Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

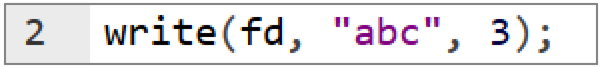
Outline

- I/O Buffering
- Process Environment
- Process Creation

I/O Buffering

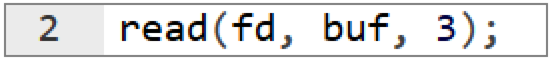
Buffering

- For speed and efficiency, I/O systems calls and I/O library calls buffer data.
- Two levels of buffering
  - Kernel buffer cache
    - Makes read and write calls faster
    - Reduces number of disk access by kernel
  - Buffering in the standard i/o library (optional)
    - Reduces number of system calls to access data



```
write(fd, "abc", 3);
```

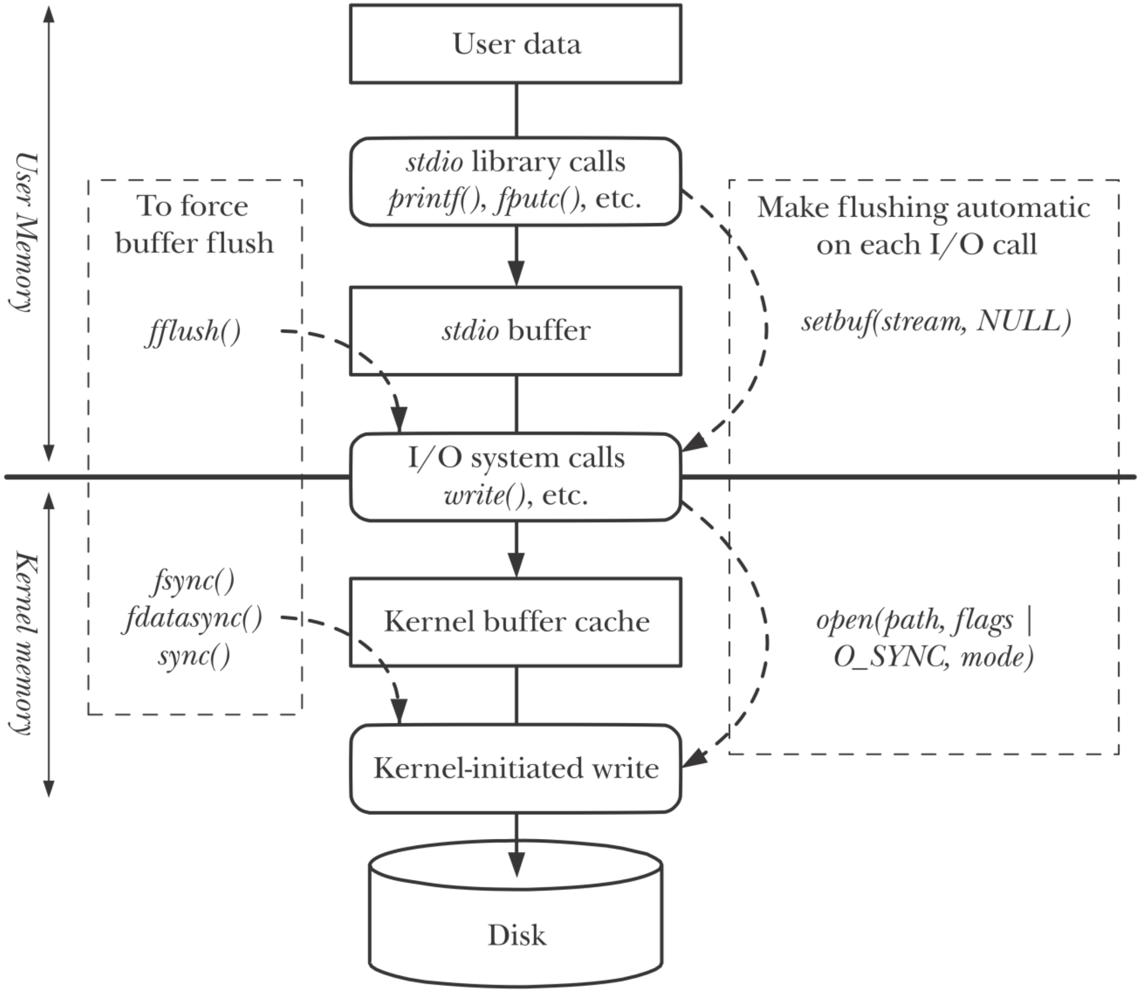
- This call can’t directly write to the disk. This writes to a buffer in the kernel. Kernel later syncs these contents with the disk.



```
read(fd, buf, 3);
```

- This call transfers 3 bytes of data from kernel buffer to user buffer *buf*.

Writing Data to a File

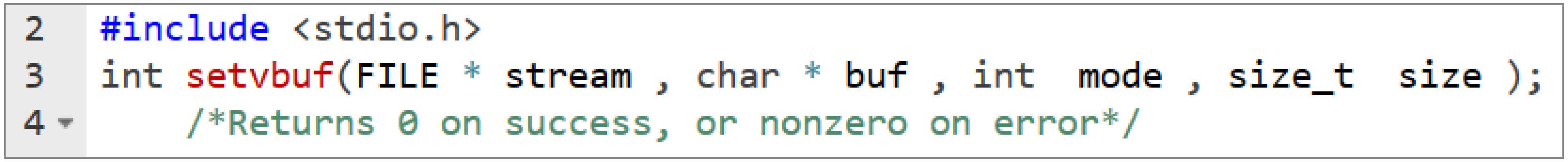


Kernel Buffering of File I/O

- Buffer cache:
  - Set of buffers Kernel maintains to store disk blocks. Seize of buffer cache is adapted as per the availability of the physical memory.
  - When a read() call is issued, Kernel reads the disk block from the disk and stores it in a buffer.
  - Data is copied from buffer cache to the buffer in the user space.
  - Similarly when a user process writes, kernel writes to the buffer.
  - Kernel periodically syncs dirty buffers with disk.
- This allows read() and write to be faster.

Buffering in *stdio* Library

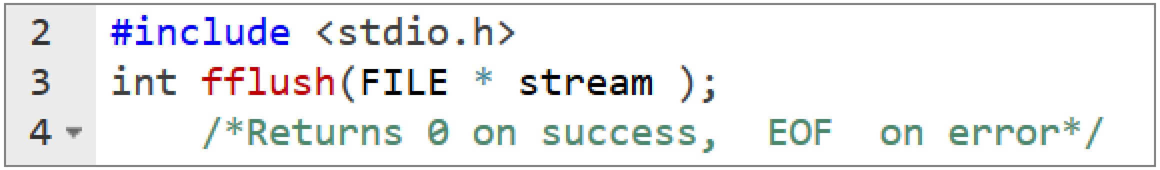
- C library buffers the data to reduce the number system calls (read, write). *fopen()* call opens a buffered stream for a file.



```
#include <stdio.h>
int setvbuf(FILE * stream , char * buf , int  mode , size_t  size );
    /*Returns 0 on success, or nonzero on error*/
```

  - This is a library function that controls the type of buffering.
  - This function must be called before any I/O operation.
  - If *buf* is null, stdio automatically allocates the buffer for use with the *stream*.
  - mode
    - _IONBF: no buffering. E.g. stderr
    - _IOLBF: line buffering. Default for terminal devices. Output is buffered until newline char. Data is read a line at a time.
    - _IOFBF: fully buffered I/O. data is read or written in units of buffer size. Default for disk files.

Flushing a stdio Buffer

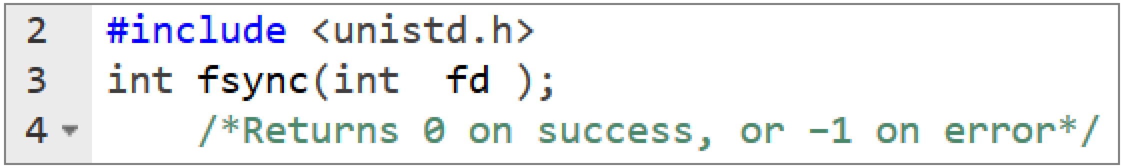


```c
#include <stdio.h>
int fflush(FILE * stream );
    /*Returns 0 on success,  EOF  on error*/
```

- Regardless of the current buffering mode, at any time, we can force the data to written.
- fflush() function flushes the output buffer for that particular FILE stream.
  - It can be used on input stream also.

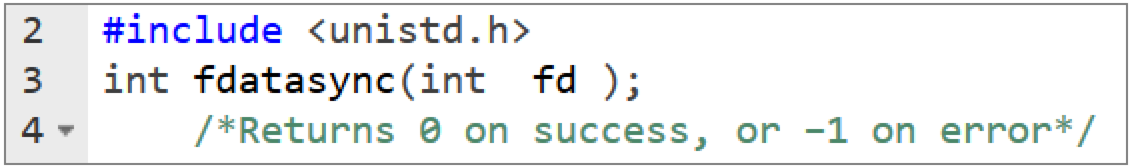
Controlling Kernel Buffering of File I/O

- Two type of synchronization
  - Synchronized I/O file integrity
    - Both data and meta data about the file are synchronized with disk.
  - Synchronized I/O data integrity
    - Only data is synchronized with the disk.



```c
#include <unistd.h>
int fsync(int  fd );
    /*Returns 0 on success, or -1 on error*/
```

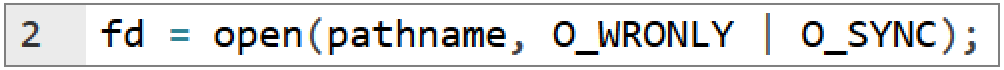
- Flushes both data and metadata such as file size, time stamps etc associated with *fd*.



```c
#include <unistd.h>
int fdatasync(int  fd );
    /*Returns 0 on success, or -1 on error*/
```

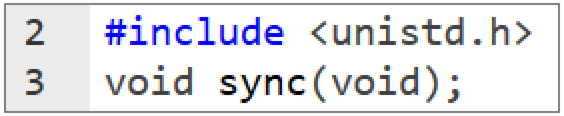
- Flushes only data buffers of file descriptor *fd.*

Controlling Kernel Buffering of File I/O



```c
fd = open(pathname, O_WRONLY | O_SYNC);
```

- If we use O_SYNC flag while opening a file, after every write, both data and metadata will be flushed to disk.

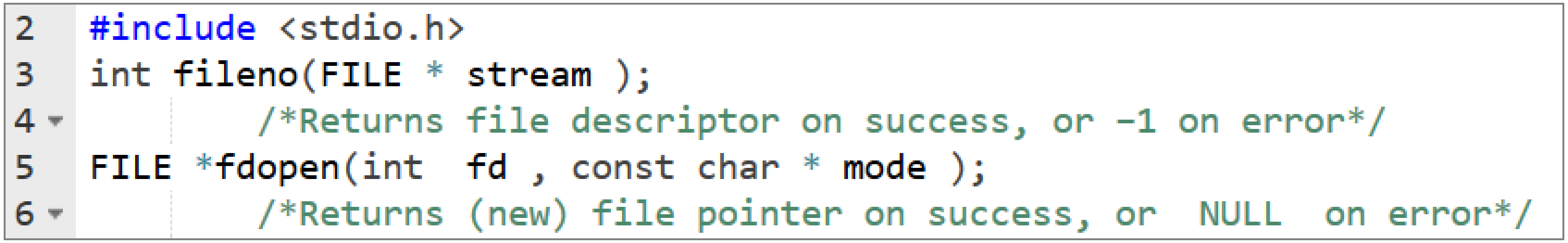


```c
#include <unistd.h>
void sync(void);
```

- It causes all kernel buffers containing modified data including metadata to be flushed to disk.
- Call returns only after syncing.

Mixing Library Functions and I/O Sys Calls

- Sys calls: open(). read(), write(), close()
  - Work with file descriptors
- Library functions: fopen(), fprintf(), fscanf(), fclose(), …
  - Work with FILE streams



```
#include <stdio.h>
int fileno(FILE * stream );
        /*Returns file descriptor on success, or -1 on error*/
FILE *fdopen(int  fd , const char * mode );
        /*Returns (new) file pointer on success, or  NULL  on error*/
```

  - Given a stream, *fileno*() returns the corresponding *fd*
  - Given a file descriptor, *fdopen()* creates corresponding FILE stream.
- *fdopen()* is useful while dealing with pipes and sockets.

BITS Pilani
Pilani Campus

Process Environment (R1:Ch6)

Processes and Programs

- A process is an instance of an executing program.
- A program is file containing a range of information that describes how to construct a process at run time.
  - A program may be used to run many processes or many processes may be running the same program.
  - */bin/ls* is a program. When it is run on the shell, a process needs to be created to run the program.
- A process is an abstract entity defined by the kernel to which system resources are allocated in order to execute a program.

Program

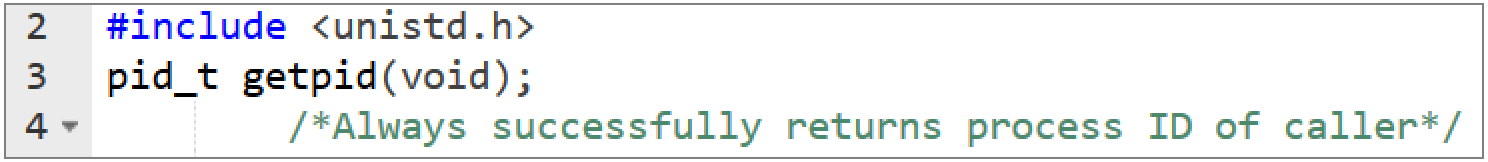
- A program includes a range of info for the Kernel
  - Binary header
    - Executable and Linking Format (ELF). This header gives the metainformation about the rest of the part in the executable file.
  - Machine language instructions
  - Program entry-point address
  - Data
  - Symbol and relocation tables
  - Shared library and dynamic linking information
  - Other information

Kernel View of the Process

- A process consists of
  - user-space memory for
    - holding program code
    - Variables used by that code
  - Kernel data structures that maintain info about the state of the process.
    - Various Ids such as PID, UID, GID, process group id etc
    - Virtual memory tables
    - Table of open file descriptors
    - Info about signal delivery and handling
    - Process resource usages and limits
    - Current working directory
    - etc

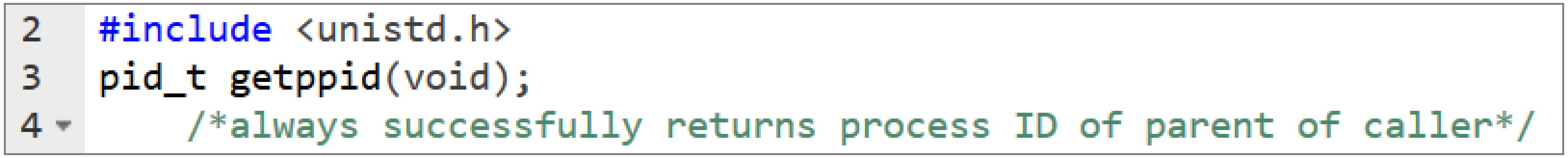
Process ID (pid) and Parent Process ID (ppid)

- Pid is s positive integer that uniquely identifies the process on the system.



```
#include <unistd.h>
pid_t getpid(void);
        /*Always successfully returns process ID of caller*/
```

  - getpid() call returns the process id (pid) of the calling process.
  - Kernel incrementally assigns the pids up to the limit of 32767.
- Each process has a parent- the process that created it.
  - All the processes except the init process have parent.
  - *pstree* – command to visualize the process hierarchy.
- If a process’s parent terminates, child is adopted by *init* process.

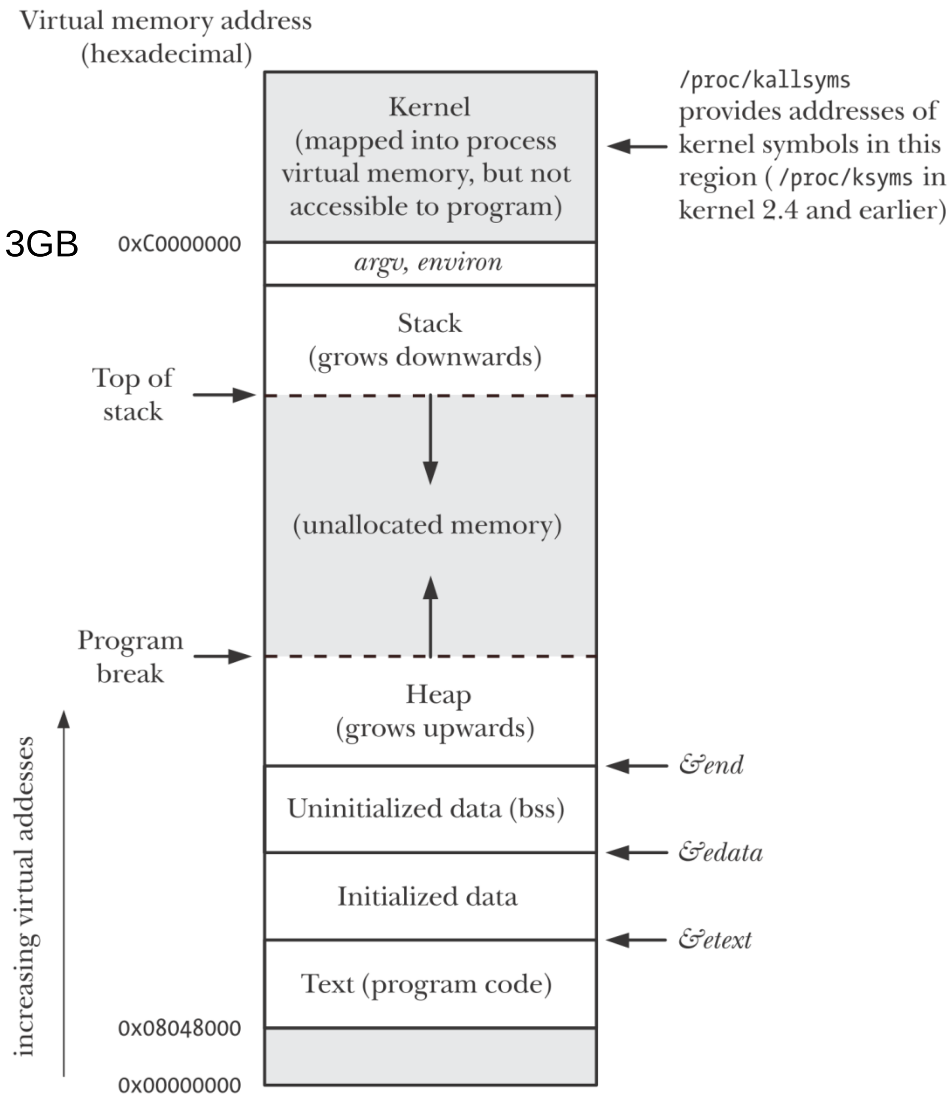


```
#include <unistd.h>
pid_t getppid(void);
    /*always successfully returns process ID of parent of caller*/
```

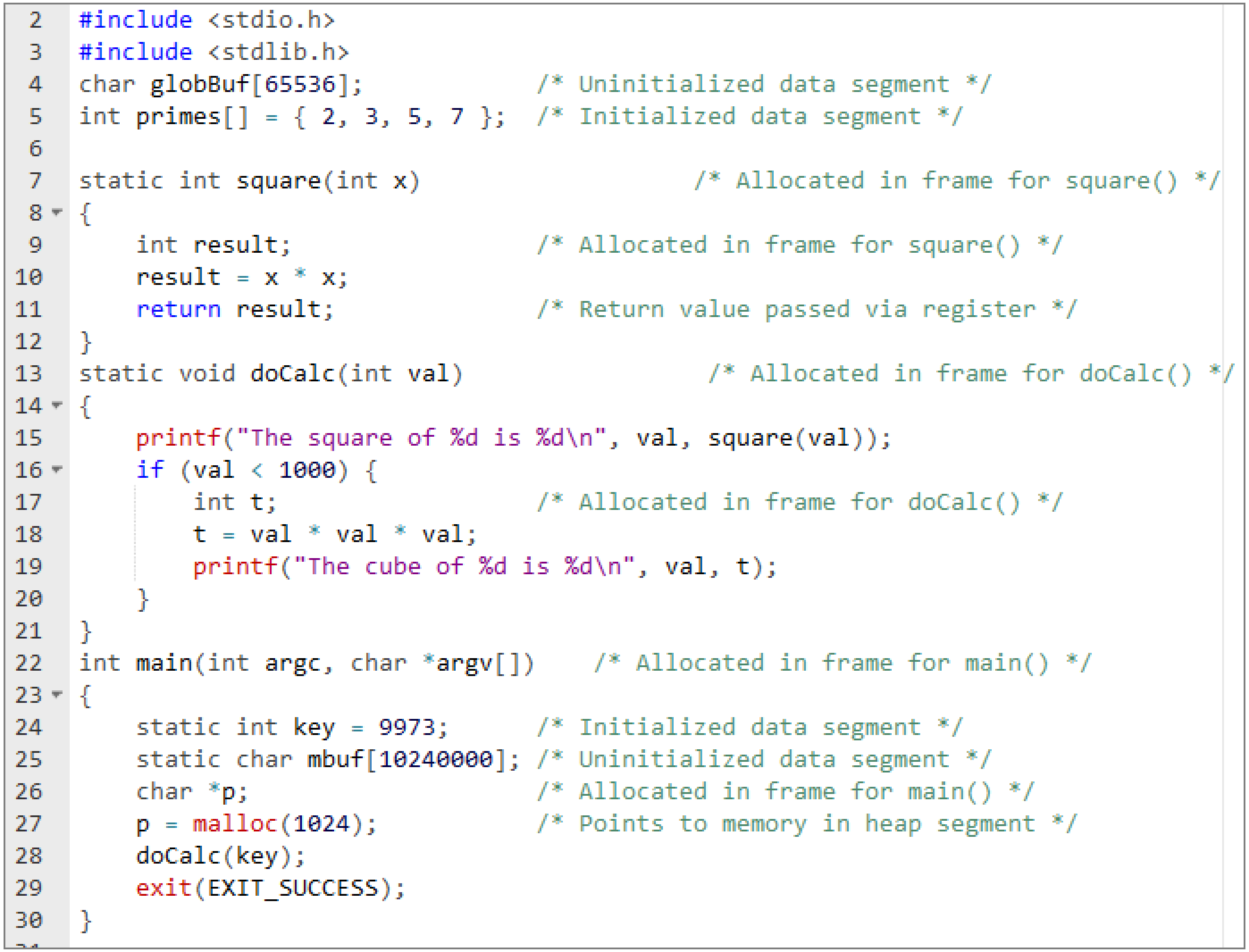
Memory Layout of a Process

- Memory allocated to each process composed of several sections/segments:
  - Text segment
    - Machine language instructions.
    - Read only and shared.
  - Initialized data segment
    - Global and static variables that are explicitly inialized.
  - Uninitialized data segment
    - Global and static variables that are not explicitly initialized.
    - Memory allocated during program loading.
  - Stack
    - One stack frame is allocated for each currently called function.
  - Heap
    - For dynamic allocation of memory

Memory Layout of a Process


Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
3GB
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
0x08048000
0x00000000
increasing virtual addresses

Program Variables in Process Memory

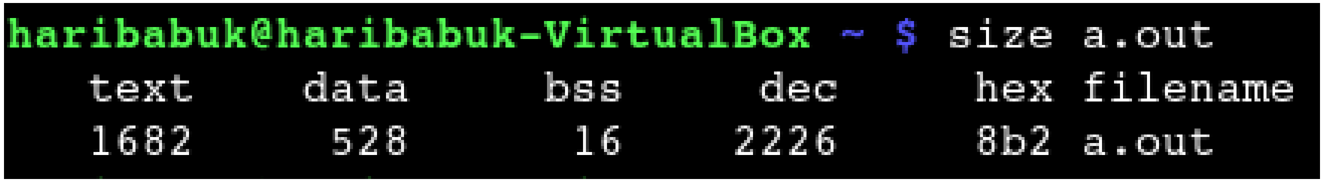


```c
#include <stdio.h>
#include <stdlib.h>
char globBuf[65536];            /* Uninitialized data segment */
int primes[] = { 2, 3, 5, 7 };  /* Initialized data segment */

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);
}
```

Source: R1: Listing 6-1

size command

- Size command lists the size of sections in an object file in bytes.



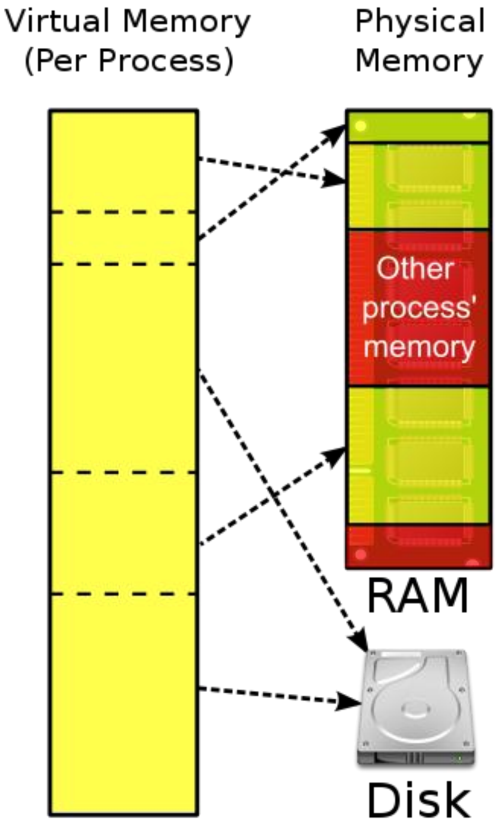
```
haribabuk@haribabuk-VirtualBox ~ $ size a.out
   text    data     bss     dec     hex filename
   1682     528      16    2226     8b2 a.out
```

Virtual Memory Management

- Linux employs virtual memory management technique for efficient use of Ram and CPU by taking advantage of *locality of reference*.
  - *Spatial locality*: tendency of a program to access the same memory addresses which are near those which were recently accessed.
    - Sequential processing of instructions and data structures.
  - *Temporal locality*: tendency to access the same instructions in the near future which it accessed in the recent past.
    - Processing loops.
- By locality of reference, it is possible to execute a program with only a part of its address space in RAM.

Virtual Memory

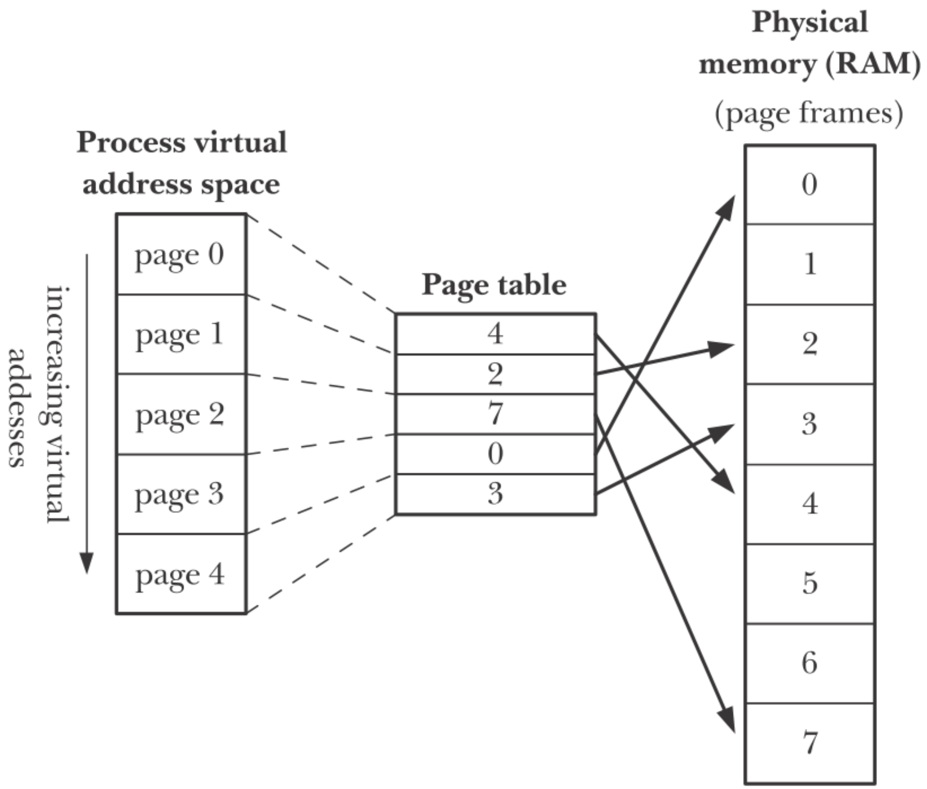
- Splits the program memory into fixed-size units calls pages. Correspondingly RAM is divided into page frames of the same size.
    - *Resident set*: set of program memory pages that are in RAM. Rest are in swap-area in the disk.
- When a process references a page that is not currently resident in RAM, a *page-fault* occurs.
    - Kernel suspends the execution of the process and loads the page from swap area to RAM.



Source: wikipedia

Page Tables

- Kernel maintains a page table for each process.
- Maps location of each page in process’s virtual address space into location of page on RAM or on disk.

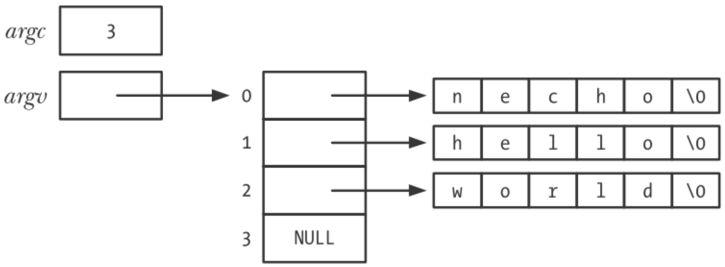

Physical memory (RAM) (page frames)
Process virtual address space
increasing virtual addresses
page 0
page 1
page 2
page 3
page 4
Page table
4
2
7
0
3
0
1
2
3
4
5
6
7

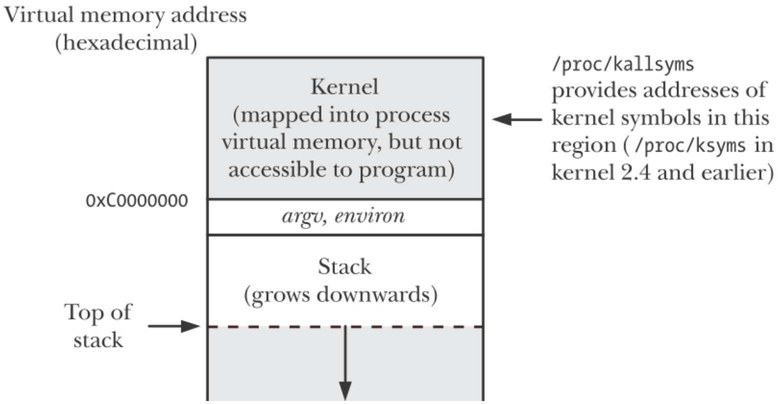
Process Address Space

- On 32-bits systems out of 4GB address space, 3GB is available for the current process.
- Not all address ranges in the process's virtual address space is used. Such addresses (pages) are not mapped in page table.
- Accessing an address for which there is no corresponding page table entry, results in segmentation fault (SIGSEGV signal).
- Valid virtual address ranges change over the life time of a process:
  - Stack grows beyond limits previously reached.
  - malloc()
  - Shared memory (shmat or shmdt)
  - Memory mapping or unmapping

Command-line Arguments

- Command line arguments are entered along with the executable.
- When the program is loaded, these arguments are stored just above the stack.
- They are available to main function via *argc* and *argv*.

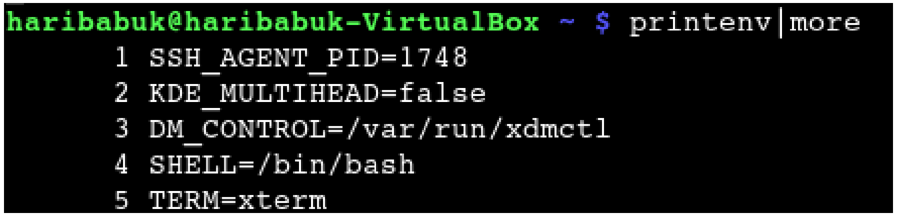




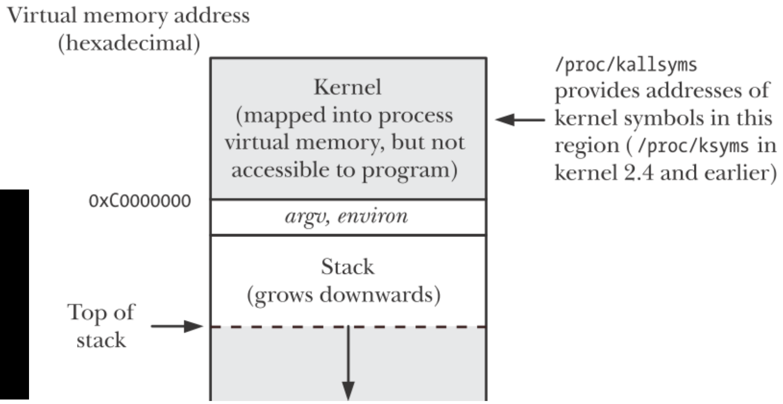
Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)

Environment Variables

- Each process has an associated array of strings called the environment list.
  - Each is a name=value pair.
  - Child inherits the parent's environment at the time of fork().
- Shell uses environment variables to pass on certain info to children. It is one-way and once only.
  - Environment is stored just above the stack in the process memory.

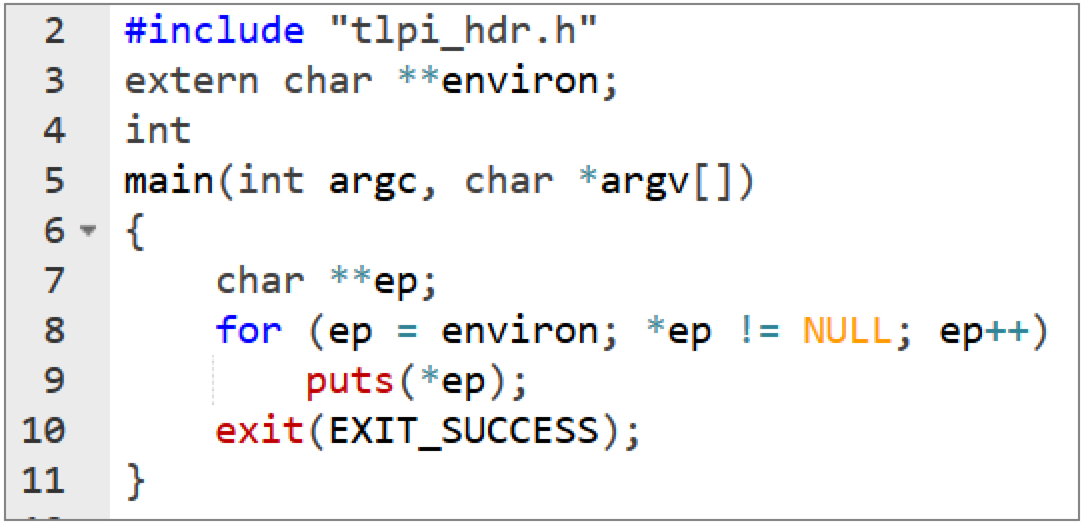


```
haribabuk@haribabuk-VirtualBox ~ $ printenv|more
      1 SSH_AGENT_PID=1748
      2 KDE_MULTIHEAD=false
      3 DM_CONTROL=/var/run/xdmctl
      4 SHELL=/bin/bash
      5 TERM=xterm
```


Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack

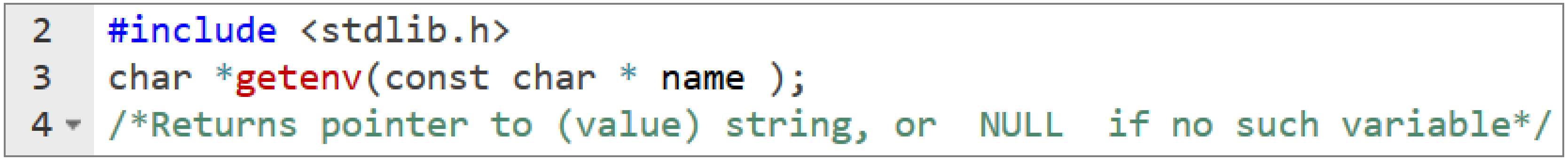
Accessing Environment in Program

- Environment list can be accessed using a global variable *environ*.



```c
#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);
}
```

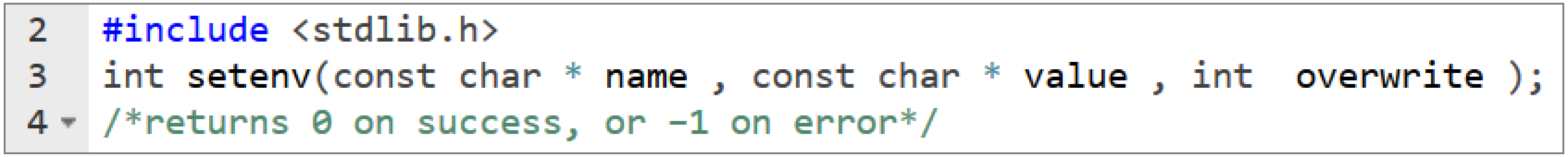
- *getenv()* returns the value for given name.



```c
#include <stdlib.h>
char *getenv(const char * name );
/*Returns pointer to (value) string, or  NULL  if no such variable*/
```

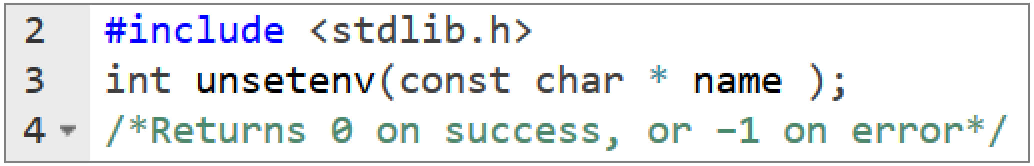
Modifying the Environment

- setenv(): add or modify the environment list.



```
#include <stdlib.h>
int setenv(const char * name , const char * value , int  overwrite );
/*returns 0 on success, or -1 on error*/
```

- unsetenv (): to remove an environment variable.

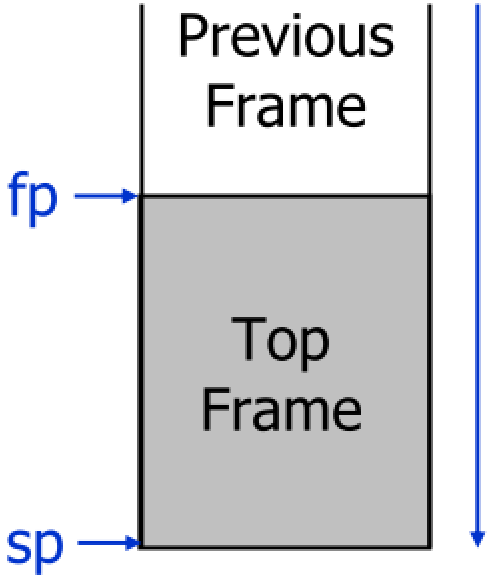


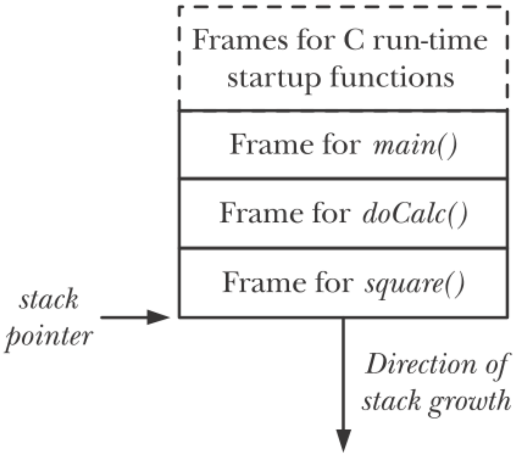
```
#include <stdlib.h>
int unsetenv(const char * name );
/*Returns 0 on success, or -1 on error*/
```

- By setting environ=NULL, entire environment list is erased.

Stack and Stack Frames

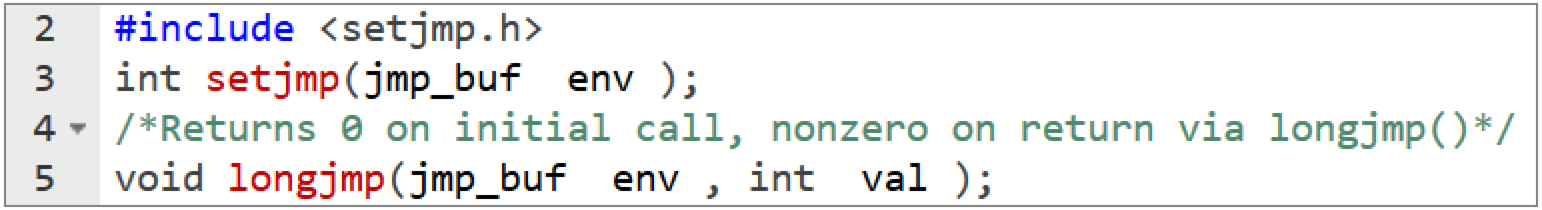
- Stack grows downwards towards the heap.
  - A special purpose register, *stack pointer*, tracks the current top of the stack.
  - Kernel stack is a per-process memory region maintained in kernel to store function calls during sys calls.
- Stack frame contains
  - Function arguments and local variables
  - Call linkage information such as program counter.





Nonlocal Goto: setjmp() & longjmp()

- Nonlocal goto refers to jumping out of the current function into the callee function or beyond that.
- Useful in error handling.
- Calling *setjmp()* establishes a target for a later jump performed by *longjmp()*.

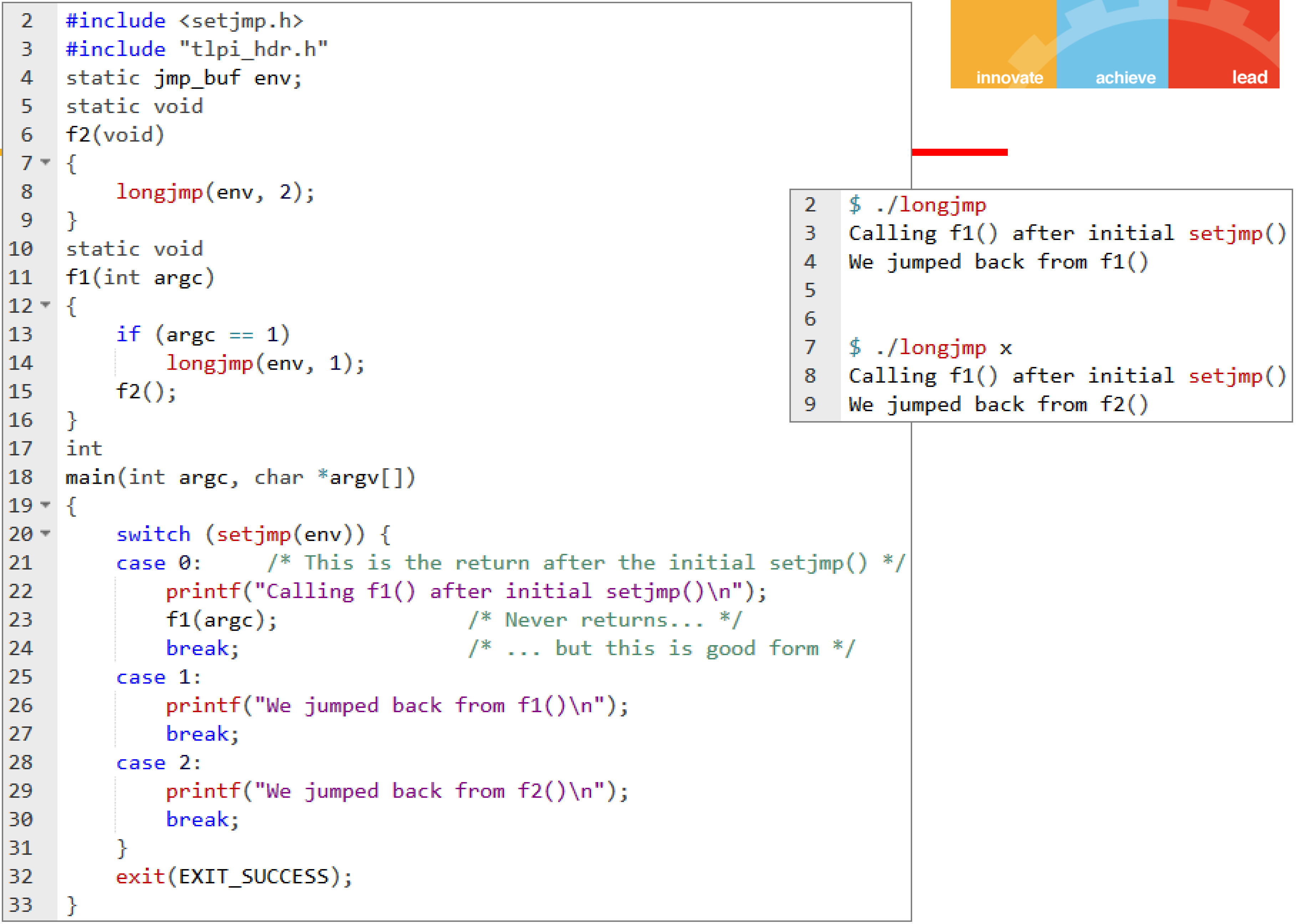


```c
#include <setjmp.h>
int setjmp(jmp_buf  env );
/*Returns 0 on initial call, nonzero on return via longjmp()*/
void longjmp(jmp_buf  env , int  val );
```

  - Initial run of *setjmp()* saves the various information about the current process environment and returns 0.
  - When *longjmp()* is called later, *setjmp()* returns with *val* supplied while calling *longjmp()*.
  - By using different values for *val, longjmp()* locations can be distinguished.

longjmp()

- *env* stores a copy of the PC (program counter) and SP (stack pointer).
- When *longjmp()* is called,
  - it unwinds the stack by resetting SP to the value saved in *env*.
  - It also resets the program counter register to the vales saved in *env.*
- Avoid *setjmp()* and *longjmp()* where possible.

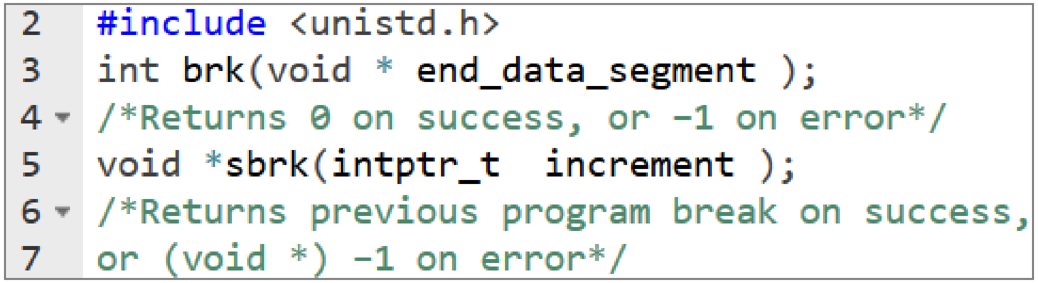


```c
#include <setjmp.h>
#include "tlpi_hdr.h"
static jmp_buf env;
static void
f2(void)
{
    longjmp(env, 2);
}
static void
f1(int argc)
{
    if (argc == 1)
        longjmp(env, 1);
    f2();
}
int
main(int argc, char *argv[])
{
    switch (setjmp(env)) {
    case 0:     /* This is the return after the initial setjmp() */
        printf("Calling f1() after initial setjmp()\n");
        f1(argc);               /* Never returns... */
        break;                  /* ... but this is good form */
    case 1:
        printf("We jumped back from f1()\n");
        break;
    case 2:
        printf("We jumped back from f2()\n");
        break;
    }
    exit(EXIT_SUCCESS);
}
```

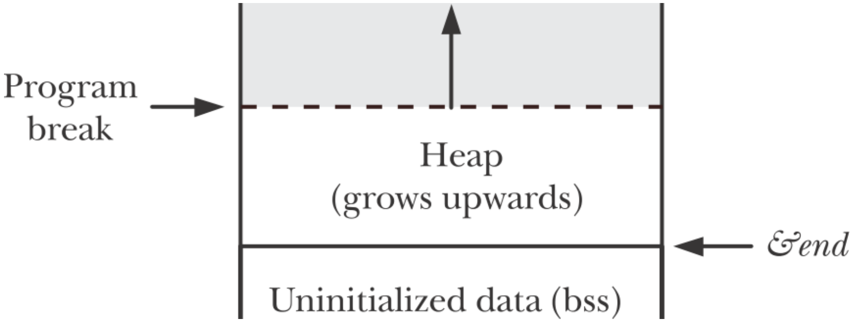
```
$ ./longjmp
Calling f1() after initial setjmp()
We jumped back from f1()


$ ./longjmp x
Calling f1() after initial setjmp()
We jumped back from f2()
```

Heap

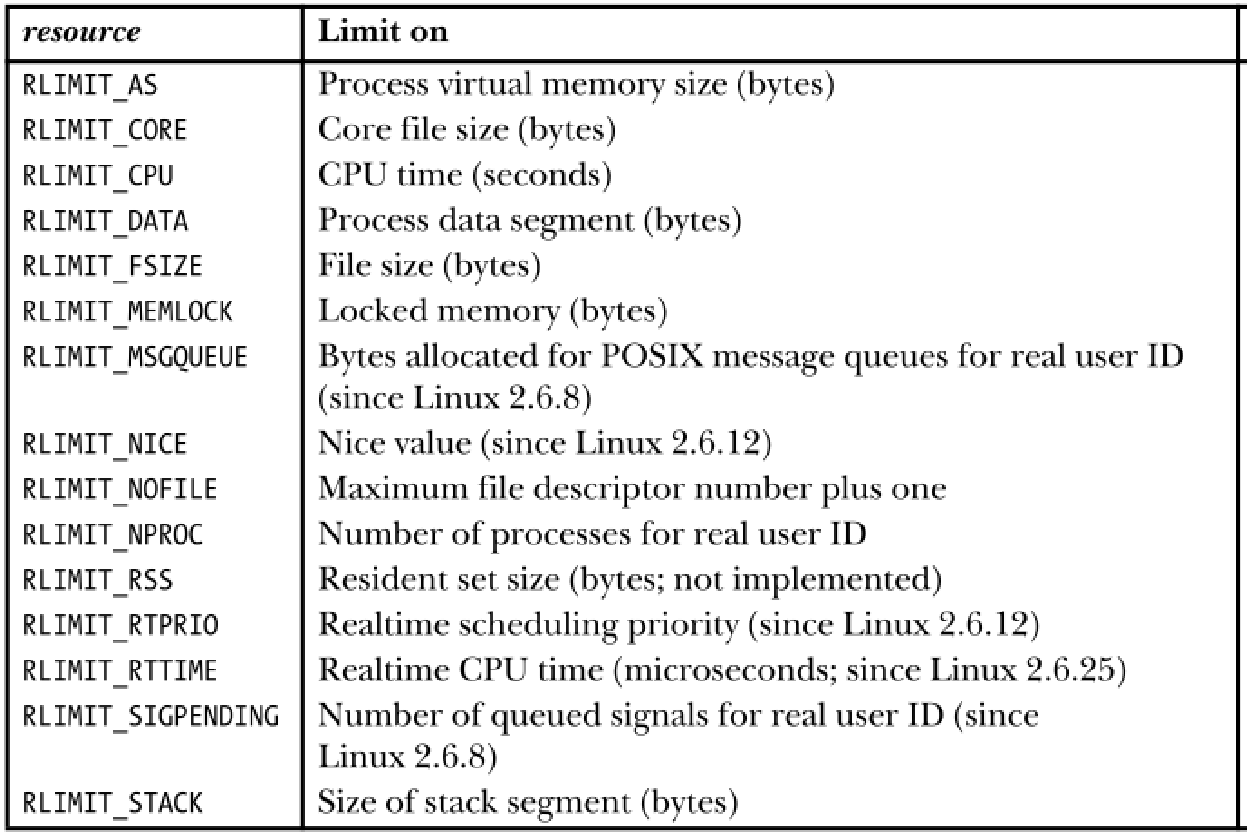
- Current limit of the heap is referred as the *program break*.
  - Initially *program break* is same as the *end.*
- A process can allocate memory by increasing the size of the heap.
- brk() or sbrk() calls can be used to increase the program break. malloc() library call uses these system calls.
  - After the program break is increased, the process may access any address in thee newly allocated area but no physical pages are allocated yet.
  - Kernel allocates whenever process references those addresses.



```c
#include <unistd.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*/
```



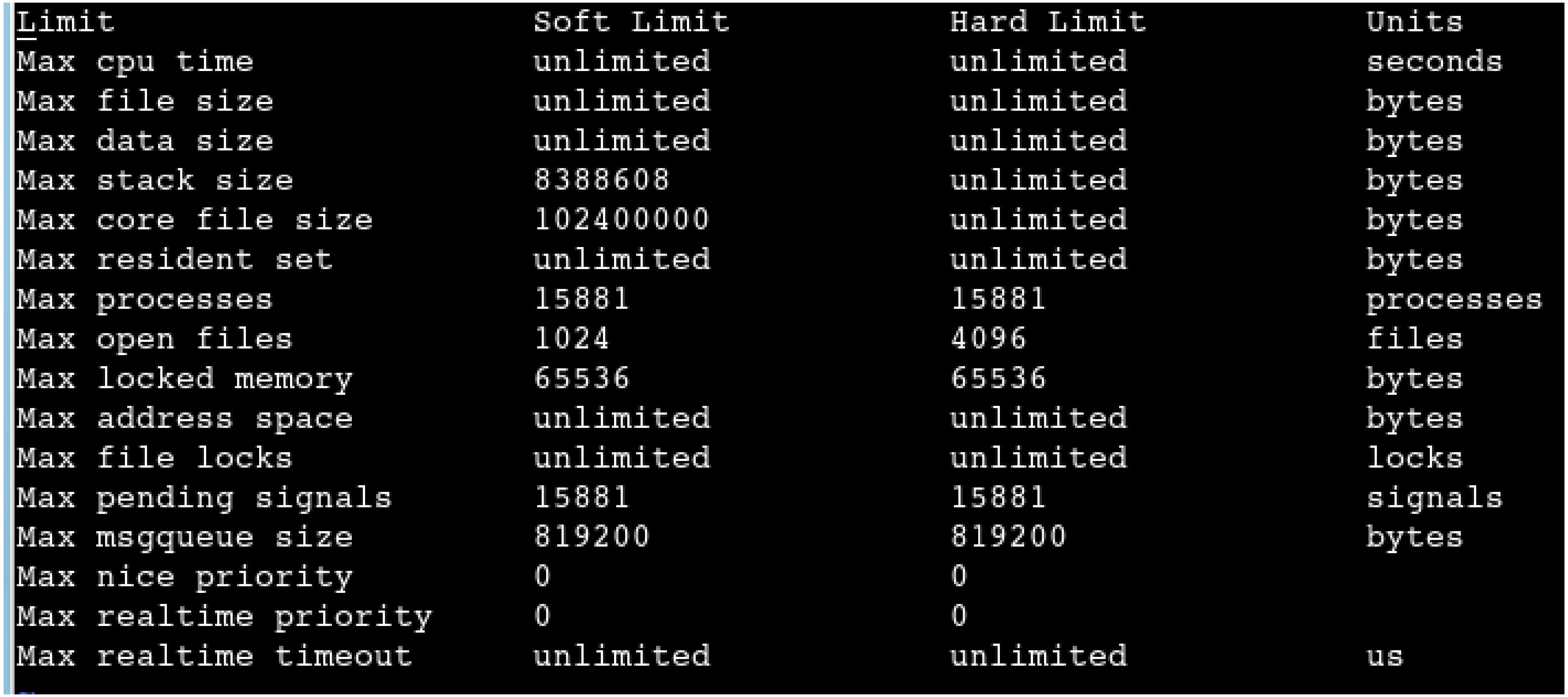
Resource Limits



resource	**Limit on**
RLIMIT_AS	Process virtual memory size (bytes)
RLIMIT_CORE	Core file size (bytes)
RLIMIT_CPU	CPU time (seconds)
RLIMIT_DATA	Process data segment (bytes)
RLIMIT_FSIZE	File size (bytes)
RLIMIT_MEMLOCK	Locked memory (bytes)
RLIMIT_MSGQUEUE	Bytes allocated for POSIX message queues for real user ID (since Linux 2.6.8)
RLIMIT_NICE	Nice value (since Linux 2.6.12)
RLIMIT_NOFILE	Maximum file descriptor number plus one
RLIMIT_NPROC	Number of processes for real user ID
RLIMIT_RSS	Resident set size (bytes; not implemented)
RLIMIT_RTPRIO	Realtime scheduling priority (since Linux 2.6.12)
RLIMIT_RTTIME	Realtime CPU time (microseconds; since Linux 2.6.25)
RLIMIT_SIGPENDING	Number of queued signals for real user ID (since Linux 2.6.8)
RLIMIT_STACK	Size of stack segment (bytes)

- The RLIMIT_DATA limit specifies the maximum size of initialized data, uninitialized data, and heap segments together.
- Attempts to extend the program break beyond this limit fail with the error ENOMEM .

/proc/self/limits



```
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        102400000            unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             15881                15881                processes
Max open files            1024                 4096                 files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       15881                15881                signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

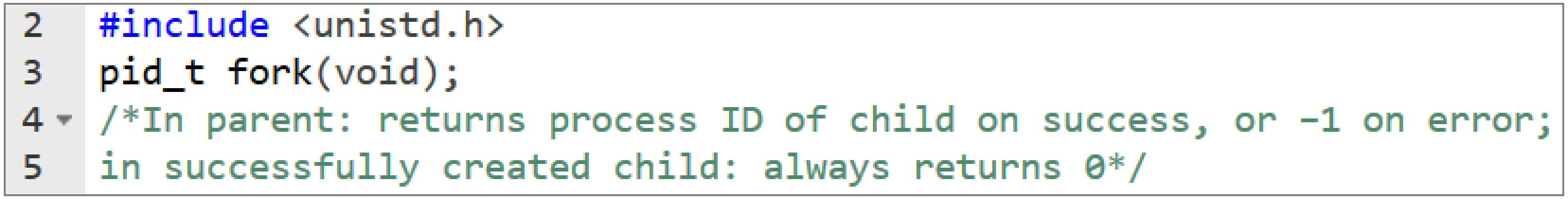
BITS Pilani

Pilani Campus

Process Creation (R1: Ch24)

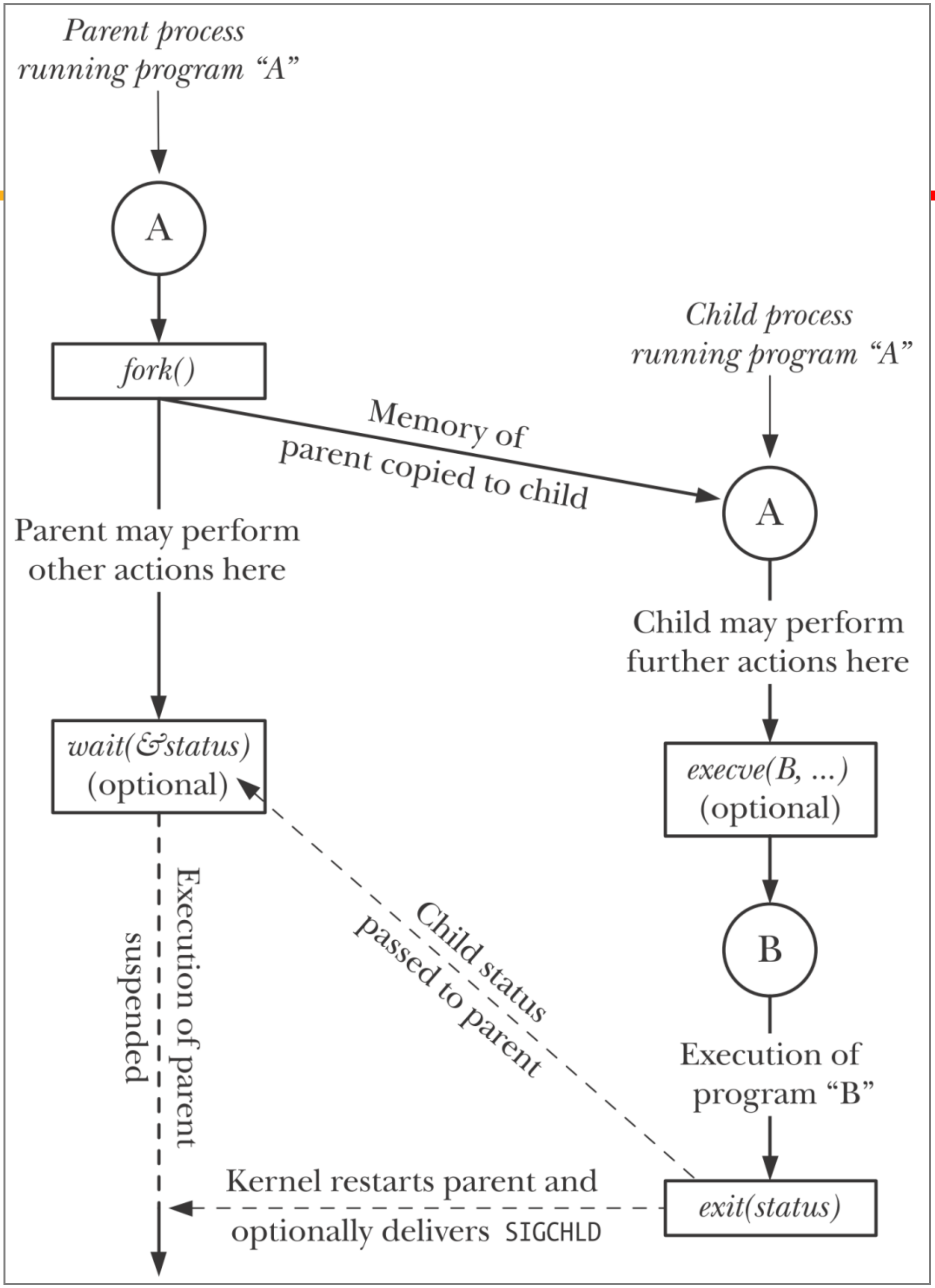
Process Creation

- An existing process can create a new process by calling fork function. The new process created by fork is called *child process*.

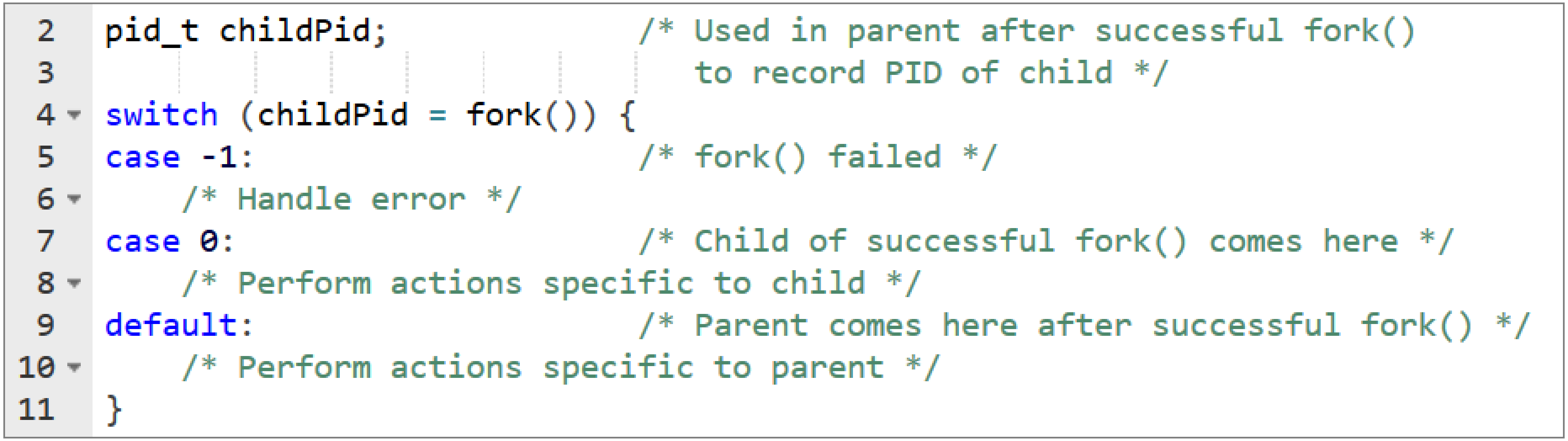


```
#include <unistd.h>
pid_t fork(void);
/*In parent: returns process ID of child on success, or -1 on error;
in successfully created child: always returns 0*/
```

  - This function is called once but returns twice.
    - The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
  - The child is a copy of the parent. The child gets a copy of the parent's data section, heap, and the stack. Memory is copied not shared.
  - The parent and the child share the text segment.


Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

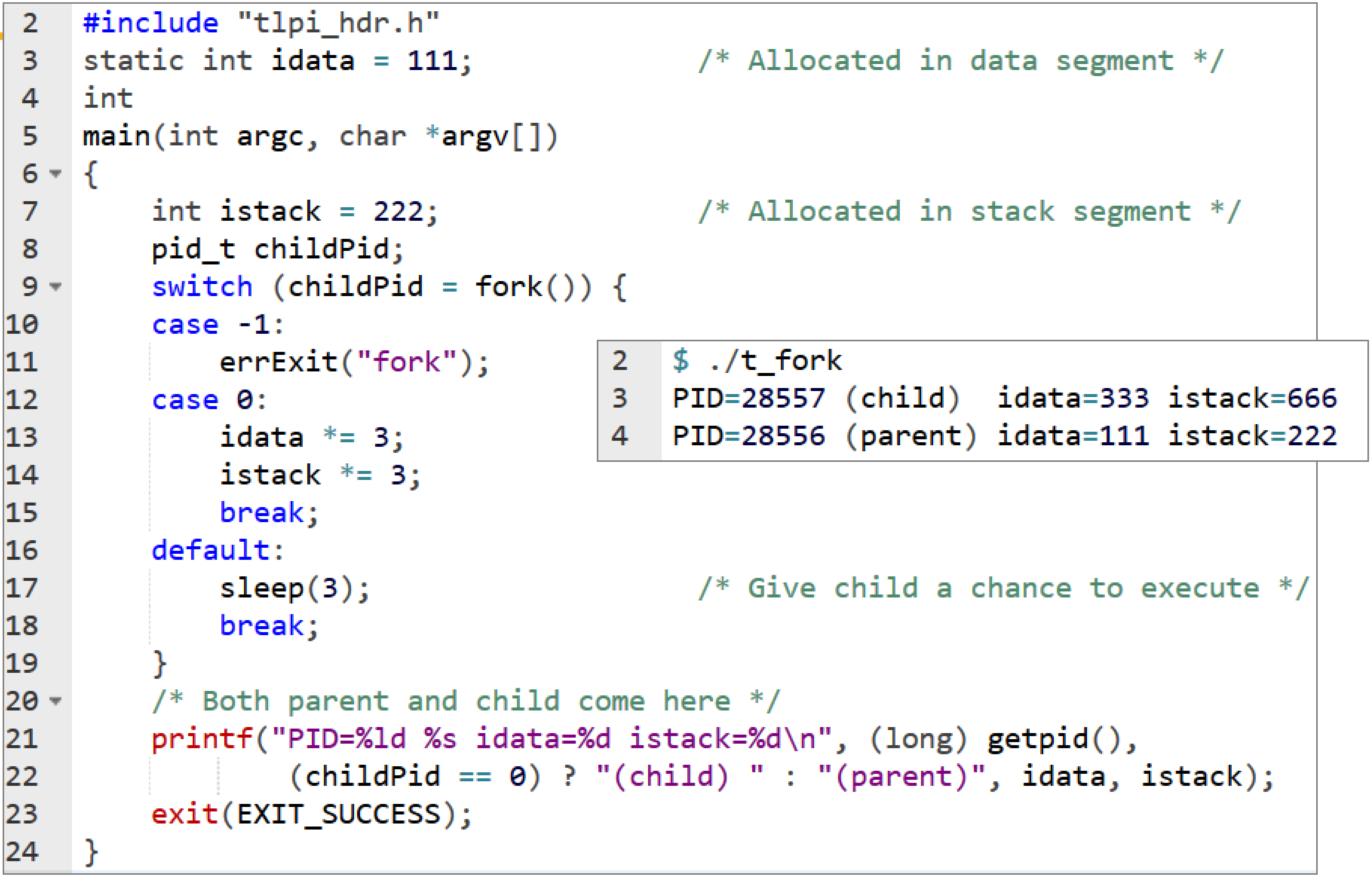
PIDs



```c
pid_t childPid;             /* Used in parent after successful fork()
                               to record PID of child */
switch (childPid = fork()) {
case -1:                    /* fork() failed */
    /* Handle error */
case 0:                     /* Child of successful fork() comes here */
    /* Perform actions specific to child */
default:                    /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

- Within the code of the program, child and parent can be distinguished by the return value of *fork()*.
  - In parent return value>0
  - In child return value==0
- In general, we never know whether the child starts executing before the parent or vice versa.
- To synchronize child and parent, some form of interprocess communication is required.

fork() demo



```c
#include "tlpi_hdr.h"
static int idata = 111;             /* Allocated in data segment */
int
main(int argc, char *argv[])
{
    int istack = 222;               /* Allocated in stack segment */
    pid_t childPid;
    switch (childPid = fork()) {
    case -1:
        errExit("fork");
    case 0:
        idata *= 3;
        istack *= 3;
        break;
    default:
        sleep(3);                   /* Give child a chance to execute */
        break;
    }
    /* Both parent and child come here */
    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);
    exit(EXIT_SUCCESS);
}
```

```
$ ./t_fork
PID=28557 (child)  idata=333 istack=666
PID=28556 (parent) idata=111 istack=222
```

Q&A

Next Time

- Please read through R1: chapters 6,7, 24-28

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You