CSE 344 System Programming

Week 2

System Programming Concepts :

- *System Calls, Library Functions, Standard C library...*

Processes :

- *Processes and Programs*
- *Memory Layout of a process*
- *Command line arguments, Environment list ...*

Memory Allocation :

- *Allocating memory on the Heap*
- *Allocating memory on the Stack*

System Calls

A *system call* is a controlled entry point into the kernel, allowing a process to request, the kernel to perform some action on the process's behalf.

The kernel makes a range of services accessible to programs via the system call application programming interface.

These services include, creating a new process, performing I/O, creating a pipe for interprocess communication and so on.

System Calls

A system call changes the processor state from *user mode* to *kernel mode*, so that the CPU can access protected kernel memory.

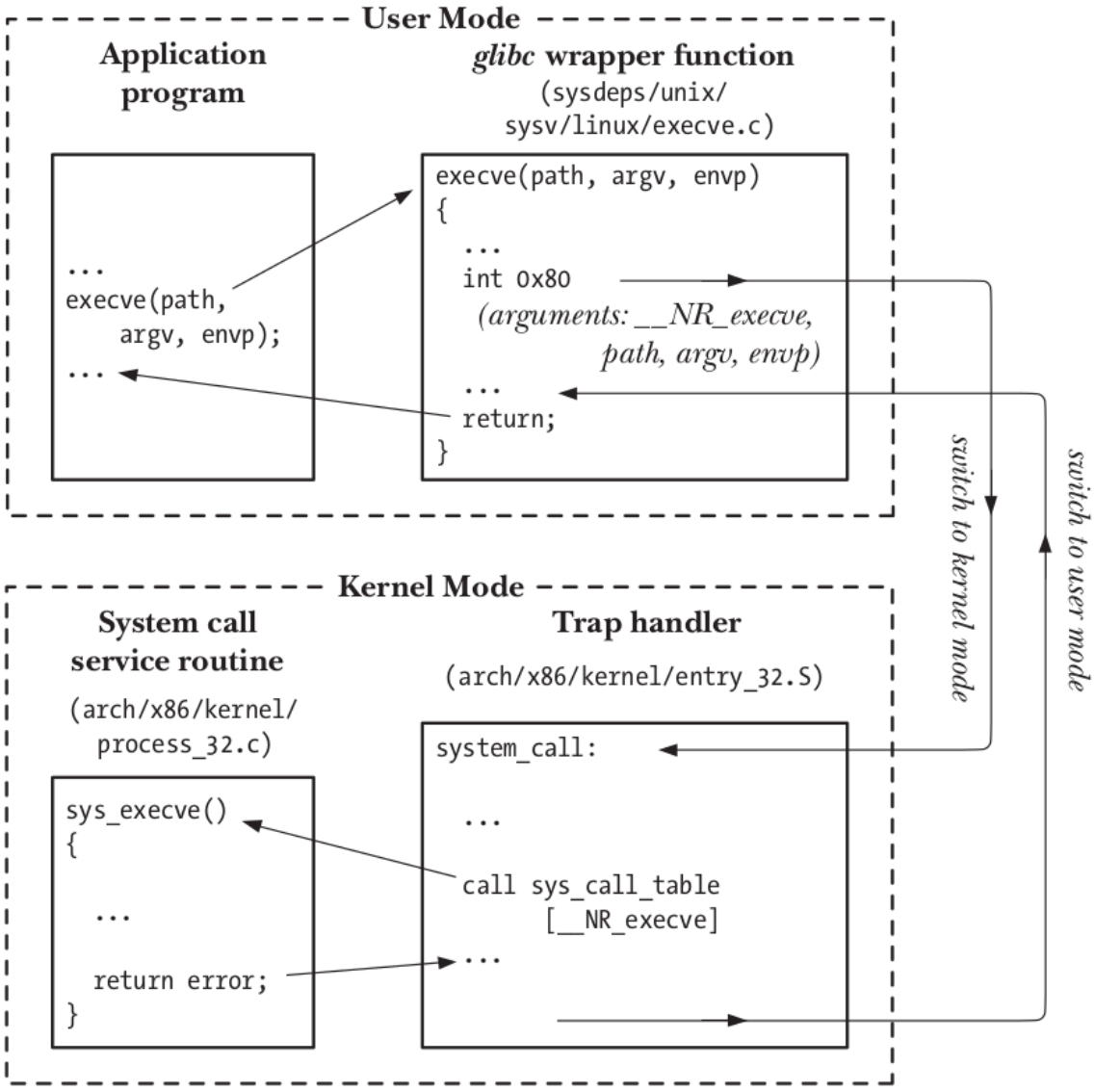
The set of system calls is fixed. Each system call is identified by a unique number, normally not visible to the programs.

Each system call may have a set of arguments that specify information to be transferred from user space to kernel space and vice-verse.

System Calls

From a programming point of view, invoking a system call looks much like calling a C function.

However, behind the scenes, many steps occur during the execution of a system call.



What is really going on ?

1. The application program makes a system call by invoking a wrapper function in the C library.

2. The wrapper function must make all of the system call arguments available to the system call *trap-handling routine*. These arguments are passed to the wrapper via the stack, but the kernel expects them in specific registers. The *wrapper function* copies the arguments to these registers.

3. Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number into a specific *CPU register* .

4.The wrapper function executes a trap machine instruction (int 0x80 in our example), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location 0x80 (128 decimal) of the system's trap vector.

What is really going on ?

5. In response to the trap to location 0x80 , the kernel invokes its *system_call()* routine to handle the trap. This handler:

a) Saves register values onto the kernel stack

b) Checks the validity of the system call number

c) Invokes the appropriate system call service routine, which is found by using the system call number to index a table of all system call service routines

d) Restores register values from the kernel stack and places the system call return value on the stack

e) Returns to the wrapper function, simultaneously returning the processor to user mode.

What is really going on ?

6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable *errno* using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

Library Functions

- A *library function* is simply one of the multitude of functions that constitutes a programming language (standard C in our case). The purposes of these functions are very diverse (like opening a file, converting a time to a human-readable format, and comparing two character strings).
- Many library functions don't make any use of system calls. However, some library functions are layered on top of system calls
- Often, library functions are designed to provide a more caller-friendly interface than the underlying system call.

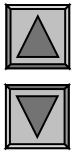
Standard C library

- There are different implementations of the standard C library on the various UNIX implementations. The most commonly used implementation on Linux is the GNU C library (*glibc*)

Error Handling

- Nearly all system call and library function returns some type of status value indicating whether the call succeeded or failed. This status value should always be checked to see whether the call succeeded. If it did not, then appropriate action should be taken—at the very least, the program should display an error message warning that something unexpected occurred.
- Many hours of debugging time can be wasted due to a check not made on the status return of a system call or library function that “couldn’t possibly fail”
- The manual page for each system call documents the possible return values of the call, showing which value(s) indicate an error. Usually, an error is indicated by a return of –1.

```
...


fd = open(pathname, flags, mode);        /* system call to open a file */
if (fd == -1) {
/* Code to handle the error */
}
...

if (close(fd) == -1) {
/* Code to handle the error */
}


...
```



Code Segment for Error Handling

Error Handling (system calls)

- When a system call fails, it sets the global integer variable *errno* to a positive value that identifies the specific error. Including the <*errno.h*> header file provides a declaration of *errno*, as well as a set of constants for the various error numbers.

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
     if (errno == EINTR)
     fprintf(stderr, "read was interrupted by a signal\n");
else {
/* Some other error occurred */
}
}
```

- When checking for an error, one should always first check if the function return value indicates an error, and only then examine *errno* to determine the cause of the error.

Error Handling (library functions)

- A common course of action after a failed system call is to print an error message based on the *errno* value. The *perror()* and *strerror()* library functions are provided for this purpose.
- The *perror()* function prints the string pointed to by its *msg* argument, followed by a message corresponding to the current value of *errno*.

```
#include <stdio.h>

void perror(const char *msg );
```

- The *strerror()* function returns the error string corresponding to the error number given in its *errnum* argument.

```
#include <string.h>

char *strerror(int errnum );
```


Error Handling (library functions)

The various library functions return different data types and different values to indicate failure. For our purposes, library functions can be divided into the following categories

- Some library functions return error information in exactly the same way as system calls: a "–1" return value, with *errno* indicating the specific error. Errors from these functions can be diagnosed in the same way as errors from system calls.
- Some library functions return a value other than "–1" on error, but nevertheless set *errno* to indicate the specific error condition. The *perror()* and *strerror()* functions can be used to diagnose these errors.
- Other library functions don't use *errno* at all. The method for determining the existence and cause of errors depends on the particular function and is documented in the function's manual page. For these functions, it is a mistake to use *errno*, *perror()*, *or strerror()* to diagnose errors.

Processes and Programs

A *process* is an instance of an executing program.

A *program* is a file containing a range of information that describes how to construct a process at run time. This information includes:

- *Binary format identification:* Each program file includes meta information describing the format of the executable file. This enables the kernel to interpret the remaining information in the file.
- *Machine-language instructions:* These encode the algorithm of the program
- *Program entry-point address:* This identifies the location of the instruction at which execution of the program should commence
- *Data:* The program file contains values used to initialize variables and also literal constants used by the program (e.g., strings).

Processes

- *Symbol and relocation tables:* These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
- *Shared-library and dynamic-linking information:* The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should be used to load these libraries.
- *Other information:* The program file contains various other information that describes how to construct a process.

One program may be used to construct many processes, or, many processes may be running the same program.

Processes

- A process is an abstract entity, defined by the kernel, to which system resources are allocated in order to execute a program
- From the kernel's point of view, a process consists of user-space memory containing program code and variables used by that code, and a range of kernel data structures that maintain information about the state of the process
- The information recorded in the kernel data structures includes various identifier numbers (IDs) associated with the process, virtual memory tables, the table of open file descriptors, information relating to signal delivery and handling, process resource usages and limits, the current working directory, and a host of other information

Process ID and Parent Process ID

Each process has a process ID (PID), a positive integer that uniquely identifies the process on the system.

The *getpid()* system call returns the process ID of the calling process.

```
#include <uninst.h>

pid_t getpid(void);
                                        Always successfully returns process ID of caller
```

Each process has a parent—the process that created it. A process can find out the process ID of its parent using the *getppid()* system call.

```
#include <uninst.h>

pid_t getppid(void);
                                    Always successfully returns process ID of parent of caller
```


Process ID and Parent Process ID

The parent process ID attribute of each process represents the tree-like relationship of all processes on the system. The parent of each process has its own parent, and so on, going all the way back to process 1, *init* , the ancestor of all processes.

If a child process becomes orphaned because its “birth” parent terminates, then the child is adopted by the *init* process, and subsequent calls to *getppid()* in the child return 1

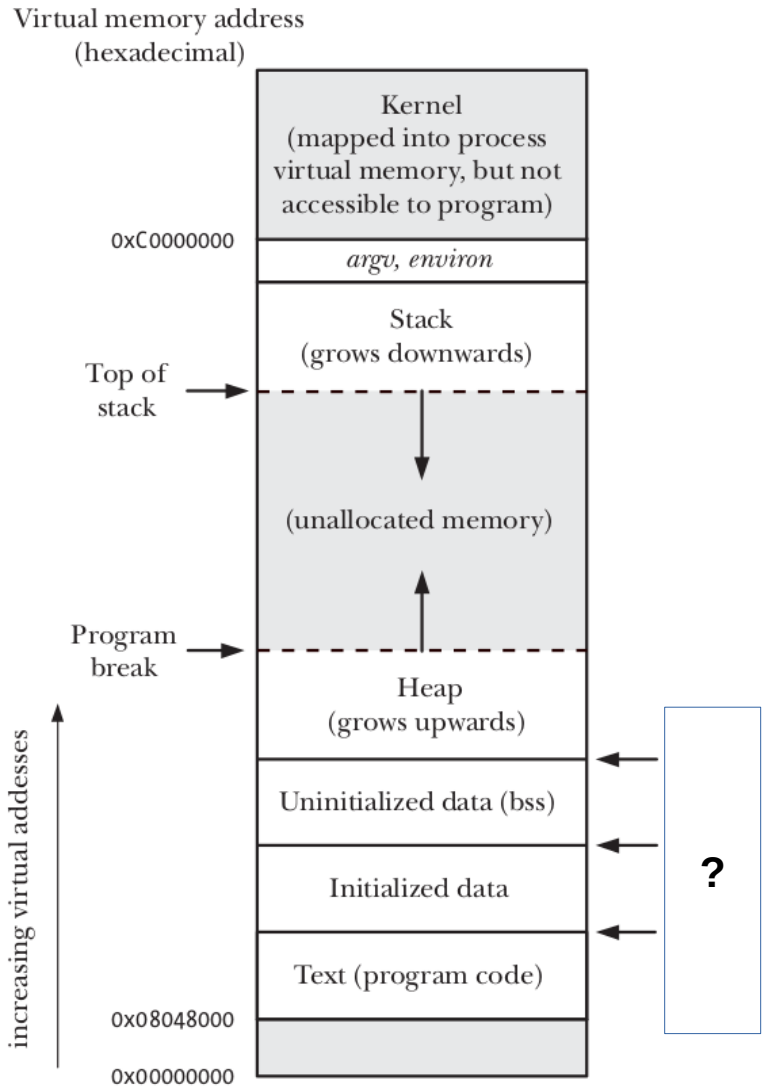
Memory Layout of a Process

The memory allocated to each process is composed of a number of parts, usually referred to as *segments*. These segments are as follows :

- The *text segment* contains the machine-language instructions of the program run by the process. The text segment is made read-only so that a process doesn't accidentally modify its own instructions via a bad pointer value.
- The *initialized data segment* contains global and static variables that are explicitly initialized.
- The *uninitialized data segment* contains global and static variables that are not explicitly initialized
- The *stack* is a dynamically growing and shrinking segment containing stack frames.
- The *heap* is an area from which memory (for variables) can be dynamically allocated at run time.

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

char globBuf[65536];                    /* Uninitialized data segment */
int primes[] = { 2, 3, 5, 7 };          /* Initialized data segment */

static int square(int x)                /* Allocated in frame for square() */
{
int result;                             /* Allocated in frame for square() */
result = x * x;
return result;                          /* Return value passed via register */
}

static void doCalc(int val)             /* Allocated in frame for doCalc() */
{
  printf("The square of %d is %d\n", val, square(val));
  if (val < 1000) {
     int t;                             /* Allocated in frame for doCalc() */
     t = val * val * val;
     printf("The cube of %d is %d\n", val, t);
  }
}

int main(int argc, char *argv[])        /* Allocated in frame for main() */
{
   static int key = 9973;                   /* Initialized data segment */
   static char mbuf[10240000];              /* Uninitialized data segment */
   char *p;                                 /* Allocated in frame for main() */
   p = malloc(1024);                        /* Points to memory in heap segment */
  doCalc(key);
exit(EXIT_SUCCESS);
}
```

Typical memory layout of a process (x86)



This part is left to the students

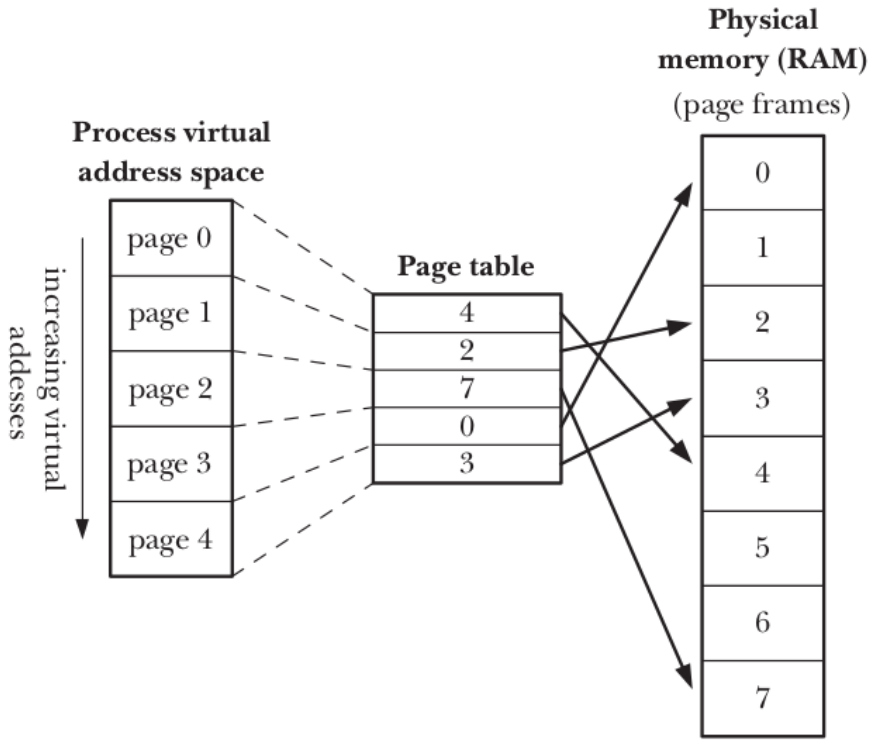
Virtual Memory Management

- Modern Operating systems employ a technique known as *virtual memory management*. The aim is to make efficient use of both the CPU and RAM (physical memory) by exploiting a property that is typical of most programs: *locality of reference*
- Most programs demonstrate two kinds of locality:
  - *Spatial locality* is the tendency of a program to reference memory addresses that are near those that were recently accessed
  - *Temporal locality* is the tendency of a program to access the same memory addresses in the near future that it accessed in the recent past

By using locality of reference, it is possible to execute a program while maintaining only part of its address space in physical memory (RAM)

Virtual Memory Management

- The kernel maintains a *page table* for each process. The page table describes the location of each page in the process's virtual address space (the set of all virtual memory pages available to the process).
- Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it currently resides on disk.



Virtual Memory Management

- Not all address ranges in the process's virtual address space require page-table entries. Typically, large ranges of the potential virtual address space are unused, so that it isn't necessary to maintain corresponding page-table entries.
- If a process tries to access an address for which there is no corresponding page-table entry, it receives a SIGSEGV signal.
- A process's range of valid virtual addresses can change over its lifetime, as the kernel allocates and deallocates pages (and page-table entries) for the process.
- Virtual memory management separates the virtual address space of a process from the physical address space of RAM.

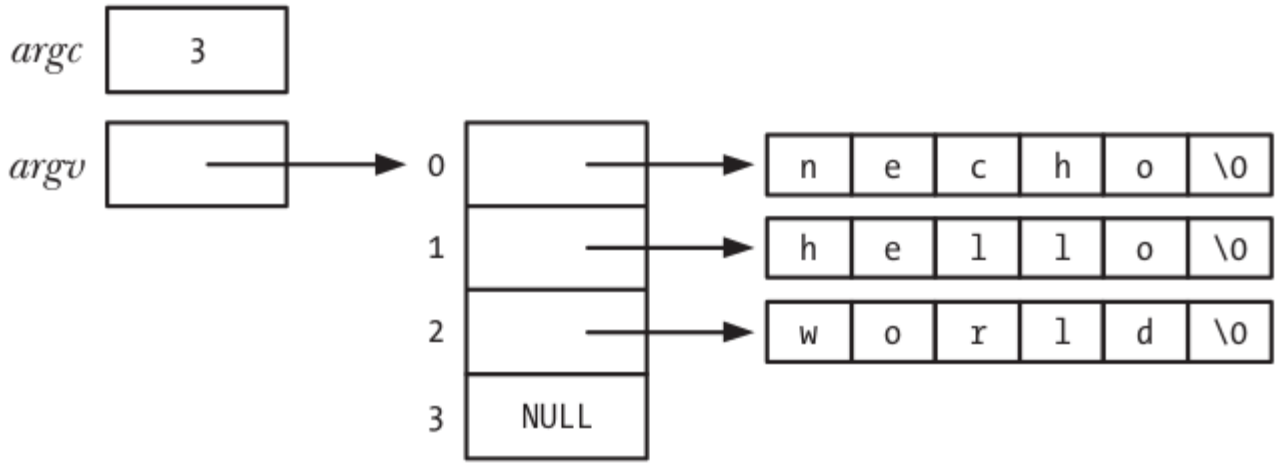
The Stack and Stack Frames

- The stack grows and shrinks linearly as functions are called and return. On most other UNIX implementations, the stack resides at the high end of memory and grows downward (toward the heap).
- A special-purpose register, the *stack pointer*, tracks the current top of the stack. Each time a function is called, an additional frame is allocated on the stack, and this frame is removed when the function returns.
- Sometimes, the term *user stack* is used to distinguish the stack we describe here from the *kernel stack*.
- The kernel stack is a per-process memory region maintained in kernel memory that is used as the stack for execution of the functions called internally during the execution of a system call

Command-Line Arguments (*argc , argv*)

- Every C program must have a function called *main()*, which is the point where execution of the program starts. When the program is executed, the command-line arguments (the separate words parsed by the shell) are made available via two arguments to the function *main()* , *argc* and *argv*
- The first argument, *int argc*, indicates how many command-line arguments there are.
- The second argument, *char *argv[]*, is an array of pointers to the command-line arguments, each of which is a null-terminated character string.
- The first of these strings, in *argv[0]*, is the name of the program itself. The list of pointers in argv is terminated by a *NULL* pointer (i.e., *argv[argc]* is *NULL*)

```
#include "tlpi_hdr.h"

int main(int argc, char *argv[])
{
int j;
     for (j = 0; j < argc; j++)
          printf("argv[%d] = %s\n", j, argv[j]);
exit(EXIT_SUCCESS);
}
                                                        /* necho.c */
```



Alternatively, since *argv* list is terminated by NULL value the body of the *main ()* above can be implemented as

```
char **p;

for (p=argv; *p !=NULL; p++)
puts(*p);
```

Environmet List

- Each process has an associated array of strings called the *environment list* (simply the *environment*). Each of these strings is a definition of the form *name=value*. Thus, the environment represents a set of name-value pairs that can be used to hold arbitrary information. The names in the list are referred to as *environment variables*.
- When a new process is created, it inherits a copy of its parent's environment. Since the child gets a copy of its parent's environment at the time it is created, this transfer of information is one-way and once-only. After the child process has been created, either process may change its own environment, and these changes are not seen by the other process.

Environmet List

- Within a C program, the environment list can be accessed using the global variable *char **environ* (a NULL-terminated list of pointers).

```
#include "tlpi_hdr.h"

extern char **environ;

int main(int argc, char *argv[])
{
     char **ep;
     for (ep = environ; *ep != NULL; ep++)
          puts(*ep);
exit(EXIT_SUCCESS);
}                                                    /* display_env.c */
```


Environmet List

In order to manipulate the environment in a C program you may also use

```
#include <stdlib.h>

char *getenv(const char * name );
int putenv(char * string );
int setenv(const char * name , const char * value , int overwrite );
int unsetenv(const char * name );
int clearenv(void)  ;
```


Memory Allocation

Allocating Memory on the Heap

- A process can allocate memory by increasing the size of the heap, a variable size segment of contiguous virtual memory that begins just after the uninitialized data segment of a process and grows and shrinks as memory is allocated and freed.
- The current limit of the heap is referred to as the *program break*
- To allocate memory, C programs normally use the *malloc* family of functions.

Memory Allocation

Adjusting the Program Break: *brk()* **and** *sbrk()*

- Resizing the heap is just requesting the kernel to adjust its idea of where the process's program break is.
- Initially, the program break lies just past the end of the uninitialized data segment . After the program break is increased, the program may access any address in the newly allocated area, but no physical memory pages are allocated yet. The kernel automatically allocates new physical pages on the first attempt by the process to access addresses in those pages.
- Traditionally, the UNIX system has provided two system calls for manipulating the program break,

```
#include <unstd.h>

int brk(void * end_data_segment );
                                    /*   Returns 0 on success, or –1 on error */
void *sbrk(intptr_t increment );
               /* Returns previous program break on success, or (void ) –1 on error*/
```


Memory Allocation

Allocating Memory on the Heap: *malloc()* **and** *free()*

C programs use the *malloc* family of functions to allocate and deallocate memory on the heap. These functions offer several advantages over *brk()* and *sbrk()*. In particular, they:

- are standardized as part of the C language;
- are easier to use in threaded programs;
- provide a simple interface that allows memory to be allocated in small units;
- allow us to arbitrarily deallocate blocks of memory, which are maintained on a free list and recycled in future calls to allocate memory.

Memory Allocation

- The *malloc()* function allocates size bytes from the heap and returns a pointer to the start of the newly allocated block of memory. The allocated memory is “not initialized.

```
#include <stdlib.h>

void *malloc(size_t size);


                              /* Returns pointer to allocated memory on success*/
```

- If memory could not be allocated (perhaps because we reached the limit to which the program break could be raised), then *malloc()* returns NULL and sets errno to indicate the error.
- Although the possibility of failure in allocating memory is small, all calls to *malloc()*, and the related functions, should check for this error return.

Memory Allocation

- The *free()* function deallocates the block of memory pointed to by its *ptr* argument, which should be an address previously returned by *malloc()*

```
#include <stdlib.h>

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

- In general, *free()* doesn't lower the program break, but instead adds the block of memory to a list of free blocks that are recycled by future calls to *malloc()*.

To *free()* or not to *free()* ?

- When a process terminates, all of its memory is returned to the system, including heap memory allocated by functions in the *malloc* package. In programs that allocate memory and continue using it until program termination, it is common to omit calls to *free()*, relying on this behavior to automatically free the memory.
- This can be especially useful in programs that allocate many blocks of memory, since adding multiple calls to *free()* could be expensive in terms of CPU time, as well as perhaps being complicated to code.
- Although relying on process termination to automatically free memory is acceptable for many programs, there are many reasons why it can be desirable to explicitly free all allocated memory.

Other Methods of Allocating Memory on the Heap

- As well as *malloc()*, the C library provides a range of other functions for allocating memory on the heap, the C library provides a range of other functions for allocating memory on the heap

```
#include <stdlib.h>

void *calloc(size_t numitems , size_t size );
                    /*Returns pointer to allocated memory on success, or NULL on error*/
```

The *calloc()* function allocates memory for an array of identical items. The *numitems* argument specifies how many items to allocate, and size specifies their size. After allocating a block of memory of the appropriate size, *calloc()* returns a pointer to the start of the block.

Note that *calloc()* initializes the allocated memory to 0

Other Methods of Allocating Memory on the Heap

- The *realloc()* function is used to resize a block of memory previously allocated by one of the functions in the *malloc* package.

```
#include <stdlib.h>

void *realloc(void * ptr , size_t size );
                    /*Returns pointer to allocated memory on success, or NULL on error*/
```

The *ptr* argument is a pointer to the block of memory that is to be resized. The size argument specifies the desired new size of the block. On success, *realloc()* returns a pointer to the location of the resized block (This may be different from its location before the call). On error, *realloc()* returns NULL and leaves the block pointed to by *ptr* untouched

Allocating Memory on the Stack: *alloca()*

- Like the functions in the *malloc* package, *alloca()* allocates memory dynamically. However, instead of obtaining memory from the heap, *alloca()* obtains memory from the stack by increasing the size of the stack frame.

```
#include <alloca.h>

void *alloca(size_t size );

                                    /*Returns pointer to allocated block of memory */
```

The *size* argument specifies the number of bytes to allocate on the stack. The *alloca()* function returns a pointer to the allocated memory as its function result.

- We need not—indeed, must not—call *free()* to deallocate memory allocated with *alloca()*. Likewise, it is not possible to use *realloc()* to resize a block of memory allocated by *alloca()*. You cannot not use *alloca()* within a function argument list.

Allocating Memory on the Stack: *alloca()*

- Using *alloca()* to allocate memory has a few advantages over *malloc()*. One of these is that allocating blocks of memory is faster with *alloca()* than with *malloc()*, as *alloca()* is implemented by the compiler as inline code that directly adjusts the stack pointer.
- Furthermore, *alloca()* doesn't need to maintain a list of free blocks