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.

System call is mostly done on the hardware (CPU & memory).

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.

A system call changes the processor state from user mode (where data is stored, state of the process, registers, previous instructions) to kernel mode (necessary registers, how many processes this program is using, libraries, links 🡪 all of them in database in kernel), so that the CPU can access protected kernel memory.

User space is larger than kernel space.

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.

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.

Diagram, schematic

Description automatically generated

wrapper function will communicate with the kernel’s entries.

Command is related with corresponding number which is 0x80. Wrapper requests kernel to execute this 0x80 instruction.

Trap handler and wrapper will take all the necessary data from the user space of the process and put those into corresponding registers. Then pass 0x80 value to the actual system call service routine so that can be executed.

When kernel decides it is your process’s time to execute that instructions, that instruction is done on the specific service routine and output of it will be returned back to the trap function.

Necessary changes are made on registers in trap function. Those registers are copied into the user space database of the application. Return values also returned to the wrapper function.

Return value in wrapped function is returned back to the application.

***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.
5. In response to the trap to location 0x80 , the kernel invokes its system\_call() routine to handle the trap. This handler:
   1. Saves register values onto the kernel stack
   2. Checks the validity of the system call number
   3. 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
   4. Restores register values from the kernel stack (where they can be passed by trap back to wrapper – user space) and places the system call return value on the stack
   5. Returns to the wrapper function, simultaneously returning the processor to user mode.
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.
   1. Careful with using errno, since it is global variable, if we were to use more than one execution in our process.

Diagram

Description automatically generated

P1 is our process and it has more than one threads.

Kernel doesn’t know how many threads in p1.

While executing any of them, an error occurred. It is fed back to there.

Normal C routines are not thread safe, that means it doesn’t keep the value with error to the corresponding thread. Any of the executions might think that error happened while they are doing their execution.

When it is time for next execution, if it tries to use the errno that is caused by previous instruction.

Using standard C libraries in a multithread environment is tricky.

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). Some functions are not thread safe but glibc is very useful.

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.

Graphical user interface

Description automatically generated with low confidence

When error happens, most library return with -1. Corresponding error is mapped into global variable called errno.

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.

A picture containing logo

Description automatically generated

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.

Birçok sistem librarysi, özellikle Linux sistemlerinde, link olarak vardır. Herkes farklı distribution kullandığı için gerçek library ve headerın nerede olduğunu bulmakta zorlanabilirsin. Ancak link birçoğunda aynıdır. Librarynin nerede olduğunu bulmak istiyorsan “ld <program ismi> | <library ismi> ”

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.



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



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. For each program you write, when you execute it, it becomes a process. Kernel will give you user space and kernel space.

For each program that became a process, we have 2 data blocks: one in user space, one in kernel space.

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-noninitialized variables and also literal constants used by the program (e.g., strings).
* 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.

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

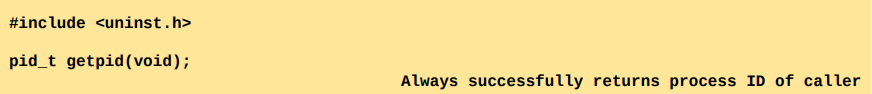
The information recorded in the kernel data structures includes various identifier numbers (IDs) associated with the process, virtual memory tables, the table of open file descriptors, information relating to signal delivery and handling, process resource usages and limits, the current working directory, and a host of other information

For each time you execute a program, another process is formed by the kernel for you with different ID of course.

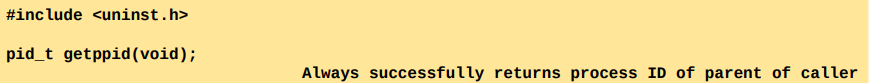
Process ID and Parent Process ID

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. This is a system call requesting kernel to return ID of the process that calls the getpid.



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.



pid\_t is an unsigned integer.

Sometimes your program, requires another process to do some extra work for you. Most of the time fork() enables you to create another process. Return of fork, we have 2 processes executing the same code that you have written now.

fork() returns ID of child to the parent process and 0 to the child process. So fork returns 2 things:

* TO PARENT ----> one who executed the fork() ----- ID of child that created
* TO CHILD ----> 0 – child will know that it is formed by another process by looking at this value

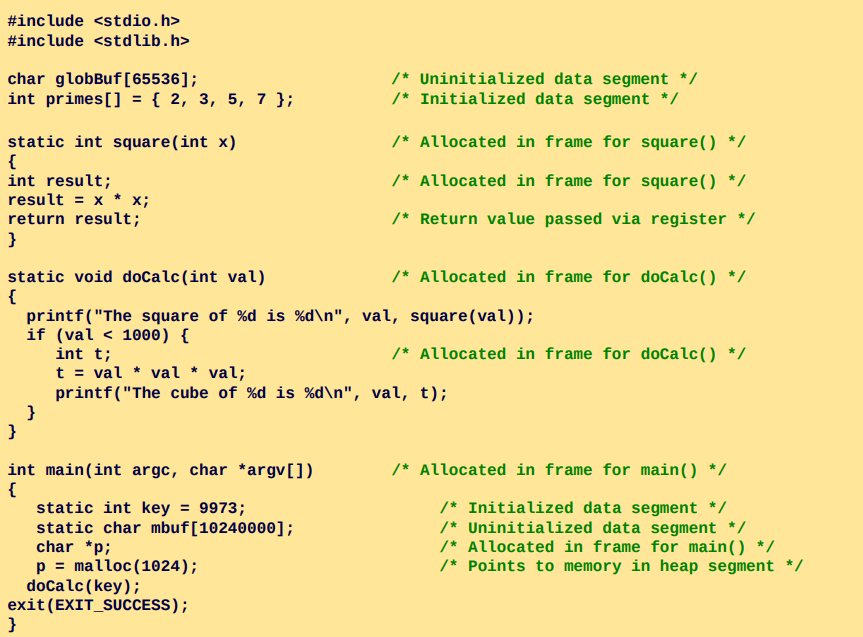
The parent process ID attribute of each process represents the tree-like relationship of all processes on the system. The parent of each process has its own parent, and so on, going all the way back to process 1, init, the ancestor of all processes.

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. Child is adopted by init so that when child do what is gotta do and return, it isn’t in a fix.

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.



allocated in stack

When we exit, all of the memory will be free out.

Typical memory layout of a process (x86)

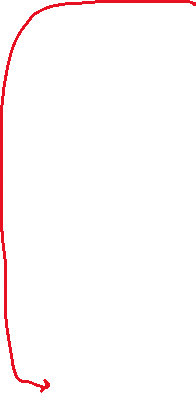
Diagram

Description automatically generated

execution code

input and environment variables required for your process

environ contains current working directory, which libraries it uses and stuff like that



Representation of user space database for each process in virtual memory. There is also kernel space (ID, last state of registers when your process left the kernel…).

Page table is here.

Kernel space:

* state of process
* which libraries you use
* which files you have opened
* signals, etc.

When you use malloc, if there is enough space in heap it will allocated to you. If not heap will grow.

Unitialized variables are moved to initialized segment after they are initialized most probably. These are handled by the OS, don’t worry about it.

Process 3 formda olabilir:

* oluşturulmuş, bekliyor
* oluşturulmuş, execute ediyor
* oluşturulmuş, işi bitti

Görev yöneticisinde gördüklerin process.

Çalıştırmadan önce programının bir ID’si yok. Programı birden fazla çalıştırırsan hepsine bir ID verilir ki OS hangisinin ne olduğunu anlayabilsin.

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)

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.

Diagram

Description automatically generated

For each process, kernel maintains a page table.

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.

* For example, stack grows too much or heap grows too much.

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.

Virtual memory management, kerneldaki database üzerinden sizin için ayrılmış pagein numarasını bulup oradan da memorydeki karşılığını bularak ilerliyor.

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 )

Table

Description automatically generated

Environment 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 onceonly. After the child process has been created, either process may change its own environment, and these changes are not seen by the other process.

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

Text

Description automatically generated with medium confidence

you don’t need this

It is kinda system dependent. On each system, the environment variables vary.

Kernel has to know your environment variables especially current working directory in order to execute your process properly.

Processin için global tanımlanmış environ variableını diğer processlerle, childınla falan paylaşabilirsin.

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

A picture containing chart

Description automatically generated

You can go upper directory and execute another process inside your process etc. In order to do that, C gives you getenv(), putenv(), setenv(), unsetenv(), clearenv()

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.

***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,

A picture containing logo

Description automatically generated

Not user friendly.

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.

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”.

A picture containing shape

Description automatically generated

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.

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



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.

init daha sonra sistemin sizin için ayırmış olduğu bölümü diğer processler için kullanır hale getirecek ama aradaki bölümde, processinin zombie olduğu bölümde kullanıma açık olması önemli (good programming practice).

Programın bir hatayla çıktıysa, sistemde oluşan bir hata sonucuyla çıktıysa, sen de o hatanın ne olduğunu araştırmak istiyorsan, orada free yapmayabilirsin. FREE() KULLANMAYACAĞIN TEK DURUM.

* Yani bir şekilde programın hata yaptı, malloc ettiğin bölgede onunla ilgili hataları tutuyorsun, o bölge senin için önemliyse 🡪 core dumped yap, sonra free yap.
* Böylece sonrasında hatanın ne olduğu, memorynin o anki durumu öğrenilebilir. Multithreaded için bu kısım zor.

YOU HAVE TO DO free()!!! When that portion is allocated and is not used anymore, stack collision might occur or other executions in your process will not be able to use that portion.

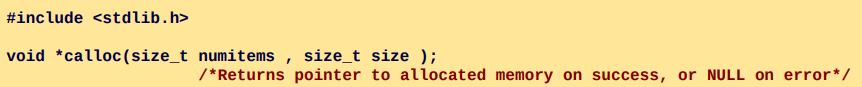
Özellikle fork, calloc kullanmaya başladıktan sonra, size ayrılan unallocated memory boyutunu artırma requestini sisteme göndermeyi öğrenirsin. Yani arkada sürekli çalışan daemon gibi, web server gibi programlarda free yapmak çok önemli.

init bir hatayla karşılaştı diyelim. Sistemin çalışabilmesi için initin yok olmaması gerekiyor. Mavi ekran sorununun engellenmesi için init kapanırken kendinin childını oluşturuyordu (Windows NT). Bir yerden sonra çok fazla child oluşturuyordu ve memory’yi yavaş yavaş dolduruyordu.

Memory leakin en büyük sebebi free edilmemiş malloc vs. kullanımı. Diğer sebep de execution oluşturup tam bitmeden ortadan kaybolman ve executionın ortada kalması.

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

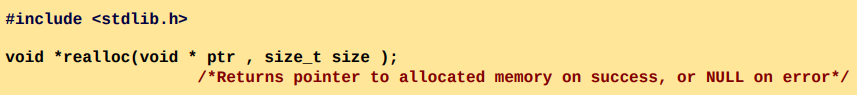


The calloc() function allocates memory for an array of identical items.

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

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



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. Top of the stack changes.

A picture containing text

Description automatically generated

When function executing the alloca() finishes, top of stack will go automatically back.

alloca() and malloc() family do not only use unallocated memory portion but also will request from kernel extra 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.

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