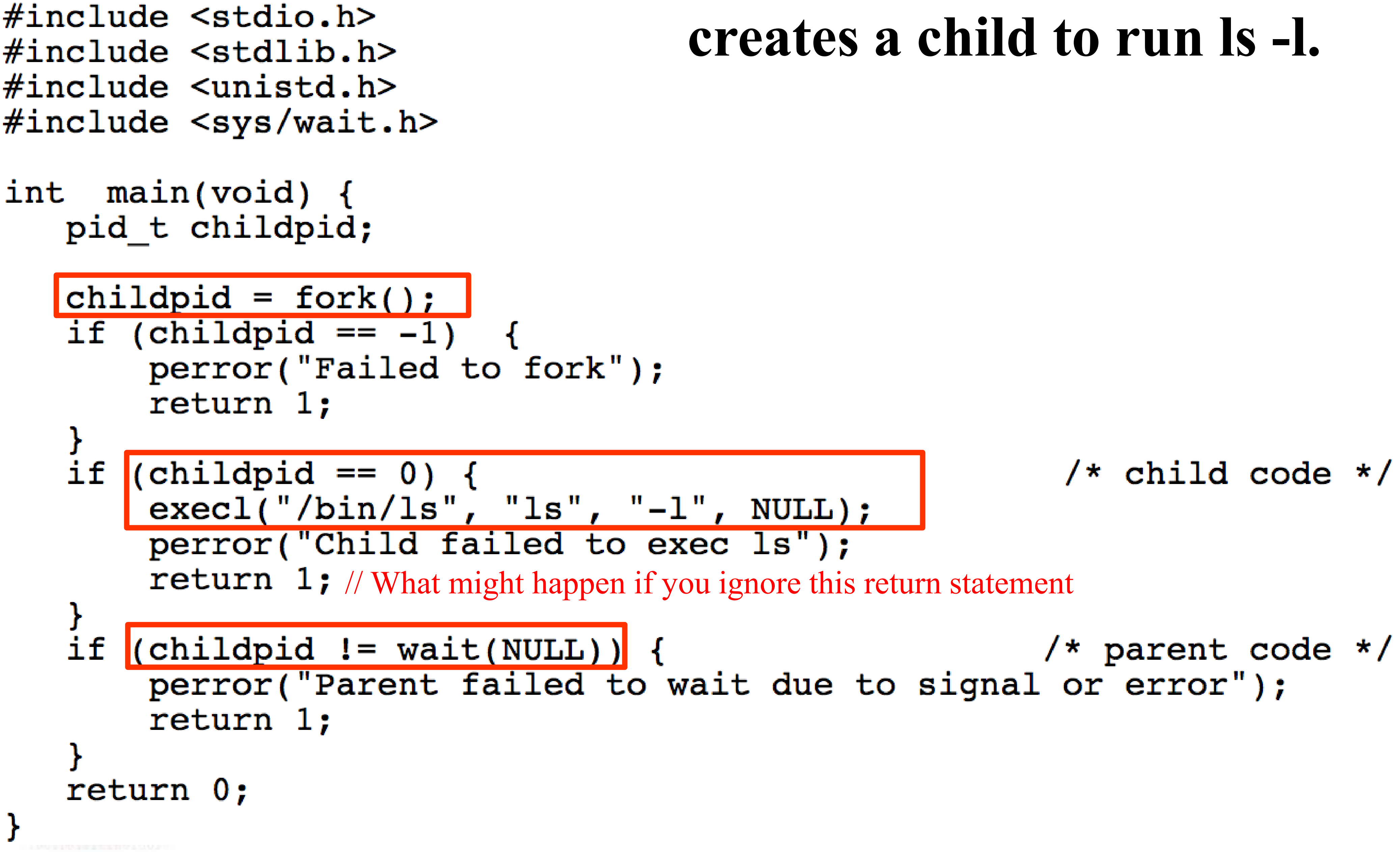
Many of the attributes of a process are inherited after an exec. These include the processID, the userID (owner), current working directory, and open file descriptors, etc.

```
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0, ... /*, char *(0) */);
int execle (const char *path, const char *arg0, ... /*, char *(0),
            char *const envp[] */);
int execlp (const char *file, const char *arg0, ... /*, char *(0) */);
int execv(const char *path, char *const argv[]);
int execve (const char *path, char *const argv[], char *const envp[]);
int execvp (const char *file, char *const argv[]);
```

- **l** forms take a variable number of parameters with a NULL-terminated list of command line arguments
- **v** form take an argv array which can be created with makeargv
- **p** forms use PATH variable to search for the executable
- **e** forms allow to set the environment of the new process

Program 3.4: execls.c, page 79

creates a child to run ls -l.



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int  main(void) {
   pid_t childpid;

   childpid = fork();
   if (childpid == -1)  {
       perror("Failed to fork");
       return 1;
   }
   if (childpid == 0) {                               /* child code */
       execl("/bin/ls", "ls", "-l", NULL);
       perror("Child failed to exec ls");
       return 1; // What might happen if you ignore this return statement
   }
   if (childpid != wait(NULL)) {                     /* parent code */
       perror("Parent failed to wait due to signal or error");
       return 1;
   }
   return 0;
}
```

Program 3.5: execcmd.c, page 81

creates a child process to execute a command given on the command line

```c
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "restart.h"

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

   if (argc < 2){      /* check for valid number of command-line arguments */
      fprintf (stderr, "Usage: %s command arg1 arg2 ...\n", argv[0]);
      return 1;
   }
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {                                      /* child code */
      execvp(argv[1], &argv[1]);
      perror("Child failed to execvp the command");
      return 1;
   }
   if (childpid != r_wait(NULL)) {                          /* parent code */
      perror("Parent failed to wait");
      return 1;
   }
   return 0;
}
```

Program 3.6: execcmdargv.c, page 82

Creates child process to execute a command given as a single string on command line

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "restart.h"

int makeargv(const char *s, const char *delimiters, char ***argvp);

int main(int argc, char *argv[]) {
   pid_t childpid;
   char delim[] = " \t";
   char **myargv;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s string\n", argv[0]);
      return 1;
   }
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {                              /* child code */
     if (makeargv(argv[1], delim, &myargv) == -1) {
        perror("Child failed to construct argument array");
     } else {
        execvp(myargv[0], &myargv[0]);
        perror("Child failed to exec command");
     }
     return 1;
   }
   if (childpid != r_wait(NULL)) {                  /* parent code */
      perror("Parent failed to wait");
      return 1;
   }
   return 0;
}
```

REF

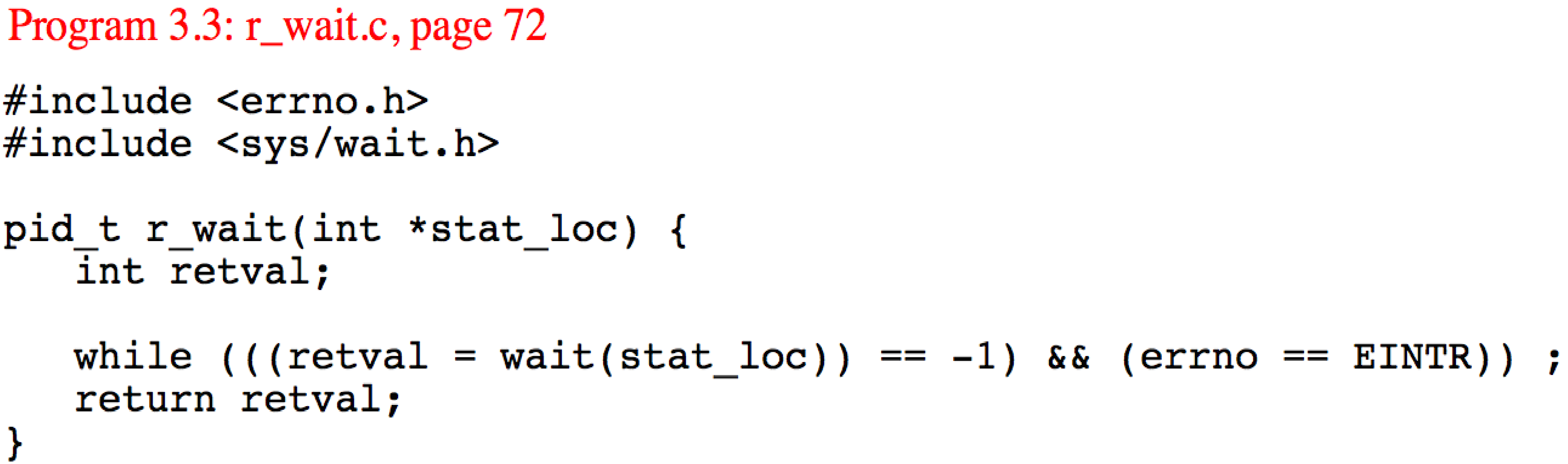
Wait for Processes

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- **wait** : parent blocks until the child finishes
  - If a child terminated, return its pid
  - Otherwise, return -1 and set `errno`
- **waitpid** : parent blocks until a specific child finishes
  - Allow to wait for a particular process (or all if pid=-1);
  - NOHANG option: return 0 if there is a specified child to wait for but it has not yet terminated
- Important values of `errno`
  - ECHILD no unwaited for children;
  - EINTR a signal was caught

Re-Start a wait call

- When the call is interrupted by a signal



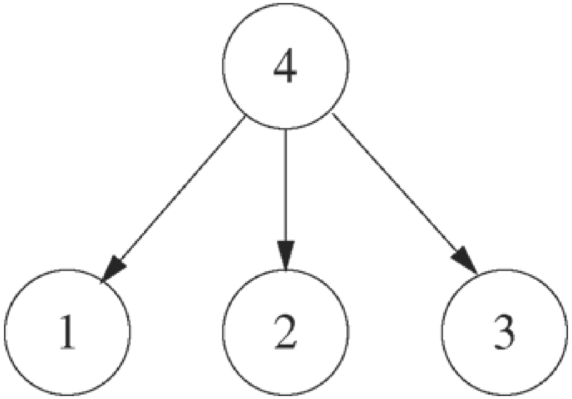
Program 3.3: r_wait.c, page 72

```
#include <errno.h>
#include <sys/wait.h>

pid_t r_wait(int *stat_loc) {
   int retval;

   while (((retval = wait(stat_loc)) == -1) && (errno == EINTR)) ;
   return retval;
}
```

Exercise 3.20: fanwaitmsg.c, page 75

Parent process waits for all of its children

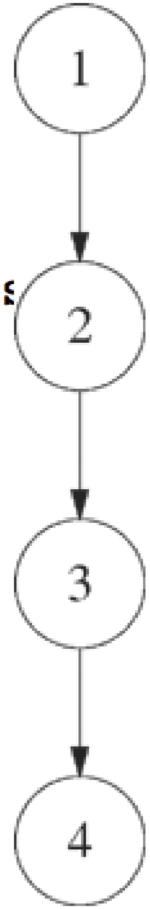
```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){         /* check number of command-line arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if ((childpid = fork()) <= 0)
         break;
   for( ; ; ) {
      childpid = wait(NULL);
      if ((childpid == -1) && (errno != EINTR))
        break;
   }
   fprintf(stderr, "I am process %ld, my parent is %ld\n",
                   (long)getpid(), (long)getppid());
   return 0;
}
```



Exercise 3.21: chainwaitmsg.c, page 76

each process waits for its child

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc, char *argv[]) {
   pid_t childpid;
   int i, n;
   pid_t waitreturn;

   if (argc != 2){   /* check number of command-line arguments
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork())
         break;
   while(childpid != (waitreturn = wait(NULL)))
      if ((waitreturn == -1) && (errno != EINTR))
         break;
   fprintf(stderr, "I am process %ld, my parent is %ld\n",
                     (long)getpid(), (long)getppid());
   return 0;
}
```



Status Values for wait

```
pid_t wait(int *stat_loc);
```

- Status value == 0 if and only if the process terminated normally and returned 0
- Other cases: the status should be examined using macros defined in sys/wait.h

```
#include <sys/wait.h>
WIFEXITED(int stat_val)
WEXITSTATUS(int stat_val)
WIFSIGNALED(int stat_val)
WTERMSIG(int stat_val)
WIFSTOPPED(int stat_val)
WSTOPSIG(int stat_val)
```

- Used in pairs

Example 3.22: showreturnstatus.c, page 77

REF

an example of using status macros

```c
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "restart.h"

void show_return_status(void) {
   pid_t childpid;
   int status;

   childpid = r wait(&status);
   if (childpid == -1)
      perror("Failed to wait for child");
   else if (WIFEXITED(status) && !WEXITSTATUS(status))
      printf("Child %ld terminated normally\n", (long)childpid);
   else if (WIFEXITED(status))
      printf("Child %ld terminated with return status %d\n",
             (long)childpid, WEXITSTATUS(status));
   else if (WIFSIGNALED(status))
      printf("Child %ld terminated due to uncaught signal %d\n",
             (long)childpid, WTERMSIG(status));
   else if (WIFSTOPPED(status))
      printf("Child %ld stopped due to signal %d\n",
             (long)childpid, WSTOPSIG(status));
}
```

Process Termination

- **Voluntarily**
  - Process finishes and asks OS to delete it (**exit**).
  - Output data from child to parent (**wait** or **waitpid**).
  - Process' resources are de-allocated by OS.
- **Involuntarily**
  - Parent terminate execution of children processes (e.g. **TerminateProcess()** in Win32, **abort**)
  - Task assigned to child is no longer required
  - Child has exceeded allocated resources
  - If parent exits
    - ✓ Some operating system do not allow child to continue
      - if its parent terminates (All children terminated - **cascading termination**)
    - ✓ Some operating system do allow, and init owns them
- Parent process is terminated (e.g., due to errors)
  - ✓ What will happen to the children process?!

Orphaned Zombies

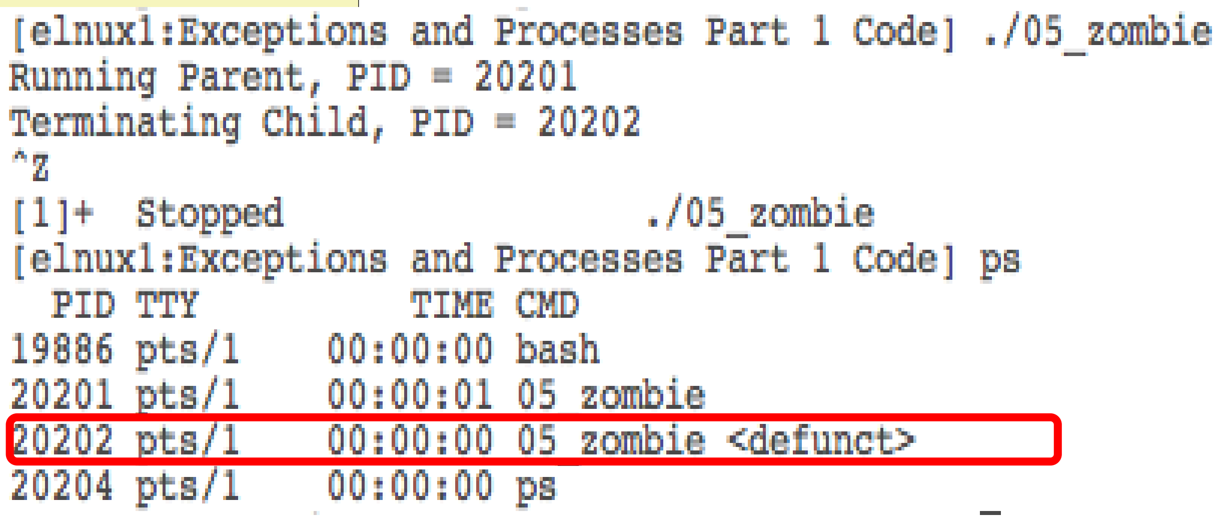
Create & Terminate Processes

What happens here?

Zombie!

```
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1) ; /* Infinite loop */
    }
}
```

05_zombie.c



```
[elnuxl:Exceptions and Processes Part 1 Code] ./05_zombie
Running Parent, PID = 20201
Terminating Child, PID = 20202
^Z
[1]+  Stopped                 ./05_zombie
[elnuxl:Exceptions and Processes Part 1 Code] ps
  PID TTY          TIME CMD
19886 pts/1    00:00:00 bash
20201 pts/1    00:00:01 05_zombie
20202 pts/1    00:00:00 05_zombie <defunct>
20204 pts/1    00:00:00 ps
```

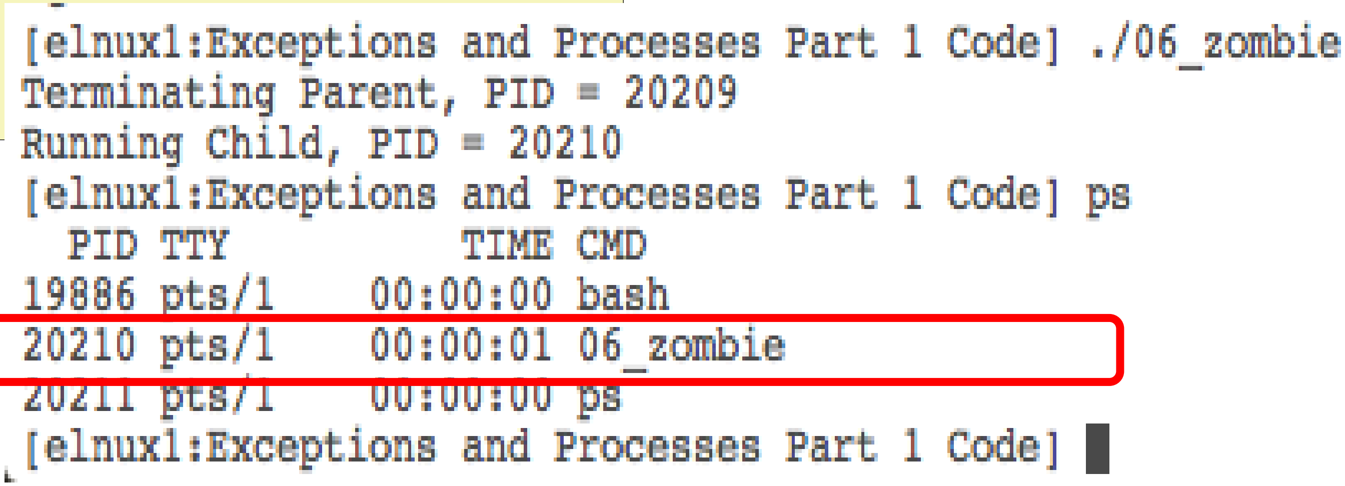
Create & Terminate Processes

What about this one?

```
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

06_zombie.c

Orphaned?



```
[elnux1:Exceptions and Processes Part 1 Code] ./06_zombie
Terminating Parent, PID = 20209
Running Child, PID = 20210
[elnux1:Exceptions and Processes Part 1 Code] ps
  PID TTY          TIME CMD
19886 pts/1    00:00:00 bash
20210 pts/1    00:00:01 06_zombie
20211 pts/1    00:00:00 ps
[elnux1:Exceptions and Processes Part 1 Code] 
```

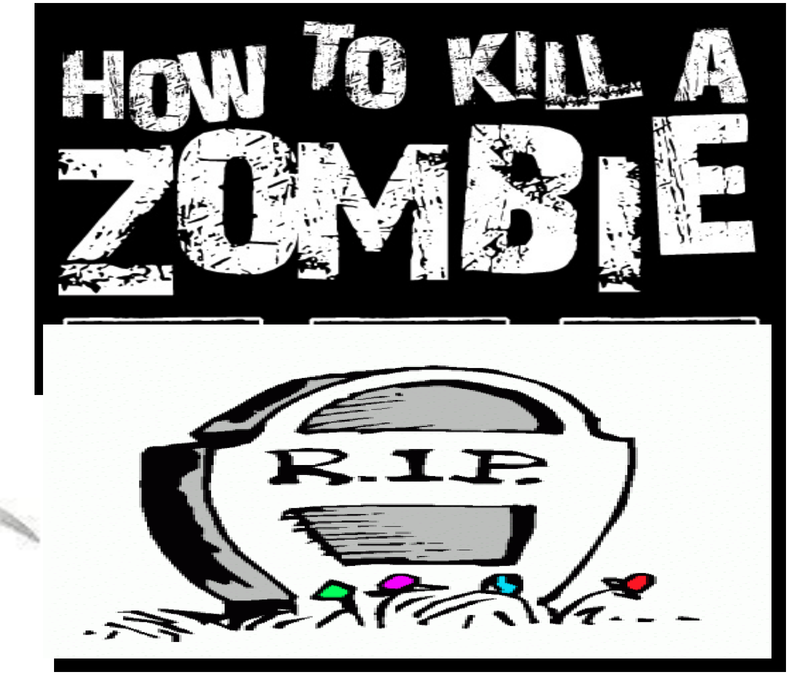
Really hard to detect since the main process already exited!

Parents must wait to avoid zombies

Need a way to kill the zombies!

`wait()`

`waitpid()`



This is called reaping!

How do we “reap” a child process programmatically?

What to do if parents did not wait?

- Orphaned processes or zombies are adopted by a special process called init (pid=1)
- Periodically waits for children…

Solve quiz 3: Unix proc creation

MORE EXAMPLES

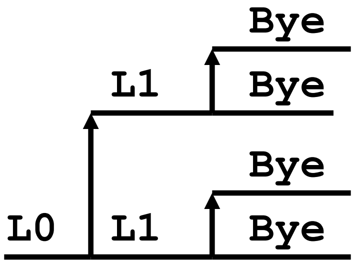
More Examples on process creation and termination

SELF-STUDY

Create & Terminate Processes

What does this print out?

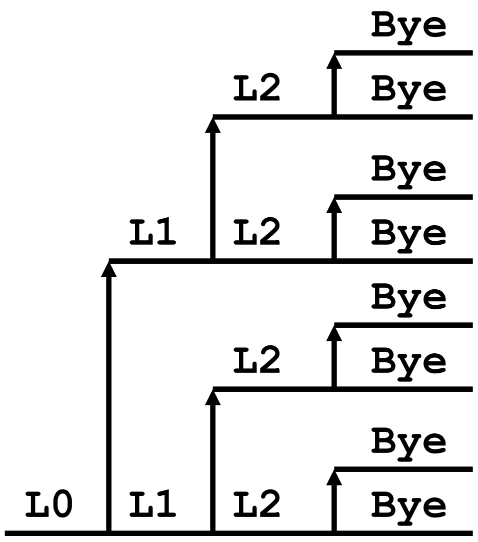
```
void main()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Create & Terminate Processes

What does this print out?

```
void main()
{
     printf("L0\n");
     fork();
     printf("L1\n");
     fork();
     printf("L2\n");
     fork();
     printf("Bye\n");
}
```



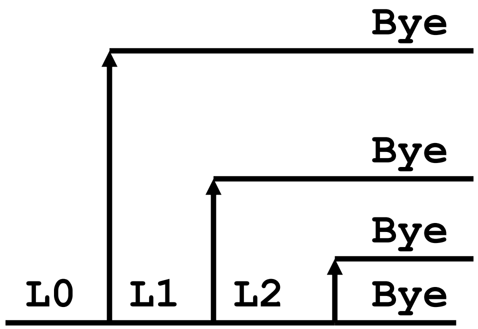
Does it always print in order?

`04_fork.c`

Create & Terminate Processes

What does this print out?

```
void main()
{
   printf("L0\n");
   if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
   }
   printf("Bye\n");
}
```



05_fork.c

Activity

```
void main()
{
    if (fork() == 0) {
      printf(“a”);
    }
    else {
      printf(“b”);
      waitpid(-1, NULL, 0);
    }
    printf(“c”);
    exit(0);
}
```

List all the possible output sequences for this program.

Activity

```
void main()
{
    if (fork() == 0) {
      printf(“a”);
    }
    else {
      printf(“b”);
      waitpid(-1, NULL, 0);
    }
    printf(“c”);
    exit(0);
}
```

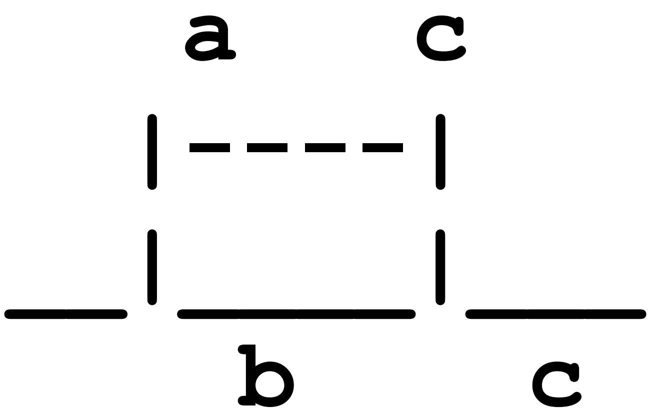
List all the possible output sequences for this program.

Solution:
We can’t make any assumption about the execution order of the parent and child. Thus, any topological sort of a -> c and b-> c is possible:

acbc

abcc

bacc



Create & Terminate Processes

int wait(int* child_status)

fork a child processes

```
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
            pid_t wpid = wait(&child_status);
            if (WIFEXITED(child_status))
                printf("Child %d terminated with exit status %d\n",
                           wpid, WEXITSTATUS(child_status));
            else
                printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Create & Terminate Processes

int wait(int* child_status)

```
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
            pid_t wpid = wait(&child_status);
            if (WIFEXITED(child_status))
                printf("Child %d terminated with exit status %d\n",
                          wpid, WEXITSTATUS(child_status));
            else
                printf("Child %d terminated abnormally\n", wpid);
    }
}
```

wait for each to terminate

Create & Terminate Processes

```
int wait(int* child_status)
```

wait for each to terminate

```
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
            pid_t wpid = wait(&child_status);
            if (WIFEXITED(child_status))
                printf("Child %d terminated with exit status %d\n",
                                wpid, WEXITSTATUS(child_status));
            else
                printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Get Info on Status

Create & Terminate Processes

```
int waitpid(pid, &status, options)
```

wait for PID to terminate

```
void main()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
            pid_t wpid = waitpid(pid[i], &child_status, 0);
            if (WIFEXITED(child_status))
                printf("Child %d terminated with exit status %d\n",
                           wpid, WEXITSTATUS(child_status));
            else
                printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Create & Terminate Processes

```
int waitpid(-1, &status, 0)
```

is the same as…

```
int wait(&status)
```