Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

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

innovate achieve lead

- 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.

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

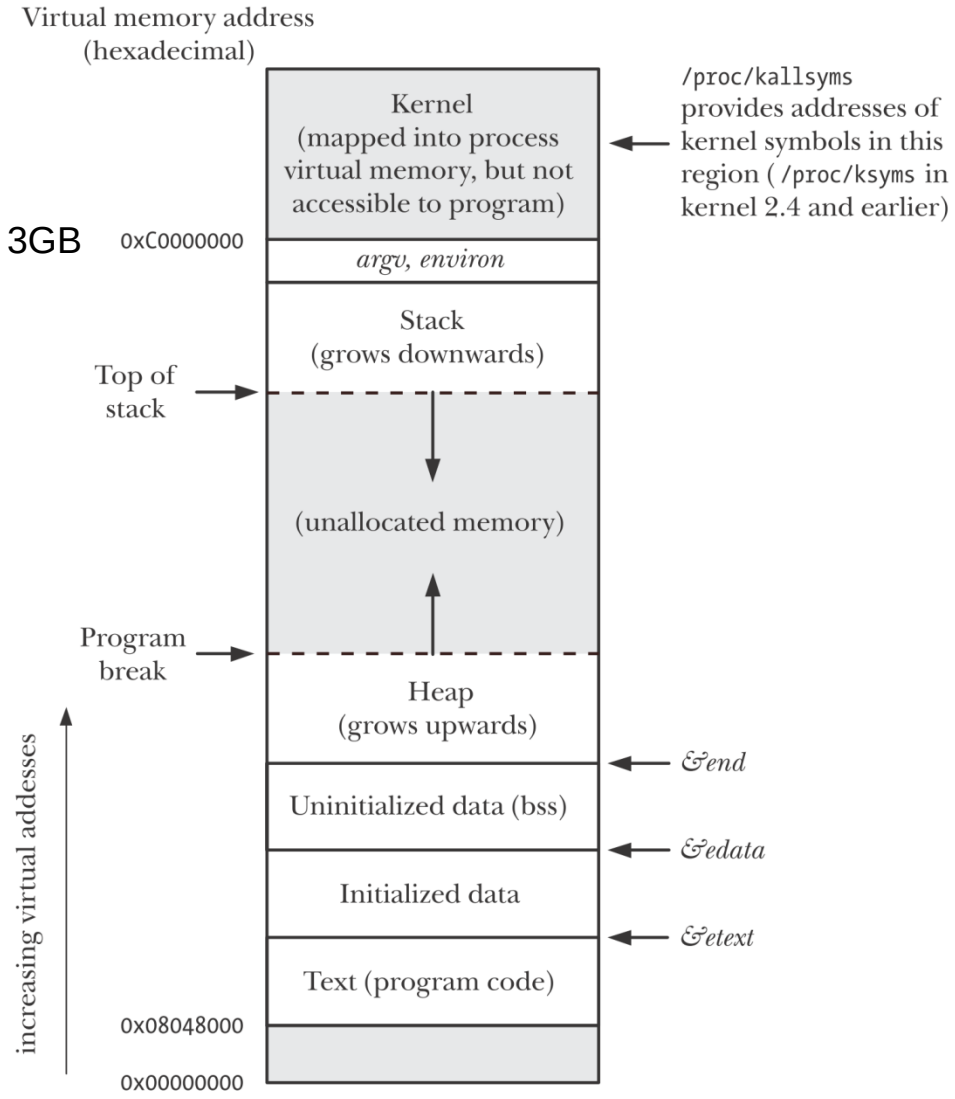
  - getpid() call returns the process id (pid) of the calling process.
  - Kernel incrementally assigns the pids up to the limit of 32767.
- Each process has a parent- the process that created it.
  - All the processes except the init process have parent.
  - *pstree* – command to visualize the process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

```
2  #include <unistd.h>
3  pid_t getppid(void);
4      /*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



Program Variables in Process Memory

innovate achieve lead

```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.

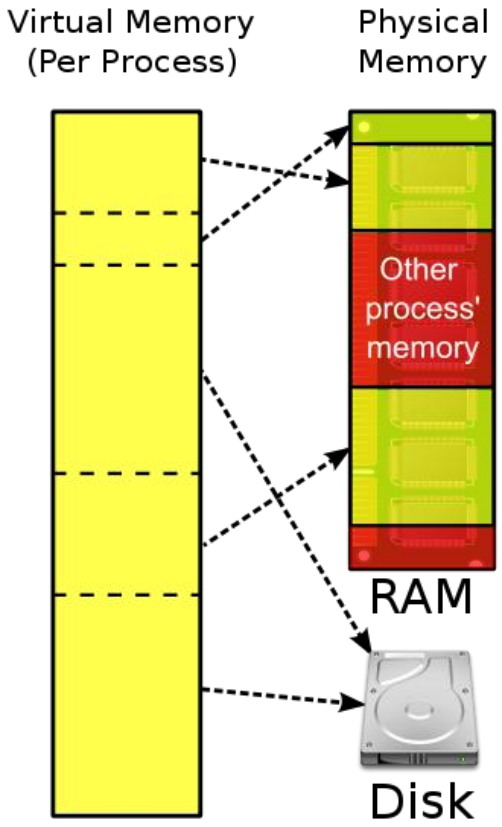
```
haribabuk@haribabuk-VirtualBox ~ $ size a.out
   text    data     bss     dec     hex filename
   1682     528      16    2226     8b2 a.out
```

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

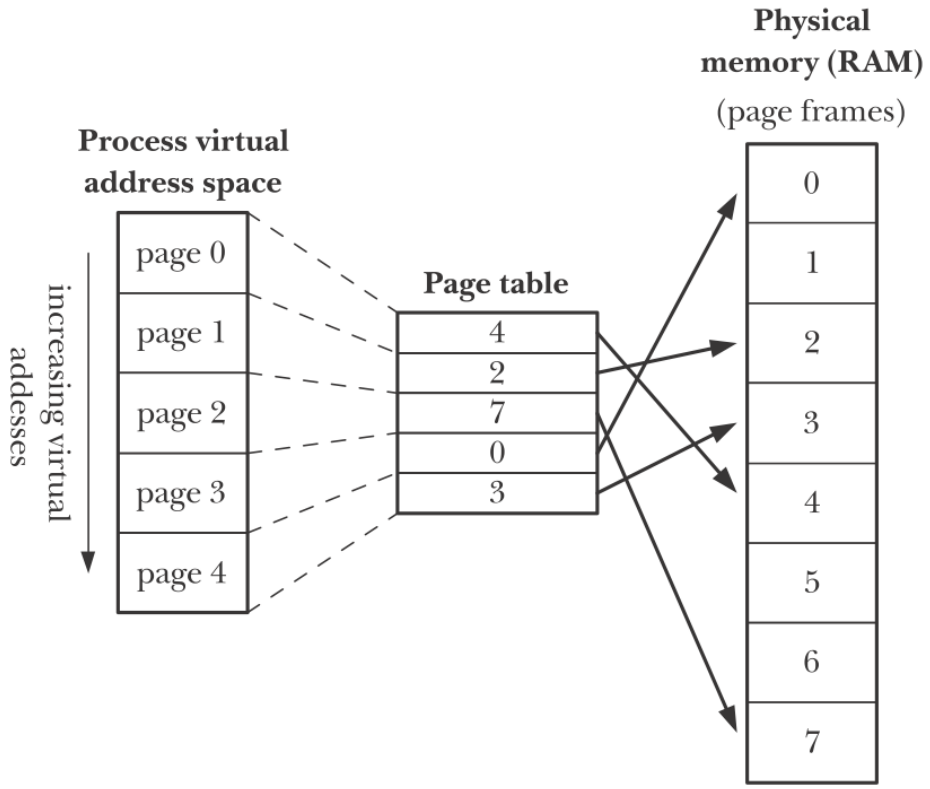
- Splits the program memory into fixed-size units calls pages. Correspondingly RAM is divided into page frames of the same size.
  - *Resident set*: set of program memory pages that are in RAM. Rest are in swap-area in the disk.
- When a process references a page that is not currently resident in RAM,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

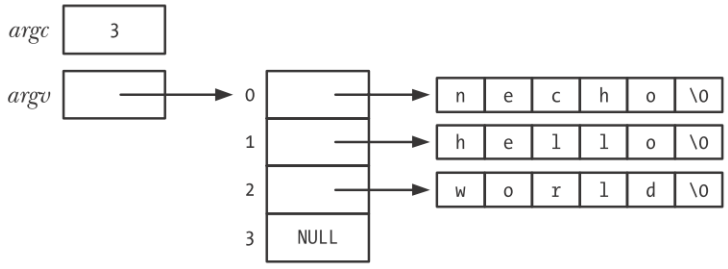


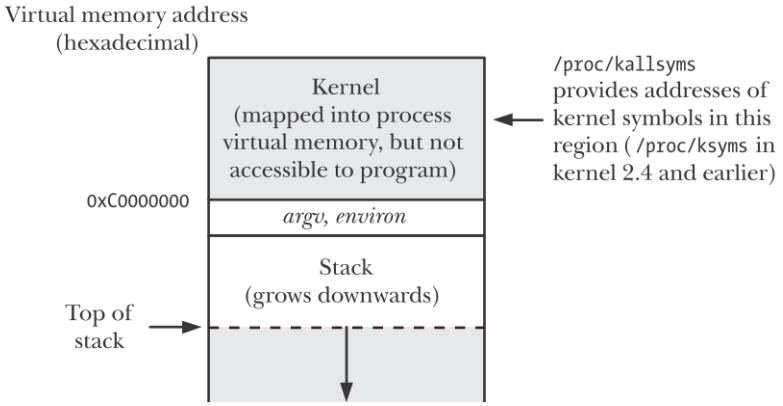
Process Address Space

- 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*.

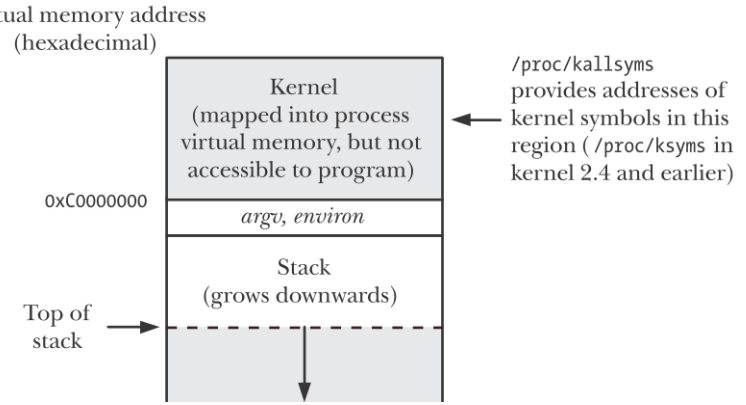




Environment Variables

- 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
```



Accessing Environment in Program

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

```
 2  #include "tlpi_hdr.h"
 3  extern char **environ;
 4  int
 5  main(int argc, char *argv[])
 6  {
 7      char **ep;
 8      for (ep = environ; *ep != NULL; ep++)
 9          puts(*ep);
10      exit(EXIT_SUCCESS);
11  }
```

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

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

Modifying the Environment

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

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

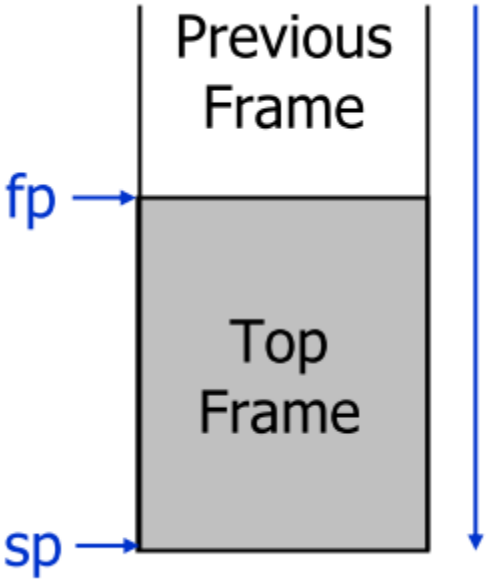
- unsetenv (): to remove an environment variable.

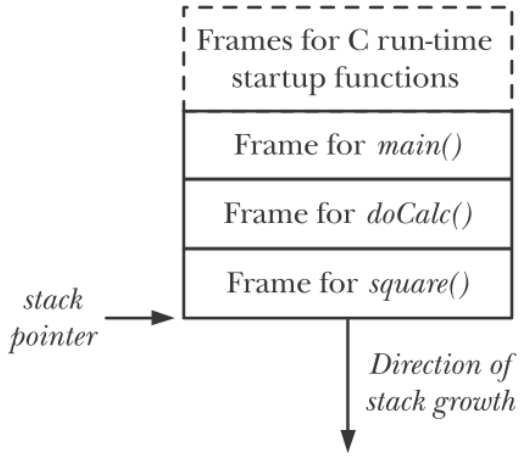
```
2  #include <stdlib.h>
3  int unsetenv(const char * name );
4  /*Returns 0 on success, or -1 on error*/
```

- 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()*.

```
2  #include <setjmp.h>
3  int setjmp(jmp_buf  env );
4  /*Returns 0 on initial call, nonzero on return via longjmp()*/
5  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);
}
```

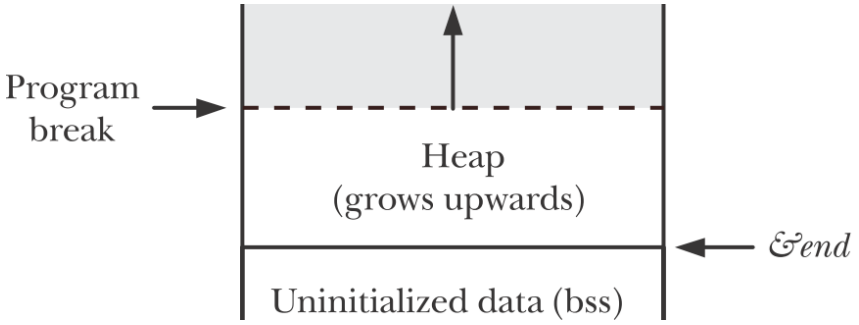
```
$ ./longjmp
Calling f1() after initial setjmp()
We jumped back from f1()


$ ./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.

```
2  #include <unistd.h>
3  int brk(void * end_data_segment );
4  /*Returns 0 on success, or -1 on error*/
5  void *sbrk(intptr_t  increment );
6  /*Returns previous program break on success,
7  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

innovate achieve lead

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

innovate achieve lead

BITS Pilani

Pilani Campus

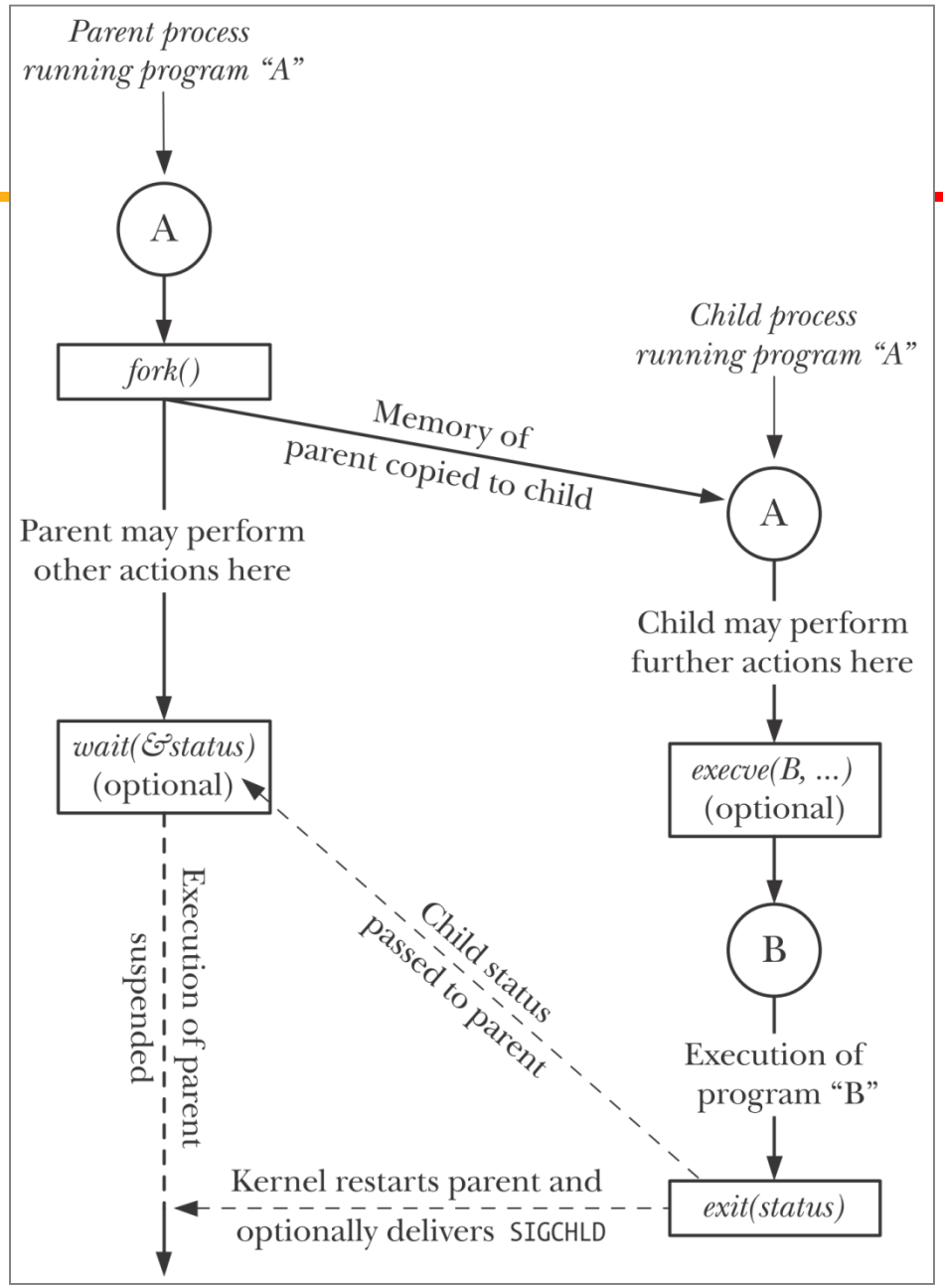
Process Creation (R1: Ch24)

Process Creation

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

```
2  #include <unistd.h>
3  pid_t fork(void);
4  /*In parent: returns process ID of child on success, or -1 on error;
5  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)
Child status
passed to parent
Execution of parent
suspended
B
Execution of
program "B"
Kernel restarts parent and
optionally delivers SIGCHLD
exit(status)

PIDs


```
2   pid_t childPid;             /* Used in parent after successful fork()
3                                  to record PID of child */
4   switch (childPid = fork()) {
5   case -1:                    /* fork() failed */
6       /* Handle error */
7   case 0:                     /* Child of successful fork() comes here */
8       /* Perform actions specific to child */
9   default:                    /* Parent comes here after successful fork() */
10      /* Perform actions specific to parent */
11  }
```

- 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.

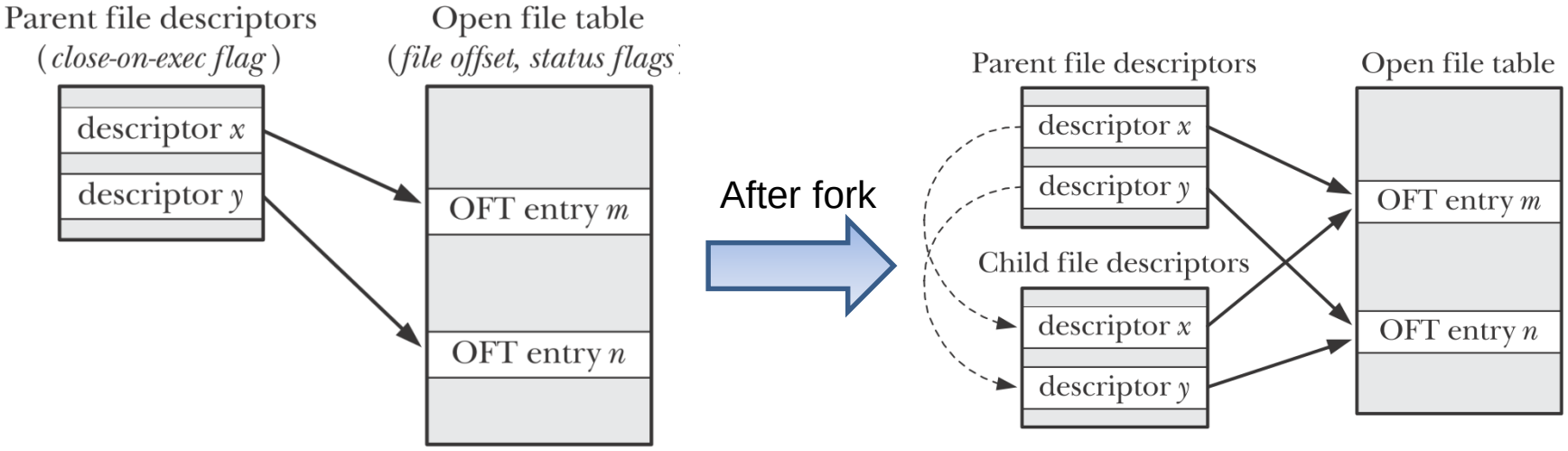
fork() demo


```
2   #include "tlpi_hdr.h"
3   static int idata = 111;                 /* Allocated in data segment */
4   int
5   main(int argc, char *argv[])
6   {
7       int istack = 222;                   /* Allocated in stack segment */
8       pid_t childPid;
9       switch (childPid = fork()) {
10      case -1:
11          errExit("fork");
12      case 0:
13          idata *= 3;
14          istack *= 3;
15          break;
16      default:
17          sleep(3);                       /* Give child a chance to execute */
18          break;
19      }
20      /* Both parent and child come here */
21      printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
22              (childPid == 0) ? "(child) " : "(parent)", idata, istack);
23      exit(EXIT_SUCCESS);
24  }
```

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

File Sharing Between Parent and Child

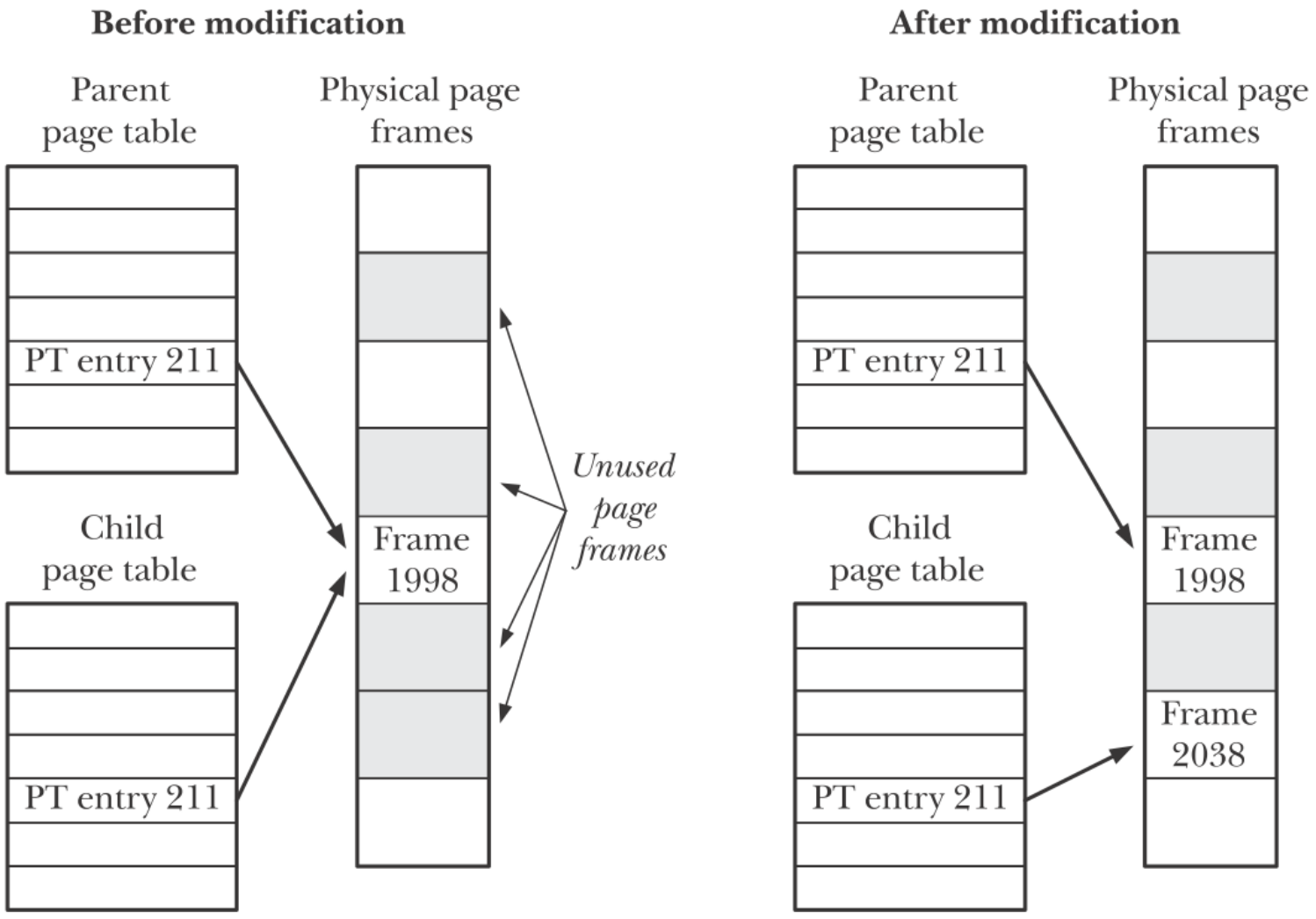
- When a *fork()* is performed, the child receives duplicates of all of the parent's file descriptors.
- Open file attributes are shared between the parent and the child.
  - Sharing the file offset:
    - parent and child do not over write each other's output.
    - but the outputs may be randomly intermingled. IPC required.



fork(): Copy-on-write

- Old UNIX implementations used to literally make copy of the parents memory.
  - This is wasteful.
- Modern UNIX implementations use two techniques to avoid it.
  - Mark text segment as read-only. Build child's page table such away that pages in text segment point to the same physical memory pages as those of parent.
  - Data, Heap, Stack segments: *copy on write*.
    - Mark these segments as read-only. Build child's page tables such a way that they point to the physical pages used by parent.
    - Kernel traps any attempt to modify the page. Makes a duplicate of the page and modifies the page table of the modifying process to point to this page.

Copy on Write: Page Tables



Process's Memory Footprint

- Process's memory footprint is the range of virtual memory pages used by the process.
- Controlling process's memory footprint:
  - Executing functions known for memory leaks. Execute such functions within a child process. That won't affect the parent.

```
2  /*from procexec/footprint.c*/
3  pid_t childPid;
4  int status;
5  childPid = fork();
6  if (childPid == -1)
7      errExit("fork");
8  if (childPid == 0)                  /* Child calls func() and */
9      exit(func(arg));                /* uses return value as exit status */
10 /* Parent waits for child to terminate. It can determine the
11    result of func() by inspecting 'status'. */
12 if (wait(&status) == -1)
13     errExit("wait");
```

vfork() system call

- In early UNIX implementations, *vfork()* was proposed as efficient version of fork()system call.
  - With the copy-on-write implementation, eliminated need for *vfork()*.
- *vfork()* is designed to be used where the child performs an immediate *exec()* call.
- Differences between vfork() and fork()
  - No duplication of page tables for the child.
    - Child shares parent's memory until it calls *_exit()* or *exec()*
    - Changes made by the child will be visible to parent.
  - Execution of the parent is suspended until the child has performed an *exec() or _exit().*

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

vfork() demo

innovate achieve lead

```c
/*procexec/t_vfork.c*/
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    int istack = 222;
    switch (vfork()) {
    case -1:
        errExit("vfork");
    case 0:             /* Child executes first, in parent's memory space */
        sleep(3);                   /* Even if we sleep for a while,
                                       parent still is not scheduled */
        write(STDOUT_FILENO, "Child executing\n", 16);
        istack *= 3;                /* This change will be seen by parent */
        _exit(EXIT_SUCCESS);
    default:            /* Parent is blocked until child exits */
        write(STDOUT_FILENO, "Parent executing\n", 17);
        printf("istack=%d\n", istack);
        exit(EXIT_SUCCESS);
    }
}
```

```
$ ./t_vfork
Child executing /*Even though child slept, parent was not scheduled*/
Parent executing
istack=666
```

Race Conditions

- After fork(), it is indeterminate which process parent or child next has access to the CPU.
- Assuming a particular order of execution (e.g. parent and child) can lead to race conditions.
  - Race condition means that the result of execution will vary depending on the order of execution.
- A particular order can be achieved using either of the following
  - Signals
  - Pipes/message queues
  - Semaphores etc.

innovate achieve lead

BITS Pilani

Pilani Campus

Process Termination (R1: Ch25)

Process Termination

- A process may terminate in two ways
  - Abnormal termination
    - Cause by a signal such as SIGSEGV.
  - Normal termination
    - By calling _exit()

```
2  #include <unistd.h>
3  void _exit(int  status );
4  /*this function never returns*/
```

  - *_exit()* is a system call. *exit()* is a library call which invokes *_exit()*.
  - status is made available to the parent by the kernel. Only lower 8 bits of the *status* variable.
  - Generally status=0 means successfully completed.

Process Termination

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
- During both normal and abnormal termination of a process
  - Open file descriptors are closed and file locks (if any) are released.
  - Shared memory segments or memory mappings are released.
  - semadj value is adjusted.
  - Kernel releases the memory that it was using, and the like.
- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- Kernel stores the exit status, resource usage statistics for the parent to know.

exit() – Library Function


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

- During execution of exit()
  - Exit handlers are called.
  - *stdio* stream buffers are flushed.
  - _exit() system call is invoked.
- Exit handlers:

```
2   #include <stdlib.h>
3   int atexit(void (* func )(void));
4       /*Returns 0 on success, or nonzero on error*/
```

  - Exit handlers are useful for cleanup tasks at the time of termination.
  - *atexit()* adds a *func* to the list of functions to be called at the time of termination.
  - Functions are invoked in reverse order.
  - Child inherits the exit handlers of the parent.

innovate achieve lead

BITS Pilani

Pilani Campus

Monitoring Child Processes (R1: Ch26)

Monitoring Child Processes

- It is useful for the parent process to know when and how a child process got terminated
- wait() or waitpid() system call is used to know the status of the child.
- wait() system call:

```
2  #include <sys/wait.h>
3  pid_t wait(int * status );
4  /*Returns process ID of terminated child, or -1 on error*/
```

  - Waits for one of the children to terminate and returns the termination status in *status* buffer.
  - It blocks until one of the child terminates.
  - It returns the pid of the child process.
  - returns -1 in case there are no more children.

waitpid() System Call

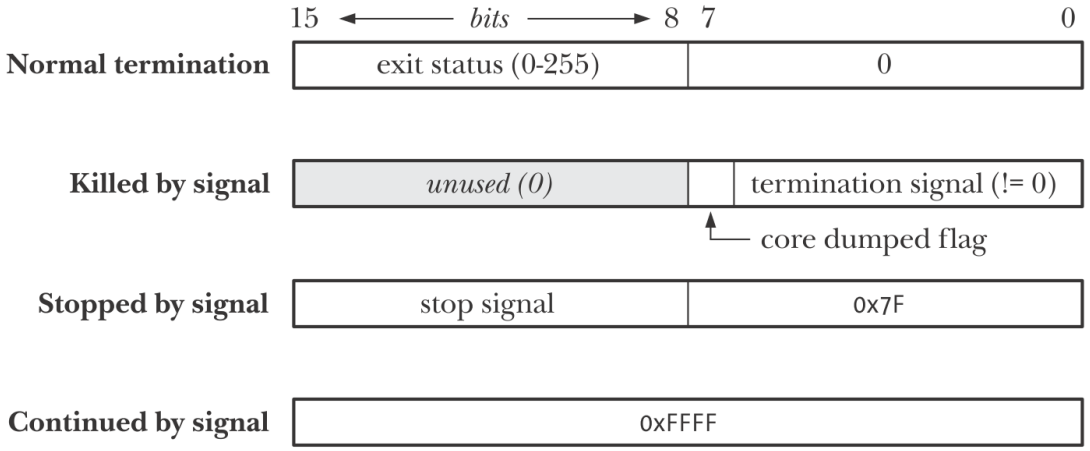
- wait() system call has limitations
  - Can’t wait for a specific child
  - It always blocks until some child terminates.
  - Can’t find out about children which are stopped or continued by a signal.
- waitpid() system call:

```
2   #include <sys/wait.h>
3   pid_t waitpid(pid_t  pid , int * status , int  options );
4   /*Returns process ID of child, 0 (WNOHANG option), or -1 on error*/
```

  - If pid>0, wait for a specific child process. If pid=-1, then same as wait().
  - Options:
    - WUNTRACED: child stopped by a signal SIGSTOP or SIGTTIN
    - WCONTINUED: child continued by SIGCONT signal
    - WNOHANG: do not block if the child is not terminated yet. Return 0.

Wait Status Value

- Although int (4 bytes) but only lower 2 bytes are used.
- There are macros to extract this info from the *status* returned by wait.




```
2   if (WIFEXITED (status))
3       printf ("normal termination, exit status = %d\n", WEXITSTATUS (status));
4   else if (WIFSIGNALED (status))
5     printf ("abnormal termination, signal number = %d \n", WTERMSIG (status));
6   else if (WIFSTOPPED (status))
7       printf ("child stopped, signal number = %d\n", WSTOPSIG (status));
8   else if (WIFCONTINUED (status)) /*available since Linux 2.6.10*/
9       printf ("child continued");
10  }
```

Orphan & Zombie Process

innovate achieve lead

- Orphan process: What happens if parent terminates before child?
  - the init process becomes the parent process of any process whose parent terminates (process has been inherited by init)
  - parent process ID of the surviving process is changed to be 1 (the process ID of init). This way, we're guaranteed that every process has a parent.
- Zombie process: What happens when a child terminates before its parent ?
  - Kernel keeps information (process ID, the termination status of the process, and the amount of CPU time taken by the process) until parent asks for it.
  - a process that has terminated, but whose parent has not yet waited for it, is called a zombie.
  - Zombies can hold up pids and resources in long-live servers.

innovate achieve lead

BITS Pilani

Pilani Campus

Program Execution (R1: Ch27)

Executing a New Program: *execve()*

- Replaces the program running in the process by a new program.
  - Process's text, stack, heap, data are replaced by those of the new program.
  - The process ID does not change across an exec, because a new process is not created;

```
2  #include <unistd.h>
3  int execve(const char * pathname , char *const  argv [], char *const  envp []);
4      /*Never returns on success; returns -1 on error*/
```

  - *pathname: absolute or relative path name of the image*
  - *Argv:* command-line arguments to be passed to the new program
  - *envp:* specifies the environment list for the new program.

Library Functions

innovate achieve lead

```
2   #include <unistd.h>
3   int execle(const char * pathname , const char * arg , ...
4                   /* , (char *) NULL, char *const  envp [] */ );
5   int execlp(const char * filename , const char * arg , ...
6                   /* , (char *) NULL */);
7   int execvp(const char * filename , char *const  argv []);
8   int execv(const char * pathname , char *const  argv []);
9   int execl(const char * pathname , const char * arg , ...
10                  /* , (char *) NULL */);
11  /*None of the above returns on success; all return -1 on error*/
```

- Above functions are library functions. They invoke execve() system call.
  - *l* = list of arguments
  - *v* = vector of arguments
  - *e* = environment specified.
  - *p* = look for file in $PATH environment variable.

Passing Arguments as a List


```c
/*procexec/t_execl.c*/
#include <stdlib.h>
#include "tlpi_hdr.h"
int
main(int argc, char *argv[])
{
    printf("Initial value of USER: %s\n", getenv("USER"));
    if (putenv("USER=britta") != 0)
        errExit("putenv");
    execl("/usr/bin/printenv", "printenv", "USER", "SHELL", (char *) NULL);
    errExit("execl");           /* If we get here, something went wrong */
}
```

- This is useful when we know the number of arguments at the time of writing program.

```
$ echo $USER $SHELL          Display some of the shell's environment variables
blv /bin/bash
$ ./t_execl
Initial value of USER: blv   Copy of environment was inherited from the shell
britta                       These two lines are displayed by execed printenv
/bin/bash
```

Passing Environment to New Program

innovate achieve lead

```
2  /*procexec/t_execle.c*/
3  #include "tlpi_hdr.h"
4  int
5  main(int argc, char *argv[])
6  {
7      char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };
8      char *filename;
9      if (argc != 2 || strcmp(argv[1], "--help") == 0)
10         usageErr("%s pathname\n", argv[0]);
11     filename = strrchr(argv[1], '/');       /* Get basename from argv[1] */
12     if (filename != NULL)
13         filename++;
14     else
15         filename = argv[1];
16     execle(argv[1], filename, "hello world", (char *) NULL, envVec);
17     errExit("execle");           /* If we get here, something went wrong */
18 }
```

- Only those variables mentioned in *envVec* will be passed to new program.

File Descriptors and *exec()*

innovate achieve lead

- By default all open file descriptors remain open across the exec().
  - The fds are available for use in the new program.
  - Shell takes advantage of this feature. e.g. > or |
- close-on-exec flag (FD_CLOEXEC)
  - FD_CLOEXEC is the only bit used in file descriptor's flags.
  - If this flag is set, then that fd will be closed upon successful exec().

```
2   int flags;
3   flags = fcntl(fd, F_GETFD);
4   if (flags == -1)
5       errExit("fcntl");
6   flags |= FD_CLOEXEC;
7   if (fcntl(fd, F_SETFD, flags) == -1)
8       errExit("fcntl");
```

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You