Environment of a Process

main Function

- main()
  - int main(int argc, char *argv[]);
    - argc : the number of command-line arguments
    - argv : an array of pointers to the arguments
  - when a C program is started by the kernel, a special start-up routine is called before the main() is called.
  - the start-up routine takes values from the kernel (the command-line arguments and the environment) and sets things up.

Process Termination

- normal termination
  - return from main
  - calling exit
  - calling _exit
- abnormal termination
  - calling abort
  - terminated by a signal

Process Termination

- exit( )
  - #include <stdlib.h>
  - void exit(int status);
    - causes normal program termination and the value of status is returned to the parent
    - All functions registered with atexit() and on_exit() are called in the reverse order of their registration
    - all open streams are flushed and closed.

Process Termination

- atexit( )
  - #include <stdlib.h>
  - int atexit(void (*function)(void));
    - registers the given function to be called at normal program termination
    - no arguments are passed
    - up to 32 functions can be registered
    - returns the value 0 if successful; otherwise the value -1 is returned

Process Termination

- on_exit( )
  - #include <stdlib.h>
  - int on_exit(void (*function)(int , void *), void *arg);
    - registers the given function to be called at normal program termination, whether via exit( ) or via return from the program's main.
    - The function is passed the argument to exit( ) and the arg argument from on_exit()
    - The integer argument of function is the exit status.

Process Termination

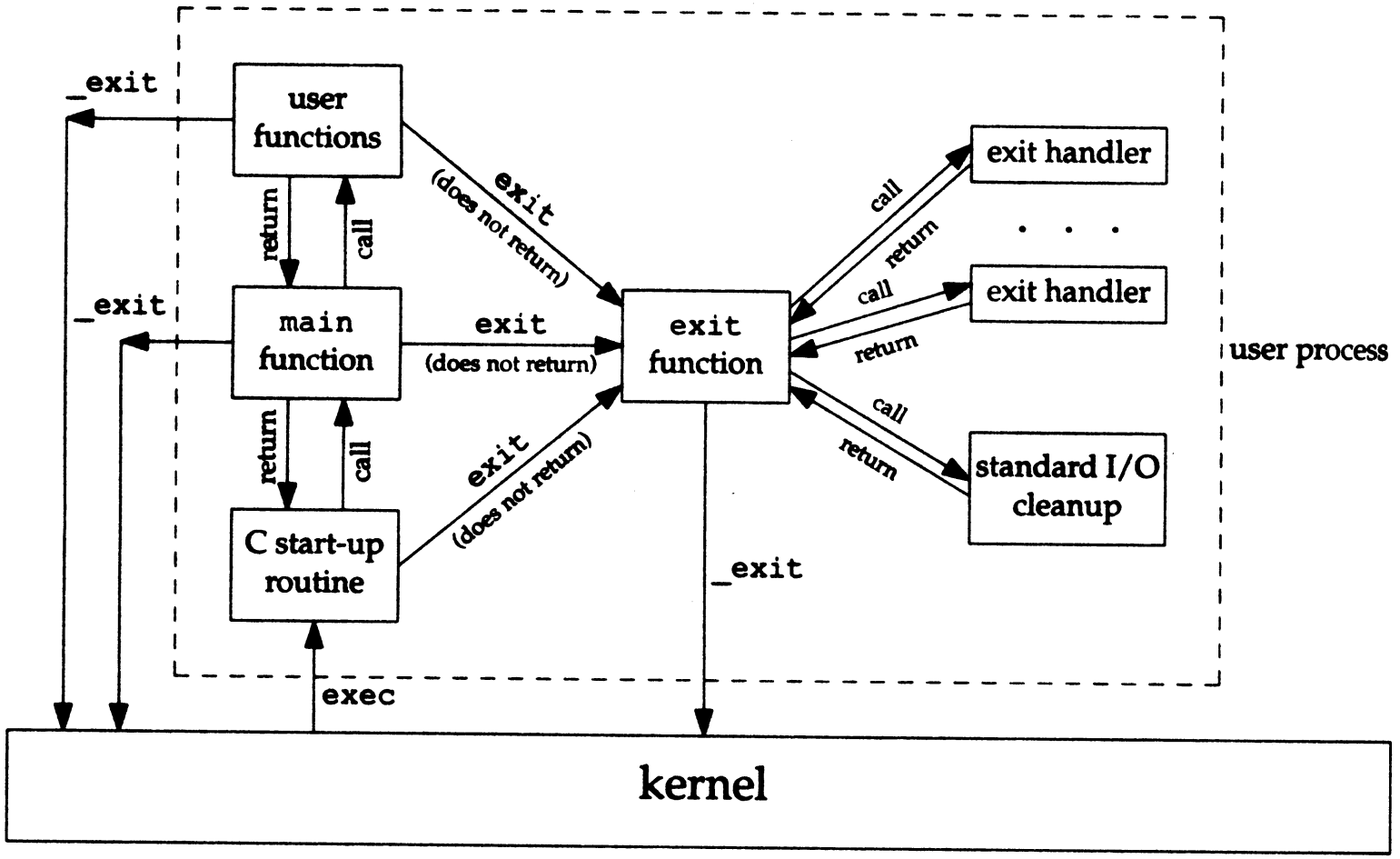
- _exit( )
  - #include <unistd.h>
  - void _exit(int status);
    - terminates the calling process immediately
    - Any open file descriptors belonging to the process are closed
    - status is returned to the parent process as the process's exit status

Process Termination

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

void exit1(void)
{       puts("exit1");
}
void exit2(void)
{       puts("exit2");
}
main()
{
        atexit(exit1);
        atexit(exit2);
        exit(3);
}
```

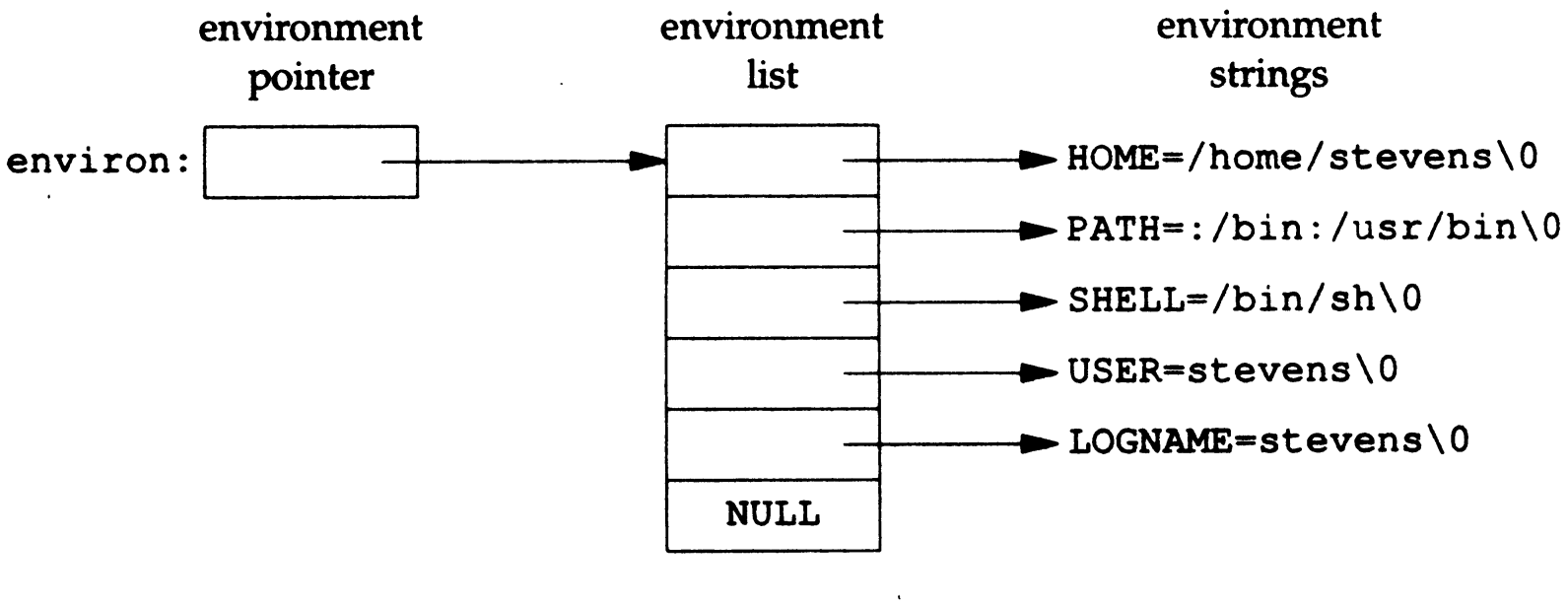
Start and Termination of a C program



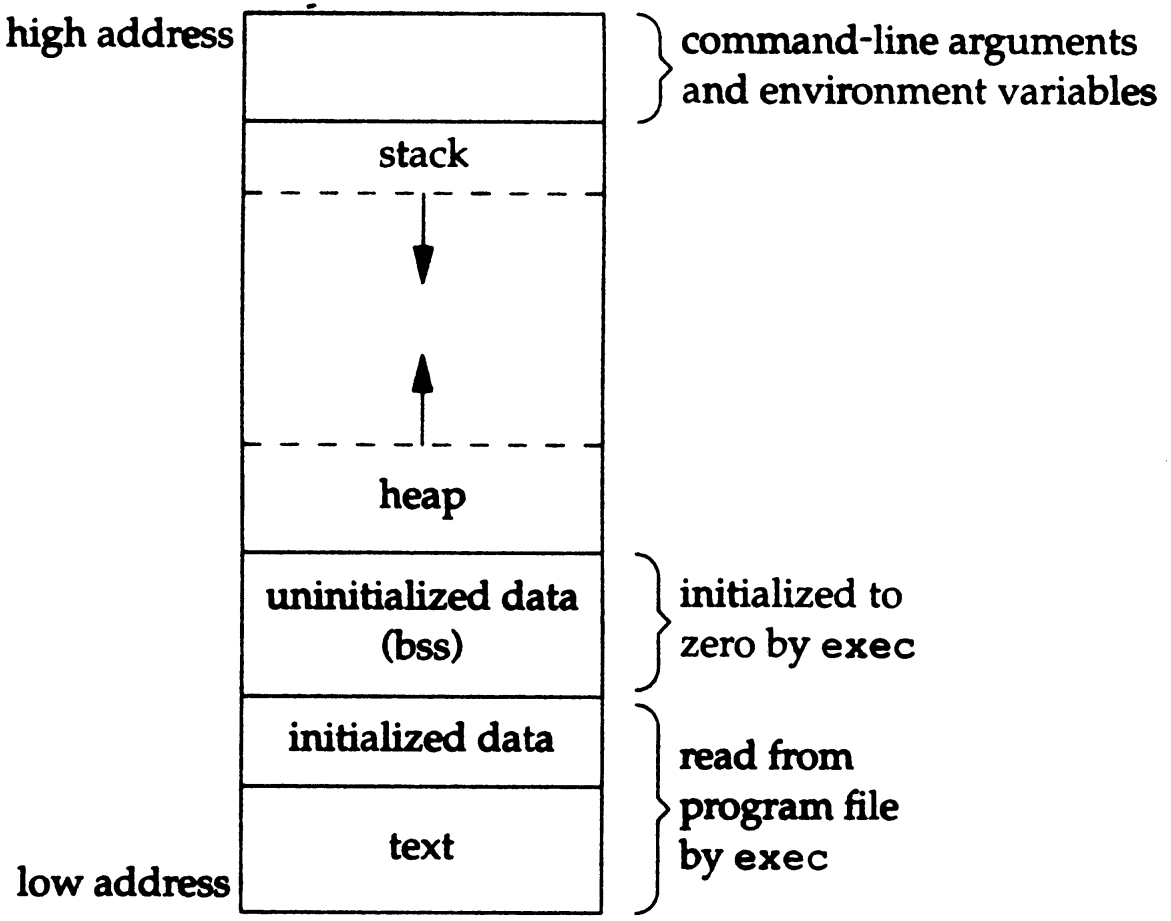
Environment List

- environment list
  - extern char **environ;
  - int main(int argc, char *argv[], char *envp[])
  - each environment consists of “name=value”
  - used to go through the entire environment.

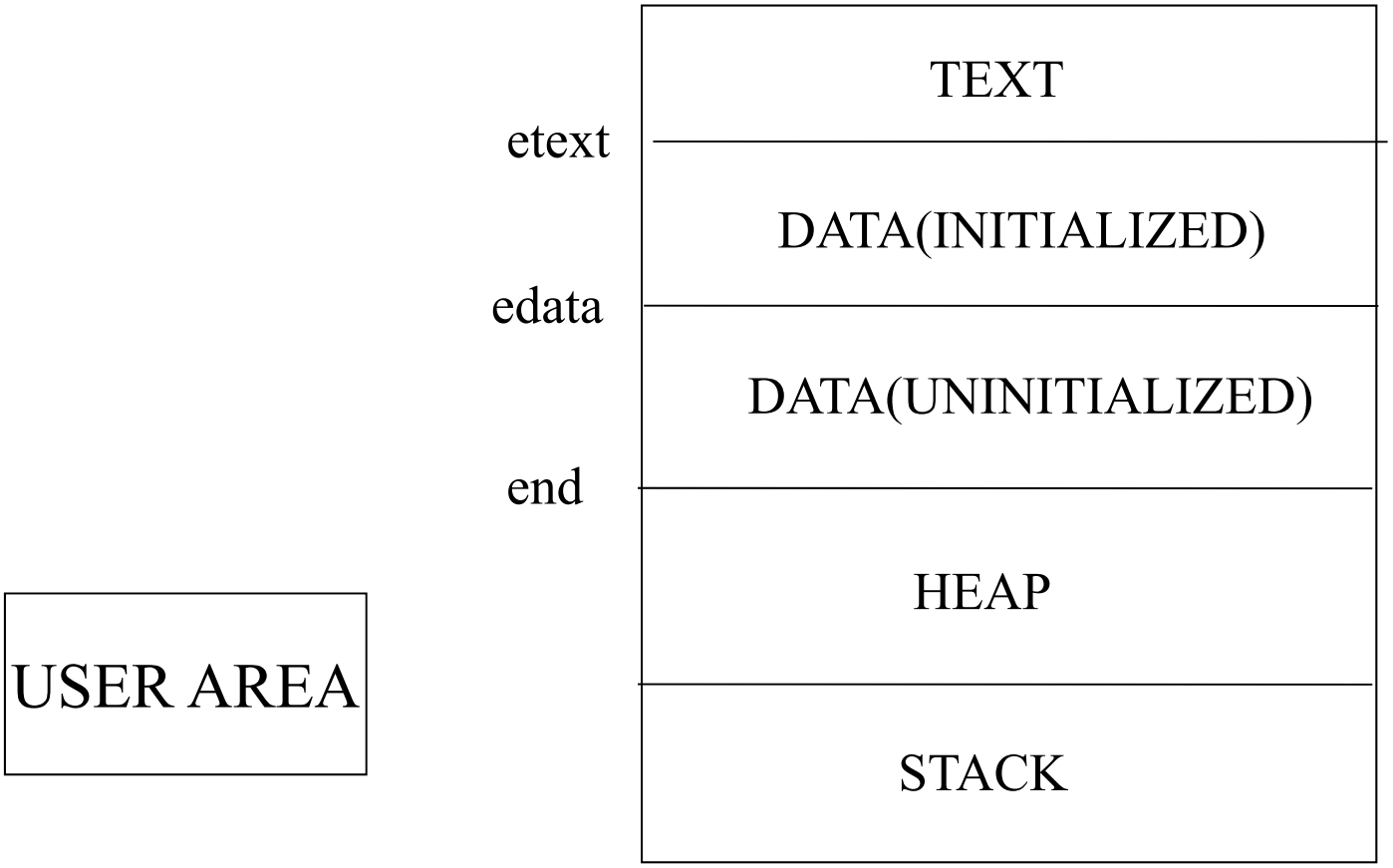
Environment List



Memory Layout of a C Program



Process Image



Process Image

- TEXT
  - user가 작성한 instruction들이 들어 가는 영역이다.
  - text는 항상 같은 virtual address에 load된다.
  - text는 공유될 수도 있고 안 될 수도 있지만 default는 공유된다
- DATA
  - static variable과 global variable을 저장한다.
  - DATA segment 부분은 initialized global variable을 가지는 initialized data 부분과 uninitialized global variable을 저장하는 uninitialized data부분으로 나뉜다.
- STACK
  - stack 부분은 local variable, function의 argument, function의 return value를 저장한다
  - data의 끝부분과 stack의 제일 윗부분 사이를 reference하면 memory fault가 발생한다.
- HEAP
  - malloc(), calloc() 등의 인터페이스로 동적 할당되는 부분
- USER AREA
  - OS의 주소공간
  - process가 수행 중일 때만 필요한 정보를 가지고 있다.
  - open file list

Process Image

```
#include <stdio.h>
#include <stdlib.h>
extern  etext,edata,end;

char a='a';
int b=1;
int e;int f;
int func(int arg)
{
 int f=10;
 printf("addr of arg : %p:%d\n",
                          &arg,arg);
 printf("addr of f : %p :%d\n",&f,f);
}
```

```
main()
{
static int c=10;
static int d;
int *dynamic;
        printf("end of text %p\n", &etext);
        printf("addr of a : %p :%c\n",&a,a);
        printf("addr of b : %p :%d\n",&b,b);
        printf("addr of c : %p :%d\n",&c,c);
        printf("end of initialized data %p\n", &edata);
        printf("addr of d : %p :%d\n",&d,d);
        printf("addr of e : %p :%d\n",&e,e);
        printf("addr of f : %p :%d\n",&f,f);
        printf("end of uninitialized data %p\n", &end);
        func(5);
        dynamic=(int *)malloc(8);
        *dynamic=100;
        printf("addr of dynamic : %p :%d\n",dynamic,*dynamic);
}
```

Shared Libraries

- shared libraries
  - remove the common library routine somewhere in memory that all processes reference
  - the executable file does not contain the library object code, but only a reference to the library name.
  - when the program is loaded, program interpreter (ld.so)
    - analyzes the library names in the executable file
    - locates the library in the system's directory tree
    - makes the requested code available to the executing process.

Memory Allocation

- library functions
  - #include <stdlib.h>
  - void *calloc(size_t nmemb, size_t size);
    - allocates memory for an array of nmemb elements of size bytes each
    - returns a pointer to the allocated memory
    - The memory is set to zero
  - void *malloc(size_t size);
    - allocates size bytes and returns a pointer to the allocated memory
    - The memory is not cleared

Memory Allocation

- void *realloc(void *ptr, size_t size);
  - changes the size of the memory block pointed to by ptr to size bytes
  - The contents will be unchanged to the minimum of the old and new sizes
  - newly allocated memory will be uninitialized
  - If ptr is NULL, the call is equivalent to malloc(size)
  - if size is equal to zero, the call is equivalent to free(ptr)
- void free(void *ptr);
  - frees the memory space pointed to by ptr

Memory Allocation

- void *alloca(size_t size);
  - allocates size bytes of space in the stack frame of the caller
  - this temporary space is automatically freed on return
- return value
  - returns a pointer to the allocated memory on success
  - the returned pointer of realloc may be different from ptr which is passed to realloc as an argument
  - returns NULL if the request fails

Memory Allocation

- system calls
  - #include <unistd.h>
  - int brk(void *end_data_segment);
    - sets the end of the data segment to the value specified by end_data_segment
    - end_data_segment must be greater than end of the text segment and it must be 16kB before the end of the stack
    - On success, brk returns zero
    - On error, -1 is returned

Memory Allocation

- void *sbrk(ptrdiff_t increment);
  - C library wrapper(actually, not a system call)
  - increments the program's data space by increment bytes
  - returns a pointer to the start of the new area
  - On error, -1 is returned
- brk and sbrk should not be used with malloc, calloc, realloc

Memory Allocation

- example

```
main()
{
        char *p;
        printf("before malloc: %p\n", sbrk(0));
        p = malloc(4096);
        printf("after malloc: %p\n", sbrk(0));
        free(p);
        printf("after free: %p\n", sbrk(0));
}
```

Environment Variables

- getenv( ) and putenv( )
  - #include <stdlib.h>
  - char *getenv(const char *name);
    - searches the environment list for a string that matches the string pointed to by name
    - returns a pointer to the value in the environment, or NULL if there is no match

Environment Variables

- int putenv(const char *string);
  - adds or changes the value of environment variables
  - The argument string is of the form name=value
  - If name does not already exist in the environment, then string is added to the environment
  - If name does exist, then the value of name in the environment is changed to value
  - returns zero on success, or -1 if an error occurs

Environment Variables

- setenv( ) and unsetenv( )
  - #include <stdlib.h>
  - int setenv(const char *name, const char *value, int overwrite);
    - adds the variable name to the environment with the value value, if name does not already exist
    - If name does exist in the environment, then its value is changed to value if overwrite is non-zero; if overwrite is zero, then the value of name is not changed.
  - void unsetenv(const char *name);
    - deletes the variable name from the environment

Environment Variables

- example

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

main()
{
    printf("Home directory is %s\n",
            getenv("HOME"));
    putenv("HOME=/");
    printf("New home directory is %s\n",
            getenv("HOME"));
}
```

setjmp and longjmp

- setjmp( ) and longjmp( )
  - #include <setjmp.h>
  - int setjmp(jmp_buf env);
    - useful for dealing with errors and interrupts encountered in a low-level subroutine of a program
    - saves the stack context/environment in env for later use by longjmp()
    - The stack context will be invalidated if the function which called setjmp() returns
    - return 0 if returning directly, and non-zero when returning from longjmp() using the saved context

setjmp and longjmp

- void longjmp(jmp_buf env, int val);
  - restores the environment saved by the last call of setjmp() with the corresponding env argument
  - After longjmp() is completed, program execution continues as if the corresponding call of setjmp() had just returned the value val
  - cannot cause 0 to be returned.
  - If longjmp is invoked with a second argument of 0, 1 will be returned instead.

setjmp and longjmp

```
#include  <setjmp.h>
#include  <stdio.h>
#include  <stdlib.h>
static void             f1(int, int, int);
static void             f2(void);
static jmp_buf          jmpbuffer;
int main(void)
{
      int     count;
      register int val;
      volatile int sum;
      count = 2; val = 3; sum = 4;
      if (setjmp(jmpbuffer) != 0) {
      printf("after longjmp:
      count= %d,val=%d,sum=%d\n",
                        count, val, sum);
      exit(0);
      }
      count = 97; val = 98; sum = 99;
/* changed after setjmp, before longjmp
      */
      f1(count, val, sum); /* never returns
      */
}
static void f1(int i, int j, int k)
{
      printf("in f1(): count = %d, val = %d,
      sum = %d\n",    i, j, k);
      f2();
}
static void f2(void)
{     longjmp(jmpbuffer, 1);}
```

```
$ gcc testjmp.c
$ gcc –O testjmp.c
```

Resource Limit and Usage

- getrlimit( ) and setrlimit( )
  - #include <sys/time.h>
  - #include <sys/resource.h>
  - #include <unistd.h>
  - int getrlimit (int resource, struct rlimit *rlim);
  - int setrlimit (int resource, const struct rlimit *rlim);

Resource Limit and Usage

- resource
  - RLIMIT_CPU /* CPU time in seconds (SIGXCPU)*/
  - RLIMIT_FSIZE /* Maximum filesize (SIGXFSZ)*/
  - RLIMIT_DATA /* max data size */
  - RLIMIT_STACK /* max stack size */
  - RLIMIT_CORE /* max core file size */
  - RLIMIT_RSS /* max resident set size */
    - if physical memory is tight, the kernel takes memory from processes that exceed their RSS

Resource Limit and Usage

- RLIMIT_NPROC /* max number of processes per real user ID*/
- RLIMIT_NOFILE /* max number of open files per process*/

- rlim

```
struct rlimit
{
    int  rlim_cur; /* soft limit: current limit */
    int  rlim_max; /* hard limit: maximum value for rlim_cur */
};
```

- RLIM_INFINITY : resource is unlimited

Resource Limit and Usage

- getrusage( )
  - #include <sys/time.h>
  - #include <sys/resource.h>
  - #include <unistd.h>
  - int getrusage (int who, struct rusage *usage);
    - who : RUSAGE_SELF, RUSAGE_CHILDREN

Resource Limit and Usage

```
struct rusage
{
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long ru_maxrss;   /* maximum resident set size */
    long ru_ixrss;    /* integral shared memory size */
    long ru_idrss;    /* integral unshared data size */
    long ru_isrss;    /* integral unshared stack size */
    long ru_minflt;          /* page reclaims */
    long ru_majflt;          /* page faults */
    long ru_nswap;      /* swaps */
```

Resource Limit and Usage

```
    long ru_inblock;       /* block input operations */
    long ru_oublock;       /* block output operations */
    long ru_msgsnd;        /* messages sent */
    long ru_msgrcv;        /* messages received */
    long ru_nsignals       /* signals received */
    long ru_nvcsw;      /* voluntary context switches */
    long ru_nivcsw;    /* involuntary context switches */
};
```