456: Network-Centric Programming

Yingying (Jennifer) Chen

Web: https://www.winlab.rutgers.edu/~yychen/
Email: yingche@scarletmail.rutgers.edu

Department of Electrical and Computer Engineering
Rutgers University

Working with Unix Processes

Processes

- Recall: Process is a running instance of a program
  - Multiple instances of the same program can run at the same time
  - E.g., open two terminal windows to run the terminal program twice

- Operating systems use the concept of "multitask" to give the impression that all processes run at the same time

Processes

- Try the “ps” command to see a list of active processes

```
chen@chen-VirtualBox:~$ ps
  PID TTY          TIME CMD
15101 pts/17   00:00:00 bash
15627 pts/17   00:00:00 ps
```

- Two processes are shown
  - Bash: the shell running on this terminal
  - The running instance of the “ps” program

Note: ps is short for “process status”

Processes

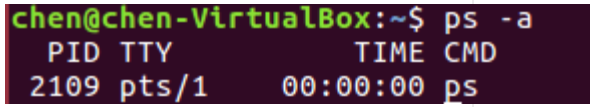
- Try the “ps” command to see a list of active processes

```
chen@chen-VirtualBox:~$ ps
  PID TTY          TIME CMD
15101 pts/17   00:00:00 bash
15627 pts/17   00:00:00 ps
```

- Four items are displayed
  - PID: the identifier of the process
  - TTY (terminal type): the name of the console or terminal
    - pts (pseudo terminal slave)
  - TIME: the amount of CPU time the process has utilized
  - CMD: the name of the command that launched the process

Options of the ps Command

Option	Description
-a	Displays all processes on a terminal, with the exception of group leaders.
-c	Displays scheduler data.
-d	Displays all processes with the exception of session leaders.
-e	Displays all processes.
-f	Displays a full listing.
-g*list*	Displays data for the *list* of group leader IDs.
-j	Displays the process group ID and session ID.
-l	Displays a long listing
-o	Specify the output format (e.g., ps -o pid,user,start_time,cmd)
-p*list*	Displays data for the *list* of process IDs.
-s*list*	Displays data for the *list* of session leader IDs.
-t*list*	Displays data for the *list* of terminals.
-u*list*	Displays data for the *list* of usernames



fork

- System call to create processes
  - Clone (and run) another instance of a process (parent process)
- Create a process using pid_t fork(void)
  - Create a child process (a duplicate process)
  - A child process executes the same program from the same place
  - Typedef pid_t to refer to process IDs (defined in <sys/types.h>)

fork

Why?

- Return twice (for parent and child)
  - For the parent process: the return value is the process ID of the child
  - For the child process, the return value is zero
- The return value
  - Is used to check whether the program is running as the parent or the child process

init process
How is the first process created?

- Every process has an identifier (PID)
- “init” is the very first process to run after booting (PID 1)
  - Created by the kernel
  - parent of all the processes
  - PID 0: kernel scheduler
- Every process except init has a parent
  - Parent is the process that calls fork
- init adopts processes that lose their parent (“orphan process”)

Example (Using fork to duplicate a process): lecture_7_code/fork_example/fork_example.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
pid_t pid;
printf ("the main program process ID is %d\n", (int) getpid ());
pid = fork();
if (pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process ID is %d\n", (int) pid);
}
else
printf ("this is the child process, with id %d\n", (int) getpid ());

return 0;
}
```

Fork system Call – "Returns twice"

Terminal
$ ./main

Example (Concurrent server): lecture_7_code/concurrent_server/server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    int pid;
    char *response = "Response from server";
```

Example (Concurrent server): lecture_7_code/concurrent_server/server.c

```
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0){
    perror("socket failed");
    exit(EXIT_FAILURE);
}
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl (INADDR_ANY);
address.sin_port = htons( PORT );      //attaching socket to the port 8080
if (bind(server_fd, (struct sockaddr *)&address,
                        sizeof(address))<0) {
    perror("bind failed");
    exit(EXIT_FAILURE);
}
if (listen(server_fd, 1024) < 0) {
    perror("listen");
    exit(EXIT_FAILURE);
}
```

Example (Concurrent server): lecture_7_code/concurrent_server/server.c

```
while(1){
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen))<0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }
    /*Create a new process to cope with each new client request*/
        if ( (pid = fork()) == 0) {                                  /* child process */
            close(server_fd);                                       /* close listening socket */
            /* Cope with the request */
            valread = read( new_socket , buffer, 1024);    // receive the msg
            printf("%s\n",buffer );
            write(new_socket, response, strlen(response));   //send the response from server
            printf("Response msg sent\n");
            exit(0);
        }
          close(new_socket);  /* parent closes connected socket */
          printf("after close socket\n");
    }
    return 0;
}
```

Example (Concurrent server): lecture_7_code/concurrent_server/client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
 #include <arpa/inet.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sock1, sock2, valread;
    struct sockaddr_in serv_addr;
    char *hello1 = "Hello from client 1";
    char *hello2 = "Hello from client 2";
    char buffer[1024] = {0};
```

Example (Concurrent server): lecture_7_code/concurrent_server/client.c

```
if ((sock1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }
    if ((sock2 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }
    // initialize serv_addr
    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
```

Example (Concurrent server): lecture_7_code/concurrent_server/client.c

```
if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)  {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
    if (connect(sock1, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
   // send messages via two sockets: sock1
    write(sock1 , hello1 , strlen(hello1));                    //send msg via sock1
    printf("Hello message sent from client 1\n");
    valread = read(sock1, buffer1, 1024);                      //receive msg from server
    printf("%s\n", buffer1);
```

Example (Concurrent server): lecture_7_code/concurrent_server/client.c

```
    if (connect(sock2, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
        printf("\nConnection Failed \n");
        return -1;
    }
    // send messages via two sockets: sock2
    write(sock2 , hello2 , strlen(hello2));                    //send msg via sock2
    printf("Hello message sent from client 2\n");
    valread = read(sock2, buffer2, 1024);                      //receive msg from server
    printf("%s\n", buffer2);
    return 0;
}
```

Terminal 1
$ ./server

Terminal 2
$ ./client

Running a Different Program in the New Process

- fork: Make a child process an exact copy of its parent process
- exec: Execute a new program image in a new process
- Two steps:
  - Use fork() to create a new process
  - Use exec family function to load the new program into that process
    - The base exec followed by one or more letters

Examples of functions in exec family:
execv, execve, execvp, execl, execle and execlp,

Running a Different Program in the New Process – exec family of function

- Loads new program into process
- Exec p/v/e
  - **p**: search current execution path for program named in the file argument
  - **v**: accept an argument list being passed to the new program as an array (**v**ector) of pointers
  - **e**: an array of pointers to environment variables is explicitly passed to the new process image

Example:
execvp (new_program, arg_list);

Example (Using exec to run a different program): lecture_7_code/fork_and_exec/fork_and_exec.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int spawn (char* program, char** arg_list){
pid_t child_pid;
/* Duplicate this process. */
child_pid = fork ();
if (child_pid != 0)            /* This is the parent process. */
    return child_pid;
else {
    /* Now execute the program, searching for it in the current path. */
    execvp (program, arg_list);

    /* The execvp function returns only if an error occurs. */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();   /*abnormal program termination*/
     }
}
```

/* Spawn a child process running a new program. program is the name of the program to run; arg_list is a NULL-terminated list of character strings to be passed as the program's argument list. Returns the process ID of the spawned process. */

Example (Using exec to run a different program): lecture_7_code/fork_and_exec/fork_and_exec.c

```
int main ()
{
/* The argument list to pass is the "ls -l /". */
        char* arg_list[] = {
        "ls",           /* argv[0], the name of the program. */
        "-l",           /*long format*/
        "/",            /*root directory*/
        NULL /* The argument list must end with a NULL. */
};
        printf("print in main process before calling spawn.");
/* Spawn a child process running the "ls" command. */
        spawn ("ls", arg_list);                 /*The new process runs "ls" command*/
        printf ("done with main program\n");
        return 0;
}
```

Terminal

$ ./main

Process Termination

- return, exit()
  - Perform cleanup operations (fclose on stdio streams)
  - Call exit handlers (register with atexit(…))
    - atexit() registers the function to run at process termination
    - If multiple handlers are registered, called in reverse order of registration

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

Return 0 if successfully

- Other termination methods
  - _exit()
    - Return to kernel immediately
  - abort()
    - Abnormal program termination

Example (atexit example): lecture_7_code/atexit_example/atexit_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.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);
}
```

Example (Atexit example): lecture_7_code/atexit_example/atexit_example.c

```
static void my_exit1(void)
{
        printf("first exit handler\n");
}
static void my_exit2(void)
{
        printf("second exit handler\n");
}
```

Terminal
$ ./main

Output?

Output

- %./atexit_example

```
main is done
first exit handler
first exit handler
second exit handler
```

wait / waitpid

- System calls used in the parent process
  - Wait for a child process to finish executing
  - Retrieve information about its child's termination
  - Clean up resources used by the child process
  - Return child process ID

- *wait* pid_t wait(int *status);
  - Blocking
  - Wait for any child process

- *waitpid* pid_t waitpid(pid_t pid, int *status, int options);
  - Wait for a specific child
  - Can be nonblocking (when use with other functions)

- Retrieve termination status of child process
  - Macros available to distinguish between normal and abnormal termination
    - E.g., WIFEXITED(status)
    - Returns True if status was returned for a normally terminated child
  - Macros defined in the header file <sys/wait.h>

Example (Wait example): lecture_7_code/wait_example/wait_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
/* Use wait to the previous fork and exec example*/
int spawn(char* program, char** arg_list)
{
    pid_t child_pid;
    /* Duplicate this process. */
    child_pid = fork();
    if (child_pid != 0)
        /* This is the parent process. */
        return child_pid;
    else {
        /* Now execute PROGRAM, searching for it in the path. */
        execvp(program, arg_list);

        /* The execvp function returns only if an error occurs. */
        fprintf(stderr, "an error occurred in execvp\n");
        abort();
    }
}
```

Example (Wait example): lecture_7_code/wait_example/wait_example.c

```
int main()
{
   int child_status;
   /* The argument list to pass is the "ls –l /". */
   char* arg_list[] = {
      "ls", /* argv[0], the name of the program. */
      "-l",
      "/",
      NULL /* The argument list must end with a NULL. */
   };
   printf("print in main process before calling spawn.");
   spawn("ls", arg_list);                    /* Spawn a child process running the "ls" command. */

   wait(&child_status);                /*wait for a child process to complete. */
   if(WIFEXITED(child_status))   /* WIFEXITED returns a nonzero value if child terminates normally*/
      printf("the child process exited normally, with exit code %d\n",WEXITSTATUS(child_status));
   else
      printf("the child process exited abnormally\n");

printf ("done with main program\n");
   return 0;
}
```

Terminal

$ ./main

Note: WEXITSTATUS macro returns the exit code specified by the child

Exit Code

- Every command returns an exit status (sometimes referred to as a return status or exit code).
- A successful command returns a 0
- Anything other than a 0 status is undesirable

exit code

`1` - Catchall for general errors

`2` - Misuse of shell builtins (according to Bash documentation)

`126` - Command invoked cannot execute

`127` - "command not found"

`128` - Invalid argument to exit

`128+n` - Fatal error signal "n"

`130` - Script terminated by Control-C

`255\*` - Exit status out of range

Zombies

- A zombie process is a process that has terminated but has not been cleaned up yet
- It is the responsibility of the parent process to clean up its zombie children
- When the parent process calls wait, the zombie child's termination status is extracted and the child process is deleted
- If not being cleaned, the zombie process stays around in the system

Example (Making a zombie process): lecture_7_code/zombie_process/zombie_process.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
pid_t child_pid;
/* Create a child process. */
child_pid = fork ();
if (child_pid > 0) {
    sleep (60);          /* This is the parent process. Sleep for a minute. */
}
else {
/* This is the child process. Exit immediately. */
     exit (0);
}
return 0;
}
```

Terminal 1

$ ./main

Terminal 2

$ ps –a –o pid,tty,stat,cmd

While the program is still running, list the processes in another window:

Example (Making a zombie process)

- Output:

```
  PID TT       STAT CMD
 3755 pts/17   S+   ./main
 3756 pts/17   Z+   [main] <defunct>
 3757 pts/18   R+   ps -a -o pid,tty,stat,cmd
```

- The child process is marked as <defunct> and the status “Z” means “zombie”

Process status code examples

R Running or runnable (on run queue)

S Interruptible sleep (waiting for an event to complete)

Z Defunct ("zombie") process, terminated but not cleaned up by its parent.

\+ is in the foreground process group

How can we pass information to a new process?

- Command Line Arguments
  - The process that can pass command-line arguments to the new program

Command Line Arguments – Output?

Example (Echo all command-line arguments to standard output): lecture_7_code/echoarg/echoarg.c

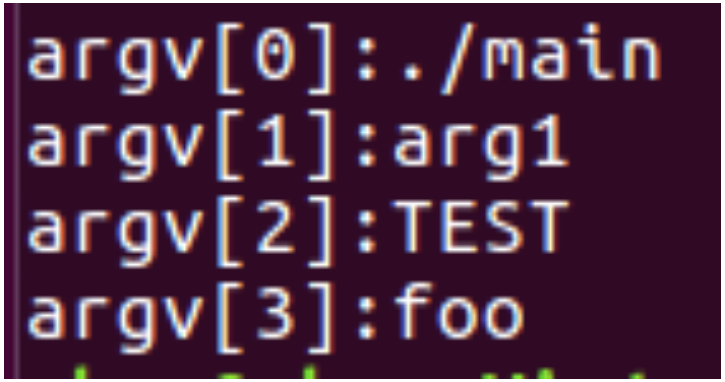
```
#include <stdio.h>

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);
}
```

Terminal

$ ./main arg1 TEST foo

Output

- $ ./main arg1 TEST foo



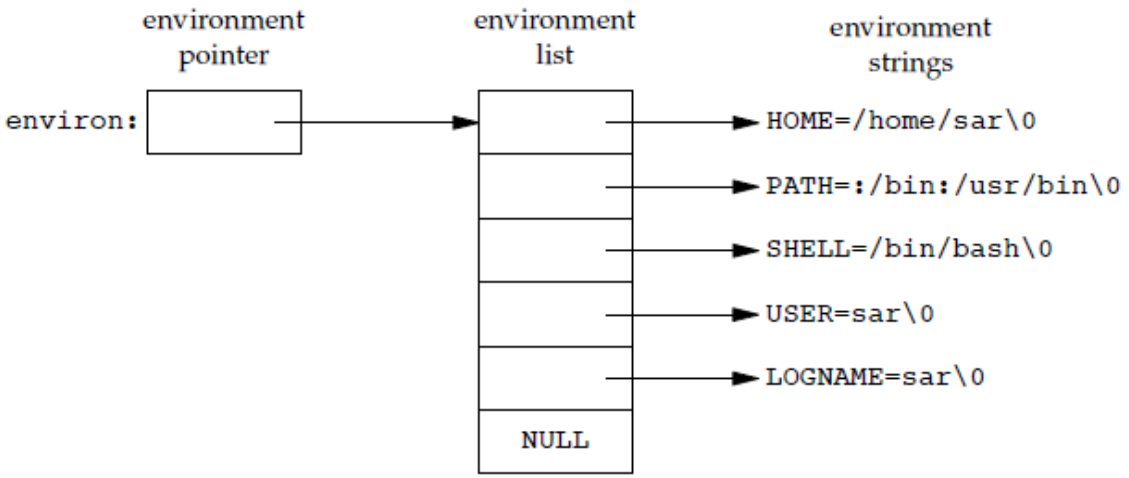
argv[0] stores the name of the program

How can we pass information to a new process?

- Environment list

Environment List

- An array of character pointers
- Each pointer contains the address of a null-terminated configuration string
- An example of the environment consisting of five strings
  - HOME contains the path to your home directory.
  - PATH contains a colon-separated list of directories through which Linux searches for commands you invoke.
  - SHELL The name of the user's default shell
  - LOGNAME the name of the user
  - USER contains your username, same purpose as LOGNAME



Environment List

- Accessible through global pointer variable
  - extern char **environ;
  - char ** : NULL-terminated array of pointers to character strings
- Or more easily be accessed through:
  - **getenv** function in <stdlib.h>

    char* getenv(const char*)

    Example: char* home = getenv (“HOME”);

Example (Print the execution environment): lecture_7_code/printenv/printenv.c

```
#include <stdio.h>
/* The ENVIRON variable contains the environment. */
extern char** environ;
int main ()
{
      char** var;
      for (var = environ; *var != NULL; ++var)
      printf ("%s\n", *var);
      return 0;
}
```

Terminal

```
$ ./main
```

Memory Layout of a C Program

One C program consists of the following pieces:

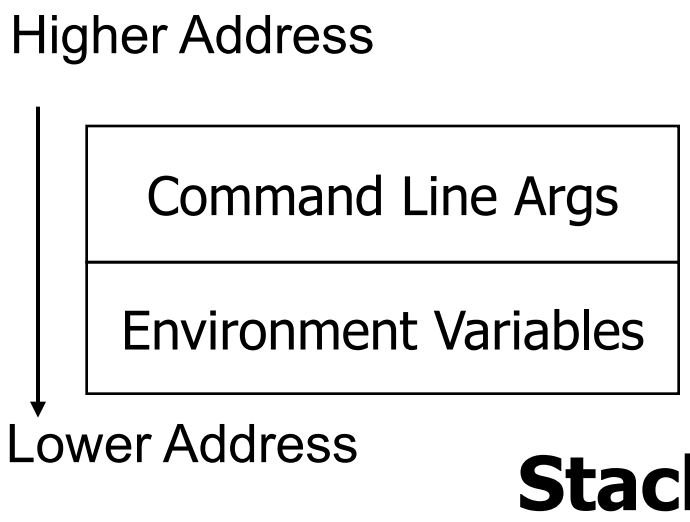
- Text segment
  - Program Code
- Initialized data segment
  - Global variables
- Uninitialized data segment
  - (Initialized to 0 by kernel)
- Stack
  - Automatic variables (local variables)
  - The information saved when a function is called (e.g., return address)
- Heap
  - Dynamic memory allocation

40

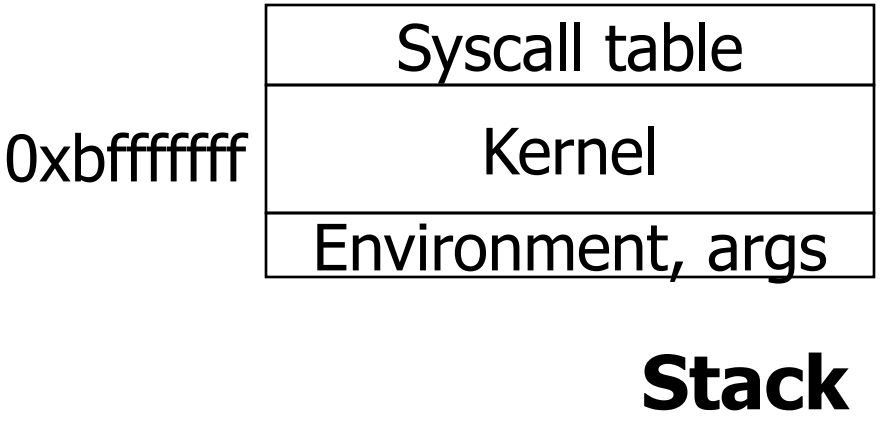
Memory Layout of a C Program

- **Stack segment:** memory area used by the process to store the local variables of function and other information that is saved every time a function is called, such as return address.

Logical arrangement



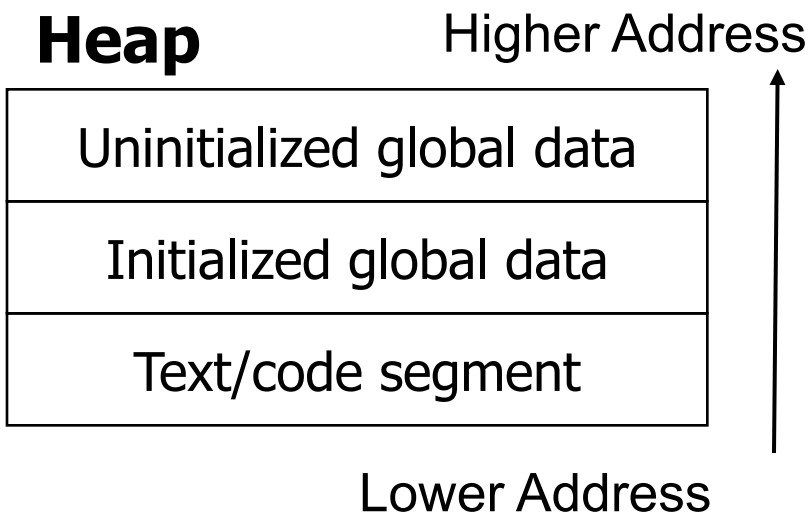
Memory Layout example in Linux system



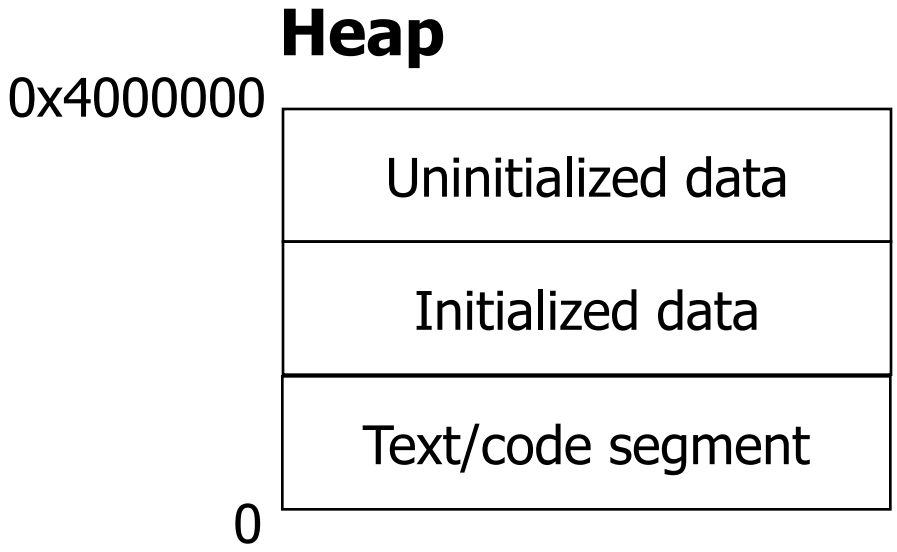
Memory Layout of a C Program

- **Heap segment** is used for dynamic memory allocation and shared among all the processes running in the system

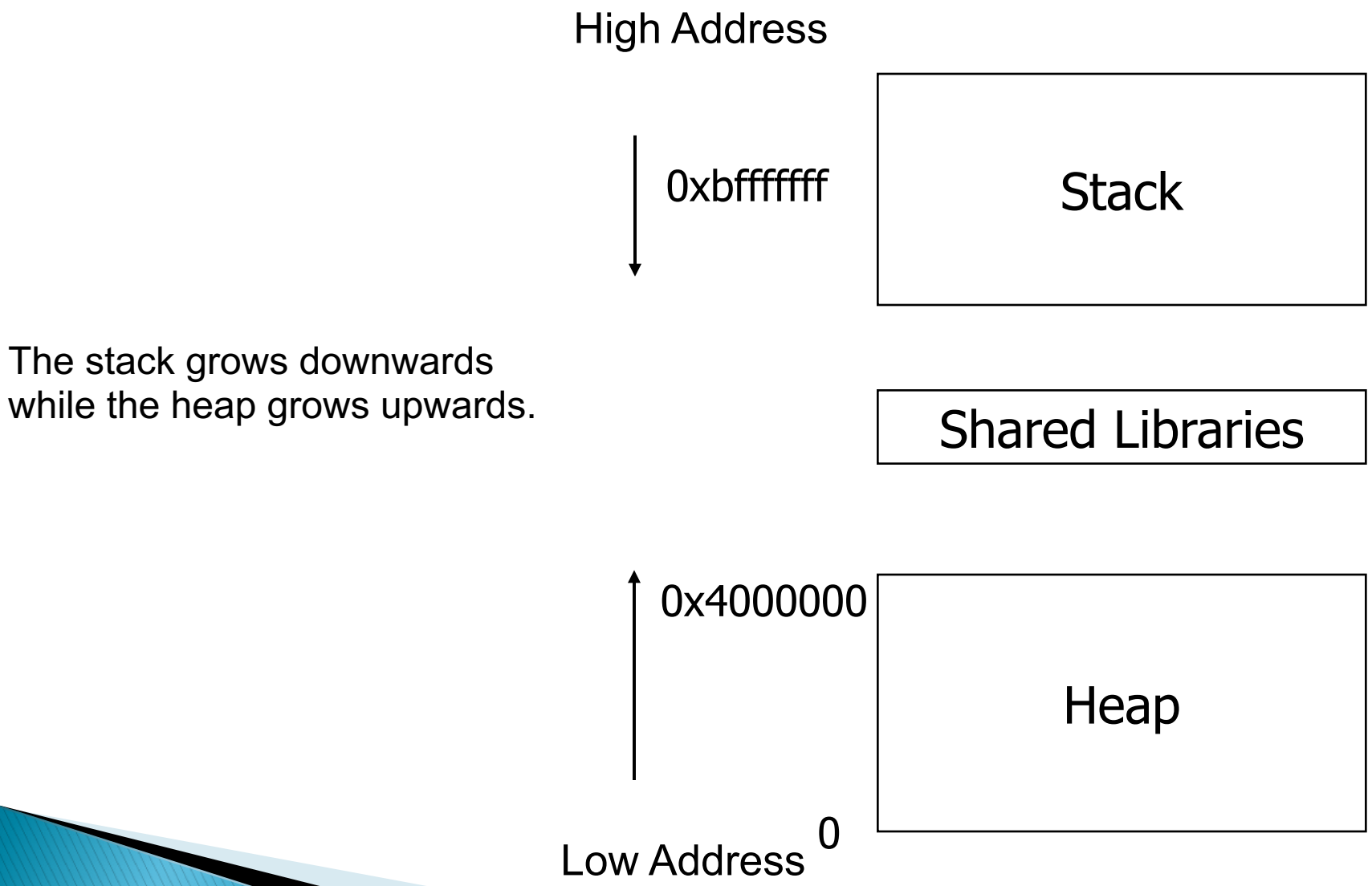
Logical arrangement



Memory Layout example in Linux system



Memory Layout of a C Program



Example (Print variable order): lecture_7_code/printorder/main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int glob = 6;                       /* external variable in initialized data */
char buf[] = "write to stdout\n";
void err_sys(const char* x);

/* This program demonstrates the fork function, showing how changes to variables in a
child process do not affect the value of the variables in the parent process*/

int main(void)
{
 int var;                           /* automatic variable on the stack */
 pid_t pid;
 var = 88;
```

Example (Print variable order): lecture_7_code/printorder/main.c

```
if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)   //write function is not buffered
  err_sys("write error");
printf("before fork\n");              /* we don't flush stdout */

 if ( (pid = fork()) < 0)
    err_sys("fork error");
 else if (pid == 0) {                 /* child */
                printf("the child process starts\n");
                glob++;               /* modify variables */
                var++;
        } else {
                sleep(2);                           /* parent */
                printf("this is the parent process\n");
        }
 printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
 exit(0);
}

void err_sys(const char* x)
{
    perror(x);
    exit(1);
}
```

Output?

Terminal
$ ./main

Output?

- $ ./main

```
write to stdout
before fork
the child process starts
pid = 430, glob = 7, var = 89
this is the parent process
pid = 429, glob = 6, var = 88
```

- Last two lines may be reversed in some cases (there is no rule whether parent or child will execute first)

- Is the output the same if we redirect to a file?

Output redirected to a file?

- $ ./main > out.txt

```
write to stdout
before fork
the child process starts
pid = 430, glob = 7, var = 89
before fork
this is the parent process
pid = 429, glob = 6, var = 88
```

- Why?

Why?

- write function is not buffered (data written once)
- Stdout by default is line buffered when connected to terminal
  - End of line character flushes the buffer (e.g., "\n")
- Stdout is fully buffered otherwise (e.g., directed to a file)
  - Child process inherits a copy of the buffer contents

Output redirected to a file

```
write to stdout
before fork
the child process starts
pid = 430, glob = 7, var = 89
before fork
this is the parent process
pid = 429, glob = 6, var = 88
```

Race Conditions

- When developing systems that involve multiple processes, we need to prevent race conditions
  - A **race hazard** (or **race condition**) is a flaw in a system or process where the output exhibits unexpected critical dependence on the relative timing of events.
  - For example, access shared memory at the same time or overwrite information before it is read.

Example (Race condition): lecture_7_code/race_condition/race_condition.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
/*a function of printing a single character of the string at a time*/
static void charatatime(char *str);
void err_sys(const char* x);

Int main(void)
{
    pid_t pid;
    if ( (pid = fork()) < 0)
            err_sys("fork error");
    else if (pid == 0) {
            charatatime("Child process wants to print a long string.\n");
    } else {
            charatatime(" Parent process is doing something else.\n");
    }
    exit(0);
}
```

Example (Race condition): lecture_7_code/race_condition/race_condition.c

```
static void charatatime(char *str)
{
    char  *ptr;
    int    c;

    setbuf(stdout, NULL);                    /* set unbuffered */
    for (ptr = str; c = *ptr++; )
            putc(c, stdout);
}

void err_sys(const char* x)
{
perror(x);
exit(1);
}
```

Terminal

$ ./main

Output:

```
Parent process is doingC hsiolmde tphrioncge sesl swea.n
ts to print a long string.
```

File Locking

File locks (“Record locks”)

- What happens when two process attempt to edit the same file at the same time ?
  - Each process independently maintains context information (such as write location), there may be coverage, crossover, and disordered content.
- File locking is the ability of a process to prevent other processes from modifying or reading a region of a file while the first process is working on it.

File locks (“Record locks”)

- Here are two types of locks :
  - Read lock (shared lock)
    - Any number of processes can have a shared read lock on a given region of a file.
    - All of them can read from the file but can’t write to it.
  - Write lock (exclusive lock)
    - Only one process can have an exclusive lock on a given region of a file.
    - The other process are prevented from getting locks or reading and writing.

File locks ("Record locks")

- Synchronize access from multiple processes to a shared file
- File locks provide fine-grained mechanisms:
  - Read and write locks: Multiple simultaneous reads are possible / Any write operation must be exclusive
  - Locking on byte ranges instead of entire file
- Use *fcntl()* with locking commands and lock data structure to acquire locks

Lock compatibility chart

	Read	Write
Read	Allowed	--
Write	--	--

fcntl()

- `fcntl()` function can change the properties of a file that is already open.
  - The `fcntl()` can be used for file lock purposes :

cmd	description
F_GETLK,F_SETLK, F_SETLKW	Get/set/lock file lock, it is discussed later.

```
#include <fcntl.h>
        int fcntl (int filedes, int cmd, struct flock *flockptr);
```

fcntl()

- For record locking ,the third argument of `fcntl()` is a pointer to a flock structure :

```
struct flock {
        short l_type;       /* F_RDLCK, F_WRLCK, F_UNLCK */
        short l_whence;     /* SEEK_SET, SEEK_CUR, SEEK_END */
        off_t l_start;      /* offset in bytes, relative to l_whence */
        off_t l_len;        /* length, in bytes ; 0 means lock to EOF */
        pid_t l_pid;        /* returned with F_GETLK */
}
```

l_pid field is the process ID of the process holding the lock. It is filled in by calling *fcntl()* with the F_GETLK command, but is ignored when making a lock.

fcntl()

- Three commands of `fcntl(record lock)` :
  - F_GETLK
    - To check the lock described by *flockptr* :
      - The information pointed by *flockptr* is overwritten by an existing lock .
      - If no lock exists, *flockptr.l_type* = F_UNLCK
  - F_SETLK
    - Set the lock described by *flockptr*.
    - If lock is not possible, `fcntl()` will return errono = EAGAIN or EACCES.
  - F_SETLKW
    - A blocking version of F_SETLK. It will wait until the desired region released.

Example (Lock and read a file): lecture_8_code/lock_unlock/lock_unlock.c

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

/*The following example demonstrates how to place a lock on 10 bytes from the offset 100
in a file and remove it afterwards*/

int main(int argc, char *argv[])
{
    int fd;
    struct flock fl;
    char buf[20];
    printf ("Opening file.\n");                                          W+
    fd = open("testfile.txt", O_RDWR); // open a file for reading and writing.
    if (fd == -1){
        perror("File open failed");
        exit(EXIT_FAILURE);
    }
```

Example (Lock and read a file): lecture_8_code/lock_unlock/lock_unlock.c

```
   /* Make a non-blocking request to place a write lock on 10 bytes from the offset 100 in a
file */
    fl.l_type = F_WRLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = 100;
    fl.l_len = 10;

    printf ("Locking the file.\n");
    if (fcntl(fd, F_SETLK, &fl) == -1) { //set non-blocking block
        if (errno == EACCES || errno == EAGAIN) {
            printf("Already locked by another process\n");
            /* We can't get the lock at the moment */
        } else {
            perror("Handle unexpected error");
        }
    } else {
        printf ("Lock was granted.\n");
```

Example (Lock and read a file): lecture_8_code/lock_unlock/lock_unlock.c

```
/* Perform I/O on bytes 100 to 109 of file */
lseek(fd, 100, SEEK_SET); /* seek to offset 100 from the start */
read(fd, buf, 10); /* read from 100 to 109 bytes */
buf[10] = '\0';//terminates the 10-byte buf string
printf("The 100-109 bytes in the file are: %s\n", buf);
/*can also perform write process in the file*/

/* Unlock the locked bytes */
fl.l_type = F_UNLCK;
fl.l_whence = SEEK_SET;
fl.l_start = 100;
fl.l_len = 10;
printf ("Unlocking the file.\n");
if (fcntl(fd, F_SETLK, &fl) == -1){ //unlock the file
    perror("Handle unexpected error");
}else{
    printf ("Unlock was granted.\n");
}
    }
    exit(EXIT_SUCCESS);
}
```

Terminal:
$./main

Example (Wait lock and write): lecture_8_code/lockfile/lockfile.c

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
/*The program in Listing opens a file for writing, and then places a write lock on it. The program waits for the user to hit Enter and then unlocks and closes the file.*/
int main (int argc, char* argv[]){
 char* file = "testfile.txt";
 char* buffer = "The additional text we want to write to the file. ";
 int fd;
 struct flock lock;

 printf ("opening %s\n", file);
 /* Open a file descriptor to the file. */
 fd = open (file, O_WRONLY);
```

Example (Wait lock and write): lecture_8_code/lockfile/lockfile.c

```
/* Initialize the flock structure. */
 memset (&lock, 0, sizeof(lock));
 lock.l_type = F_WRLCK;

 /* Place a write lock on the whole file. */
 printf ("Place a blocking lock on the file\n");
 fcntl (fd, F_SETLKW, &lock);

 printf ("Write to the file\n");
 lseek(fd, 0, SEEK_END); /* seek to the end of the file */
 write(fd, buffer, strlen(buffer)); /* write the buffer to the end */
 printf ("Finish writing; Hit Enter to unlock the file\n");

 /* Wait for the user to hit Enter. */
 getchar ();
 printf ("Enter received, unlock the file now\n");
```

Example (Wait lock and write): lecture_8_code/lockfile/lockfile.c

```
 /* Release the lock. */
  lock.l_type = F_UNLCK;
  fcntl (fd, F_SETLKW, &lock);
  printf ("file is unlocked\n");
  close (fd);
  return 0;
}
```

1. Terminal 1:
$./main
opening testfile.txt
Place a blocking lock on the file
Write to the file
Finish writing; Hit Enter to unlock the file

2. Terminal 2:
$./main
opening testfile.txt
Place a blocking lock on the file

3. Terminal 1:
hit Enter to unlock

4. Terminal 2:
hit Enter to unlock

Homework Readings

- Advanced Linux Programming
  - Ch 2.1.1
  - Ch 2.1.6
  - Ch 3 (excluding 3.3)
- Advanced Programming in Unix Environment
  - Ch 7.1 – 7.6

Optional Homework

- One problem in each homework set
- Help to boost the points lost in the programming assignments
- Counted as 5 points in each homework