UNIX SYSTEM PROGRAMMING

UNIT - III

Unix process

- Process : its an instance of a running program.
- Program : is an entity that resides on a non-volatile media(such as disk)
- Process :is an entity that is being executed(with at least some portion, i.e. segment/page) in RAM.

Objectives

- Define a process
- Define main
- Different ways for the process to terminations
- Command line arguments passed to the new program
- Use environment variables in a process
- Memory layout of a C program
- Allocation of additional memory
- Explain longjmp and setjmp functions and their interaction with the stack
- Explain Resource limits of a process

main function

- Prototype of *main*

  ```
  int main(int argc, char* argv[]);
  ```

  A special start-up routine is called before the *main* function is called

- Specifies the starting address for the program & takes values from the kernel (command-line arguments and environment) and sets things up for *main*.

Process termination

1. Normal termination
   a) return from *main*
   b) calling *exit*
   c) calling *_exit*

2. Abnormal termination
   a) calling *abort*
   b) termination by a signal

Process termination

```
#include<stdlib.h>
    void exit(int status);

  #include<unistd.h>
    void _exit(int status);
```

exit is specified by ANSI C, while _exit is specified by POSIX.1

Process termination

```
#include <stdio.h>
 main()
 {
     printf (“Hello, World\n”);
 }
```

Declaration of main should really be

```
int main (void)
```

Process termination

atexit function:

- With ANSI C a process can register up to 32 functions (*exit handlers*) that are automatically called by *exit*.
- These *exit handlers* are registered by calling the *atexit* function

  Prototype of *atexit* function:

```
#include <stdlib.h>
int atexit (void ( *func ) (void));
```

Return: 0 if OK, nonzero on error

Process termination

- *exit* calls these functions in reverse order of their registration.

- Each function is called as many times as it is registered

```
#include “ourhdr.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;
}
```

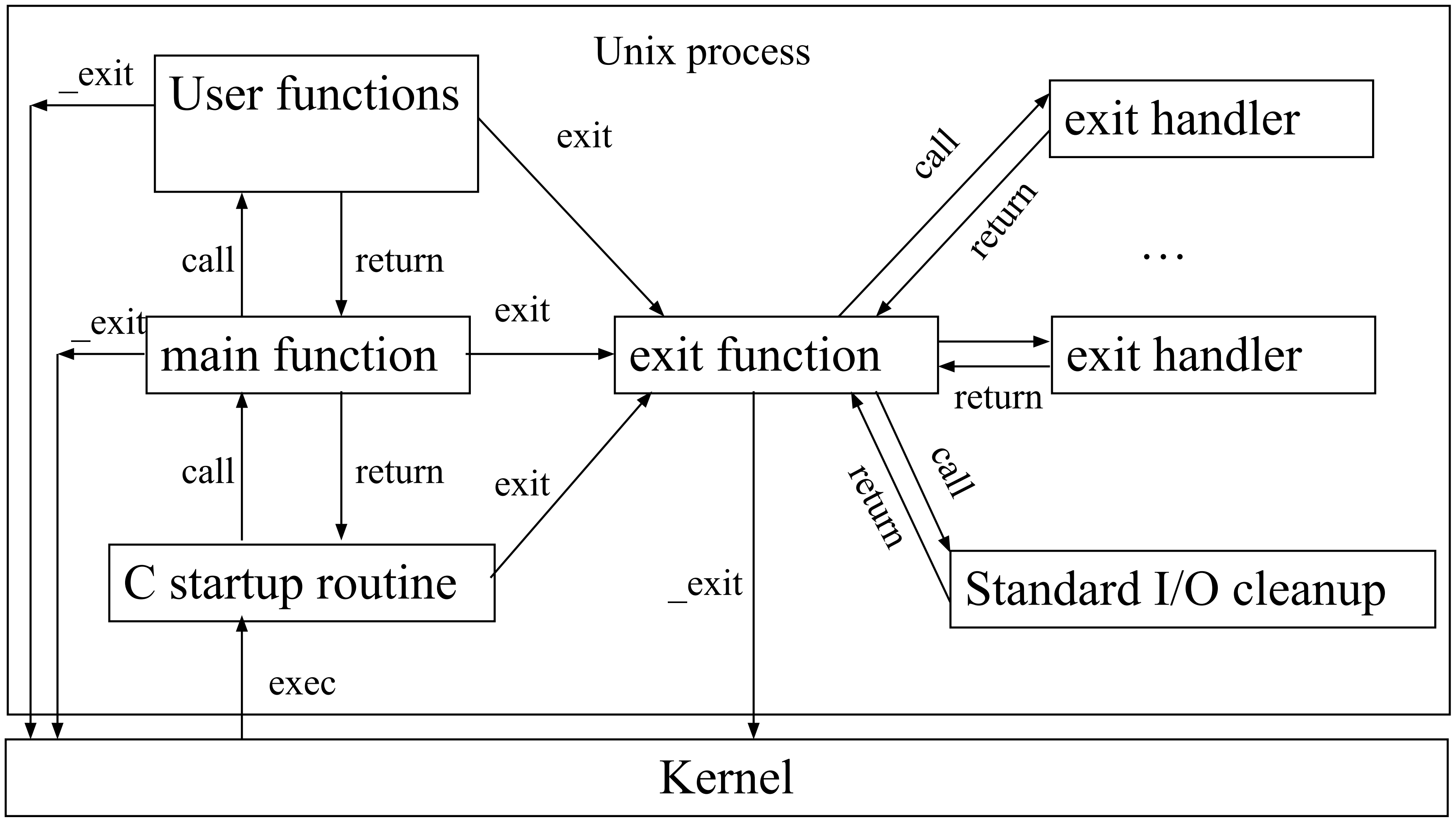
```
static void my_exit1(void)
{
      printf (“first exit handler\n”);
 }
 static void my_exit2(void)
 {
     printf (“second exit handler\n”);
 }
```

With ANSI C and POSIX.1, *exit* first calls *exit handlers* and then *fclose*s all open streams.

Output

```
$ a.out
main is done
first exit handler
first exit handler
second exit handler
```

How a C program is started & how it is terminated



Command Line Arguments

When a program is executed, the process that does the exec can pass command line arguments to the new program

Part of the normal operation of the Unix shells

```
#include “ourhdr.h”
int main(int argc, char* argv[])
{
      int i;

      for (i = 0; i<argc; i++)
         printf (“argv[%d] : %s\n”, i, argv[i]);
      return 0;
}
```

In ANSI C & POSIX.1 , argv [argc] is a null pointer

Output

```
$ ./echoarg arg1 Test foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: Test
argv[3]: foo
```

Environment List

Each program is also passed an environment list, which is an array of pointers, with each pointer containing the address of a null terminated string

Address of the array of pointers is contained in the global variable *environ*

*extern char **environ;*

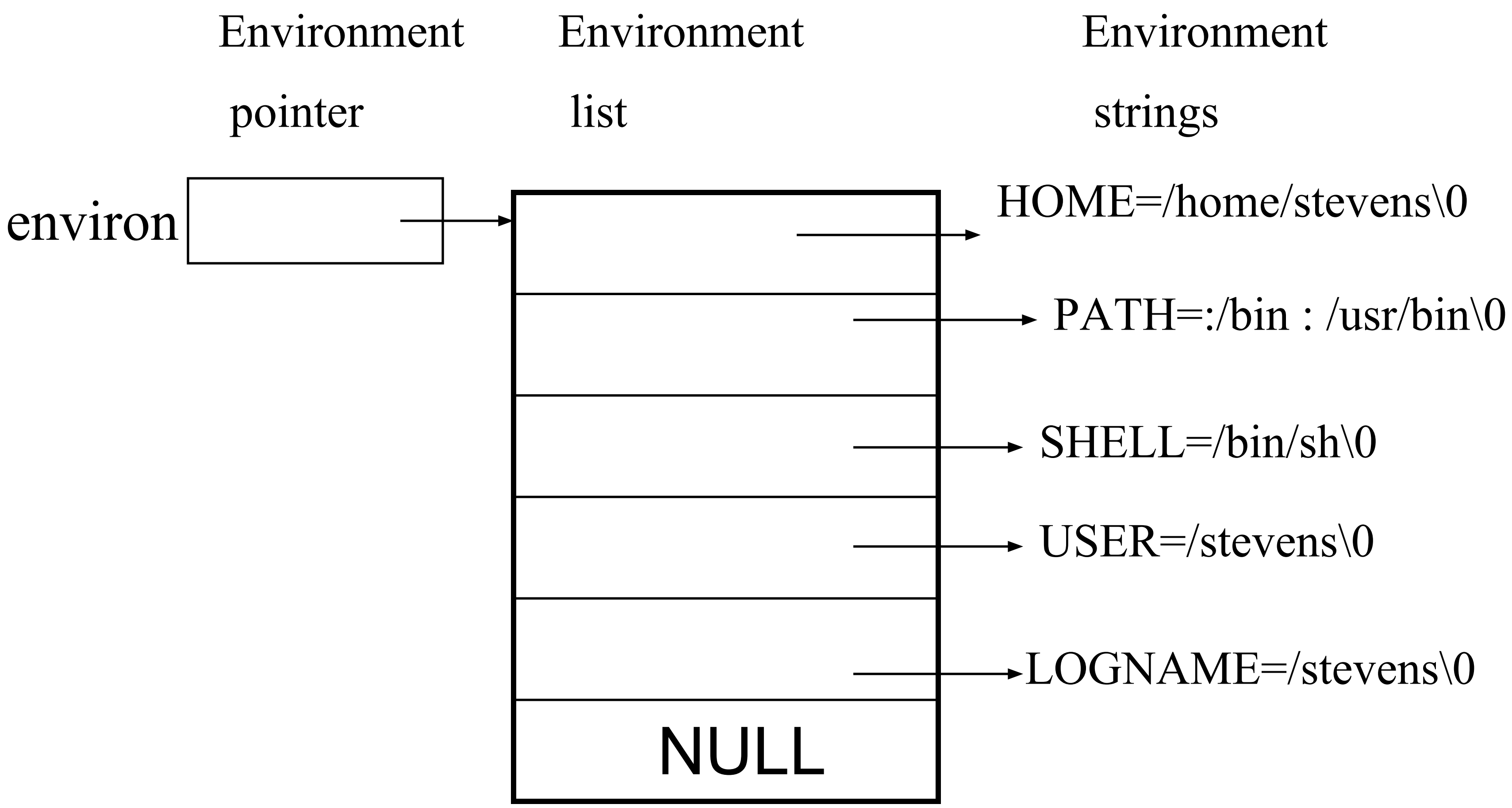


Fig: Environment consisting of five C character strings

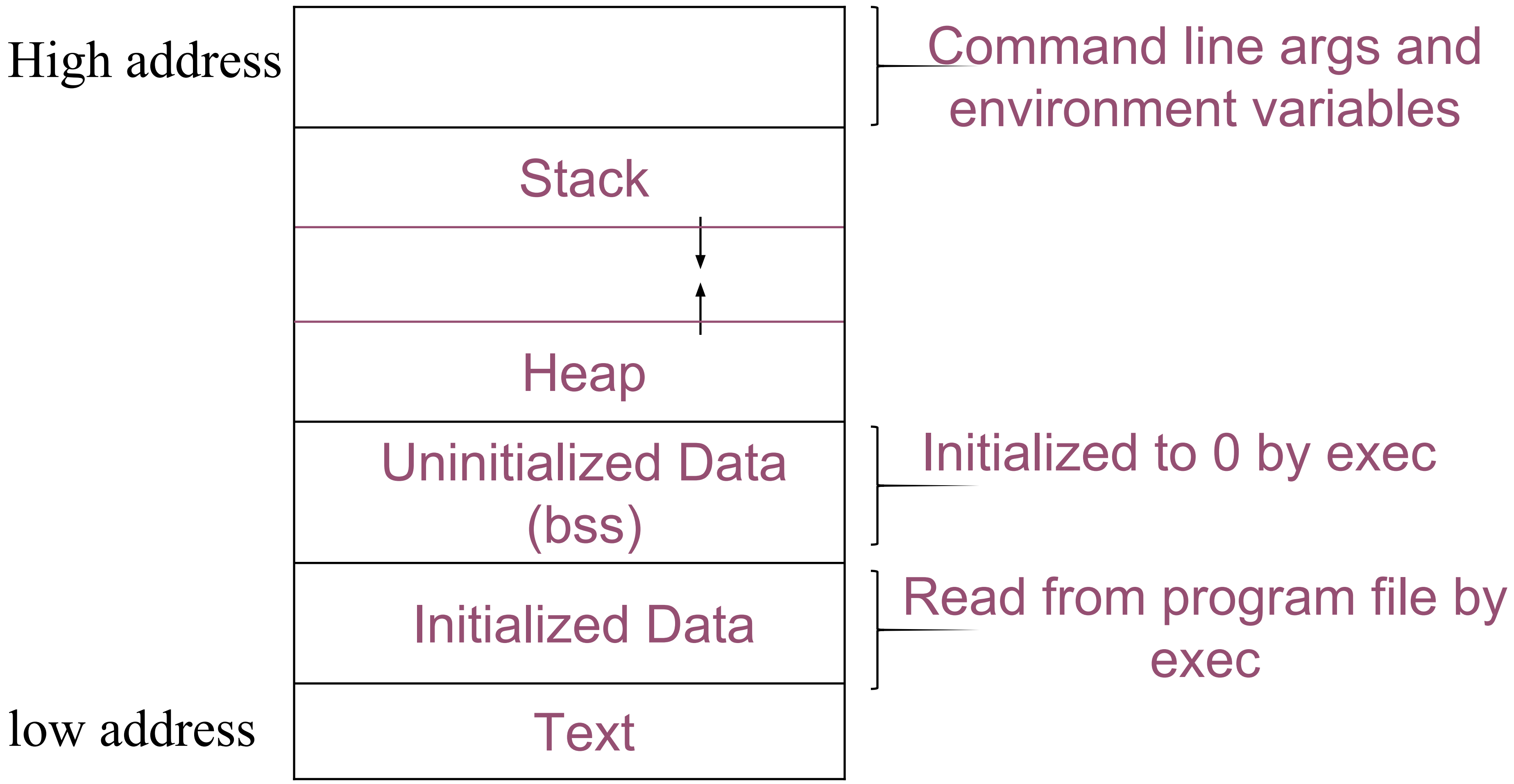
Environment List

- Most Unix systems have provided a third argument to the main function that is the address of the environment list

```
int main(int argc, char* argv[], char* env[]);
```

- Access to specific environment variable is normally through *getenv* and *putenv* functions instead of through the *environ* variable

Memory Layout of a C program



- Text segment:
  - this is the m/c instructions that are executed by the CPU
  - This segment is sharable so that only a single copy needs to be in memory for frequently executed programs

- Initialized data segment
  - It is called as DS
  - It contains the variables that are specially initialized in the program.

    ex: int count=0;

- Uninitialized data segment:
  - data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing

    ex: `int sum[100];`

- Stack:
  - Automatic variables are stored, along with information that is saved each time a function is called.
- Heap
  - Dynamic memory allocation takes place on the heap

Memory Allocation

Three functions specified by ANSI C for memory allocation:

1. malloc：Allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.

2. calloc： Allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.

3. realloc ：Changes the size of the previously allocated area (increases or decreases). The initial value of the space between the old contents and the end of the new area is indeterminate.

Prototypes:

```
#include <stdlib.h>

void  *malloc (size_t size);
void *calloc (size_t nobj, size_t size);
void  *realloc (void *ptr, size_t newsize);
```

All three return: nonnull pointer if OK; NULL on error

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

- Since the three *alloc* functions return generic pointers, if we #include <stdlib.h>, we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type
- realloc : most common usage is to increase an area.
    - If there is room beyond the end of the existing region for the requested space, then realloc doesn’t have to move anything
    - If there is no room at the end of the existing region, then realloc moves the existing block to a new region and frees the old region and returns a pointer to this new region.

- Most implementations allocate a little more space than is requested and use the additional space for record keeping.
- Possible errors:
  1. freeing a block that is already freed.
  2. calling free with a pointer that was not obtained from one of the three alloc functions.
  3. Not calling free to return unused space – memory leakage.

Environment Variables

- Environment strings are usually of the form:

  name=value

  Ex: Shell uses various environment variables, some are set automatically at login (HOME, USER, etc..) and others are for us to set (MAILPATH)

- ANSI C defines a function that we can use to fetch values from the environment

  #include <stdlib.h>

  char *getenv (const char *name);

  Returns: pointer to the value associated with name, NULL if not found

- We may also want to set an environment variable (change the value of an existing variable or add a new variable to the environment)
- The prototypes of the various functions are

putenv :

```
#include <stdlib.h>
int putenv (const char *str);
```

places a string of the form name=value in the environment list

- if the name already exists, its old definition is lost

setenv :

```
int setenv (const char *name, const char *value, int
                                                   rewrite);
```

Both return : 0 if OK, nonzero on error

- sets name to value.
- if name already exists then
  - if rewrite is nonzero, the existing definition for name is first removed
  - if rewrite is 0, an existing definition for name is not removed. No error occurs.

unsetenv :

```
void unsetenv (const char *name);
```

removes any definition of name.

- No error if such a definition does not exist

- How these functions operate when modifying the environment list?

Deleting a string :

- find the pointer in the environment list and move all subsequent pointers

Adding or modifying an existing string :

- the space at the top cannot be expanded as we can't go beyond the top of the address space of the process.
- cannot expand downward because all the stack frames below it can't be moved

- If modifying an existing *name*:

a) if size of new value <= size of existing value, copy the new string over the old string

b) if size of new value > the size of old value,

- *malloc* to obtain room for the new string,
- copy the new string to this area
- replace the old pointer in the environment list for *name* with the pointer to this *malloc*ed area.

If adding a new *name*:

- Call *malloc* to allocate room for the *name=value* string and copy the string to this area

a) if it's the first time we've added a new *name*,

  - call *malloc* to obtain room for a new list of pointers
  - copy the old environment list to this new area
  - store a pointer to the *name=value* string at the end of this list of pointers
  - also store a null pointer at the end of this list.
  - finally, set *environ* to point to this new list of pointers

b) if this isn't the first time we've added new strings to the environment list

- just call *realloc* to allocate room for one more pointer.
- the pointer to the new *name=value* string is stored at the end of the list, followed by a null pointer.

setjmp and longjmp functions

- Allows us to *goto* a label that's in another function
- Useful for handling error conditions that occur in a deeply nested function call.

```
    #include "ourhdr.h"
 #define TOK_ADD 5
 void do_line (char *);
 void cmd_add (void);
 int get_token (void);
```

```
int main (void)
   {
      char line [MAXLINE};
      while (fgets (line, MAXLINE, stdin) != NULL)
        do_line (line);
      exit(0);
   }
     char *tok_ptr;
   void do_line (char *ptr)
   {
   int cmd;     tok_ptr = ptr;
   while ( ( cmd = get_token() ) > 0 )
     {  switch (cmd) {   /* one case for each command */
                 case TOK_ADD:  cmd_add();
                                          break;     }
    }   }
```

```
void cmd_add (void)
{    int token;
    token = get_token();
    /* rest of processing for this command */
}
int get_token(void)
{
     /* fetch next token from line pointed to by tok_ptr */
}
```

Solution : Use nonlocal goto – the *setjmp* and *longjmp* functions

- Nonlocal because:
  - We're not doing a normal C goto statement within a function; instead we're branching back through the call frames to a function that is in the call path of the current function.
- Prototypes:

#include <setjmp.h>

int setjmp (jmp_buf env);

  - Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp (jmp_buf env, int val);

  - jmp_buf - This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjump

- *setjmp()* is called from the location that we want to return to.
- *val* in *longjmp()* is a nonzero value that becomes the return value from *setjmp()*.
- reason for using *val* is to allow us to have more than one *long*jmp for each *set*jmp.

```
#include "ourhdr.h"
#include <setjmp.h>
#define TOK_ADD 5
jmp_buf jmpbuffer;
int main (void)
{
    char line [MAXLINE};
    if (setjmp (jmpbuffer) != 0)
      printf ("error");
    while (fgets (line, MAXLINE, stdin) != NULL)
      do_line (line);
    exit(0);
}
```

. . .

```
void cmd_add (void)
{
    int token;
    token = get_token();

    if (token < 0)  /* an error has occurred */
        longjmp (jmpbuffer, 1);
    /* rest of processing for this command */
}
```

getrlimit and setrlimit functions

- Every process has a set of resource limits, some of which can be queried and changed by the *getrlimit* and *setrlimit* functions.

- Prototypes:

```
#include <sys/limits.h>
#include <sys/resource.h>
int getrlimit (int resource, struct rlimit *rlptr);
int setrlimit (int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, nonzero on error

- Each call to these functions specifies a single resource and a pointer to the following structure:
- Struct rlimit

```
  {
        rlim_t  rlim_cur;     /* soft limit: current limit */
        rlim_t  rlim_max;     /* hard limit: max value for
                    rlim_cur */
  }
```

Three rules govern the changing of the resource limits:

1. A soft limit can be changed by any process to a value less than or equal to its hard limit

2. Any process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.

3. Only a superuser process can raise a hard limit.

- An infinite limit is specified by the constant RLIM_INFINITY
- The resource argument takes on one of the following values

RLIMIT_CORE	Max. size in bytes of a core file. A limit of 0 prevents the creation of a core file
RLIMIT_CPU	Max. amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process
RLIMIT_DATA	Max. size in bytes of the data segment. This is the sum of initialized data, uninitialized data and the heap.
RLIMIT_FSIZE	Max. size in bytes of a file that may be created. When the soft limit is exceeded, the SIGXFSZ signal is sent to the process

RLIMIT_NOFILE	Max. number of open files per process
RLIMIT_NPROC	Max. number of child processes per real user ID
RLIMIT_RSS	Max. resident set size (RSS) in bytes (the number of virtual pages resident in RAM). If physical memory is tight, the kernel takes memory from processes that exceed their RSS.
RLIMIT_STACK	Max. size in bytes of the stack

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include “ourhrd.h”
#define doit (name)   pr_limits (#name, name)
static void pr_limits (char *, int);
int main(void)
{
  doit (RLIMIT_CORE);
  doit (RLIMIT_CPU);
  doit (RLIMIT_DATA);
  doit (RLIMIT_FSIZE);
#ifdef RLIMIT_MEMLOCK
    doit (RLIMIT_MEMLOCK);
#endif
#ifdef RLIMIT_NOFILE
    doit (RLIMIT_NOFILE);
#endif
#ifdef RLIMIT_NPROC
    doit (RLIMIT_NPROC);
#endif
#ifdef RLIMIT_STACK
    doit (RLIMIT_STACK);
#endif
     exit(0);
}
```

```
static void pr_limits (char *name, int resource)    {
    struct rlimit limit;
    if (getrlimit (resource, &limit) < 0)
        err_sys (“getrlimit error for %s”, name);
    printf(“%-14s “, name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf (“(infinite)” );
    else
        printf(“%10ld “, limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf (“(infinite)” );
    else
        printf(“%10ld “, limit.rlim_max);
}
```

UNIX Kernel support for Processes

The data structure for processes and execution of processes are dependent on operating system implementation.

The process data structure and operating system support in the UNIX System V will be considered.

A process has

- **text**: machine instructions (may be shared by other processes)
- **data**: static and global variables & their data
- **stack:** provides a storage for function arguments, automatic variables, & return addresses of all unctions for a process at any time

Process may execute either in user mode and in kernel mode

Process information are stored in two places:

- Process table:
  - It keeps track of all active processes.
  - Some of the processes belong to the kernel are called system processes.
  - Each entry in the process table contains pointers to the stack, data, stack segments
- User table (per process U Area):
  - It is an extension of a process table entry and contains other process-specific data, such as
  - the file descriptor,
  - current root and working directory inode numbers and
  - a set of system imposed resource limits, etc

Process table

an entry in process table has the following information:

- process state
- PID: process id
- UID: user id
- scheduling information
- signals that is sent to the process but not yet handled
- a pointer to per-process-region table
- Various timers – process execution time, resource utilization

There is a single process table for the entire system

U Area: has the following information

- User IDs
- various Timer:
  - Execution time in user mode
  - Execution Time in kernel mode
- An error field: keeps error during system call
- Return value field: result of system call
- I/O parameters
  - Amount of data transfer
  - Address of source and target etc
- The current directory and current root
- User file descriptor table
- The control terminal field:
  - login terminal associated with the process, if one exists

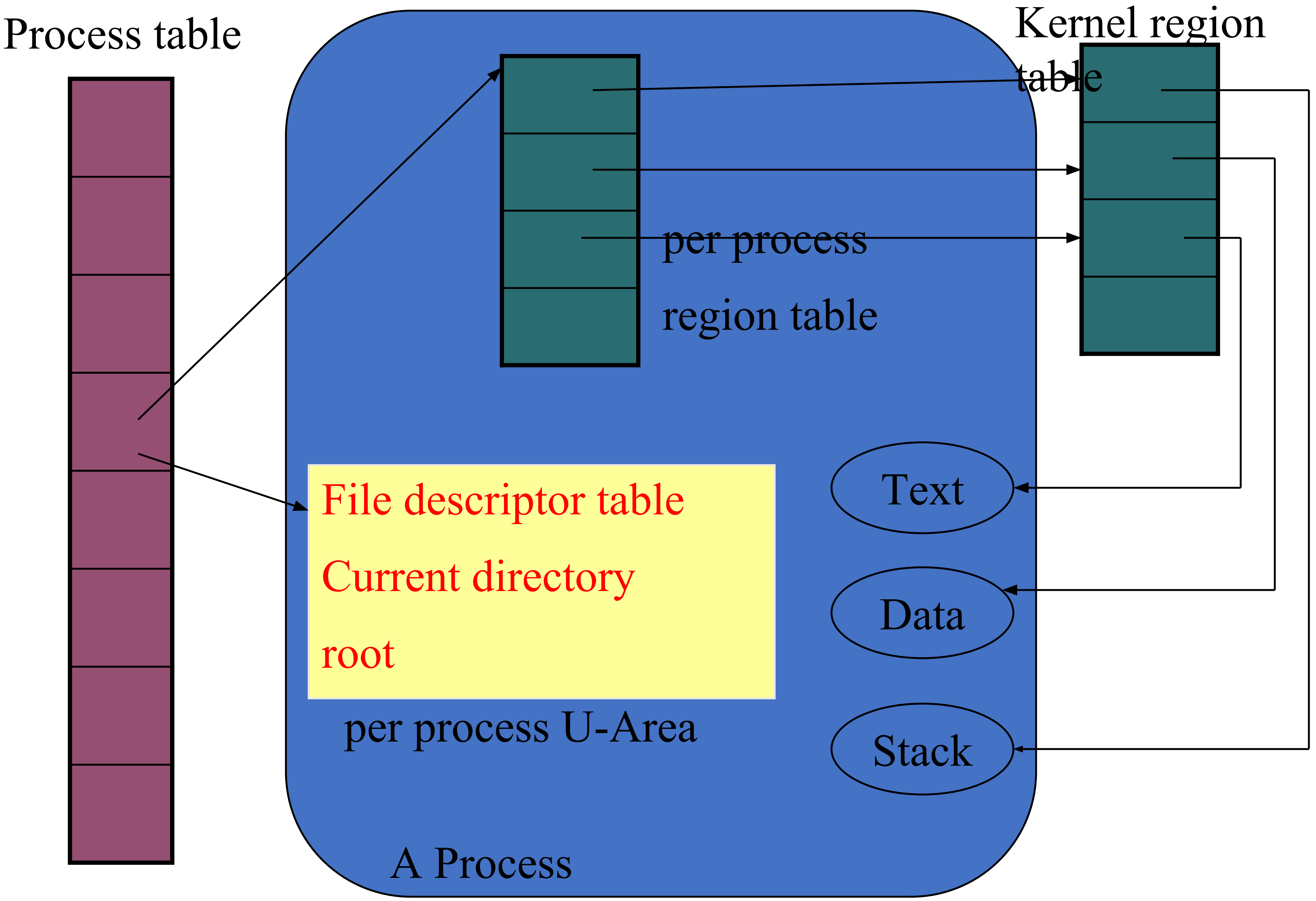


Fig: A UNIX process data structure

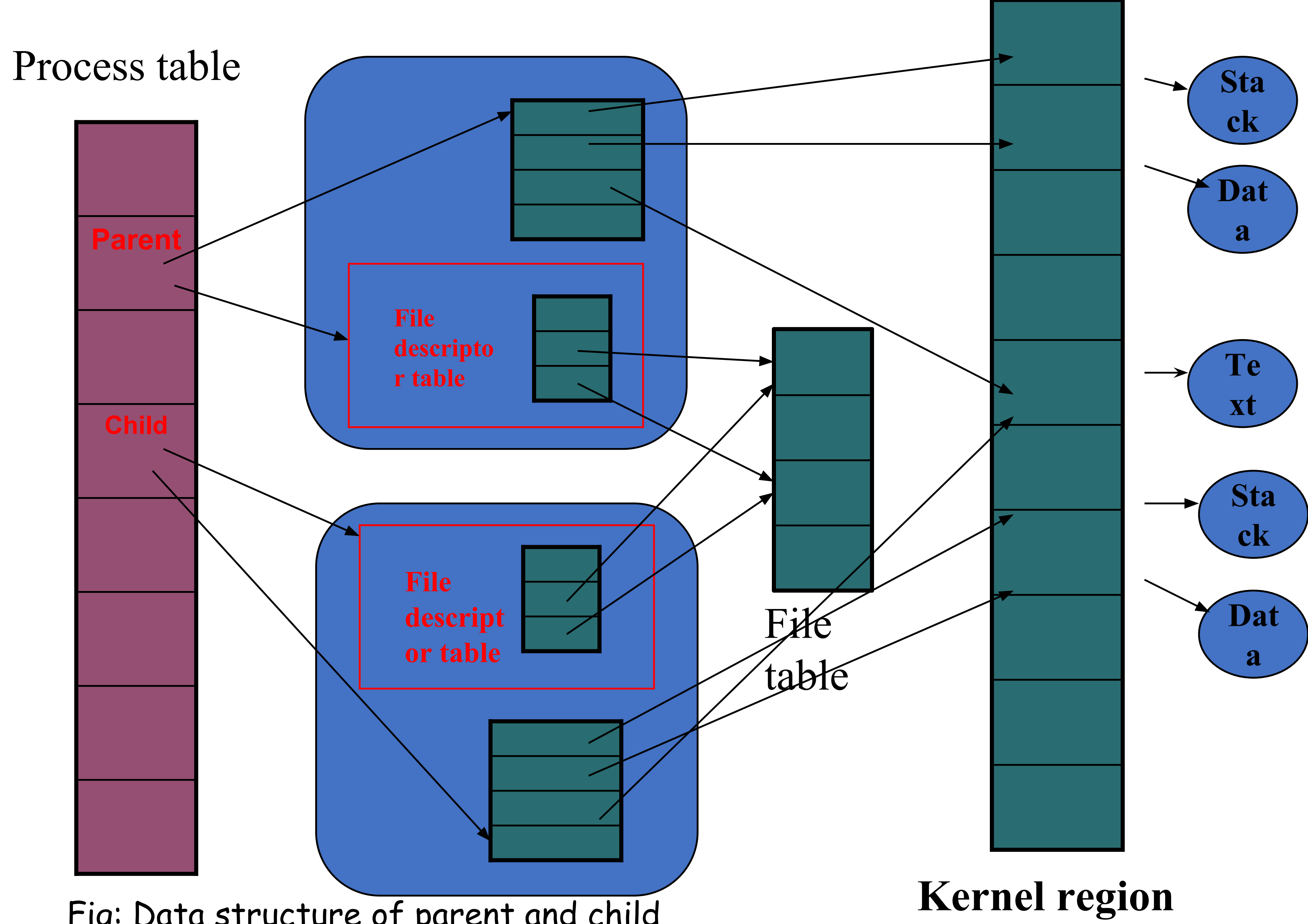


Fig: Data structure of parent and child processes after fork

A process is assigned the following attributes which are either inherited from its parent or set by the kernel:

1. Real user identification number (rUID) : user ID of the user who created the parent process.

2. Real group identification number (rGID) : group ID of the user who created the parent process.

3. Effective user identification number (eUID) : normally same as rUID, except when the file that was executed to create the process has its set-UID flag turned on. In that case, the process eUID will take on the UID of the file.

4. Effective group identification number (eGID) :normally same as rGID, except when the file that was executed to create the process has its set-GID flag turned on. In that case, the process eGID will take on the GID of the file.

5. Saved set-UID and Saved set-GID : these are assigned eUID and eGID, respectively, of the process.

6. Process group identification number (PGID) and the session identification number (SID) : These identify the process group and session of which the process is member.

7. Supplementary group identification numbers : set of additional group IDs for a user who created the process..
8. Current directory : reference to a working directory file
9. Root directory : the reference to a root directory file
10. Signal Handling : Signal handling settings
11. Signal mask : a signal mask that specifies which signals are to be blocked.
12. Umask : a file mode mask that is used in creation of files to specify which accession rights should be taken out.
13. Nice value : the process scheduling priority value
14. Controlling terminal : the controlling terminal of the process

- In addition to these attributes, the following attributes are different between the parent and child processes:

1. Process identification number (PID) : an integer identification number that is unique per process in an entire operating system.
2. Parent process identification number (PPID) : the parent process PID.
3. Pending signals: The set of signals pending delivery to the parent process. This is reset to null in the child process.
4. Alarm clock time : the process alarm clock time is reset to zero in the child process.
5. File Locks : the set of file locks owned by the parent process is not inherited by the child process

- After fork, a parent process may choose to
  - suspend its execution until its child process terminates by calling the *wait* or *waitpid* system call

    or
  - continue execution independent of its child process, where the parent process may use the *signal* or *sigaction* function to detect or ignore the child process termination.
- A process terminates its execution by calling the _exit. An exit status code of 0 means that the process has completed its execution successfully, and any nonzero exit code indicates failure has occurred

- A process can execute a different program by calling the *exec* API.
- if the call succeeds,
  - the kernel will replace the process's existing text, data and stack segments with a new set that represents the new program to be executed.
  - the process is still the same process (the process ID and the parent process ID are the same) and its file descriptor table and opened directory streams remain mostly the same.

- When the ***exec'*****ed** program completes its execution, it terminates the process.

- fork and exec are commonly used together to spawn a subprocess to execute a different program.

The advantage of this method is:

- A process can create multiple processes to execute multiple programs concurrently.

- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of the of its child process.

END