luồng điều khiển

Control Flow

Khái niệm

Concept of Control Flow

Program Counter là 1 thanh ghi dẫn dắt máy tính thực thi các chương trình. Theo thời gian nó sẽ thay đổi giá trị nằm trong nó

- The program counter (PC) assumes a sequence of values: $a_1, a_2, a_3, \ldots, a_{n-1}$

Bản chất thanh ghi là chứa địa chỉ của thanh ghi nó sẽ thực hiện

ak là địa chỉ của nơi chứa mã chỉ thị Ik

- a_k is the address of some corresponding instruction I_k.

mã chỉ thị

Bước chỉ thị

Each transition from a_k to a_{k+1} is called a control transfer.

Một tuần tự các Bước chỉ thị (Control Transfer) thì được gọi là một luồng điêuù khiển(Control Flow)

- A sequence of such control transfers is called the flow of control, or control flow of the processor.

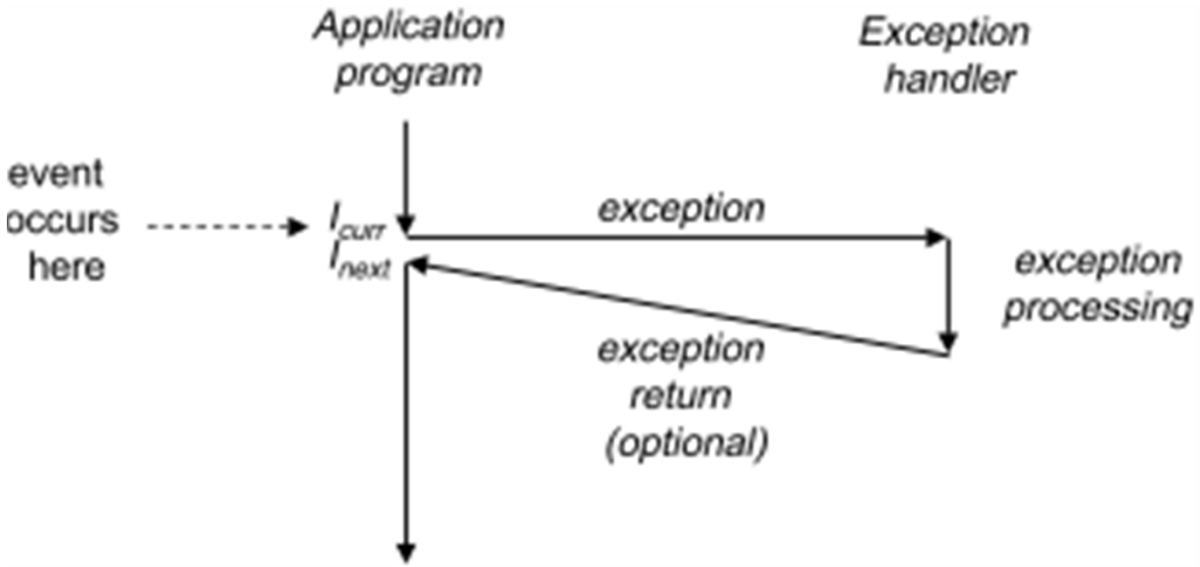
Concept of Exceptional Control Flow

- The simplest kind of control flow is a smooth (tuần tự) sequence where each I_k and I_{k+1} are adjacent (kề nhau) in memory. => thanh ghi PC hoạt động tự nhiên
- I_{k+1} is not adjacent to I_k, such as jumps, calls, and return=> they are necessary mechanisms (cơ chế) that allow programs to react (tác động) to changes in internal program state (trạng thái nội tại của chương trình) represented by program variables.
- Other hand, changes in system state that are not captured by internal program variables, such as timer, I/O… Modern systems react to these changes by making abrupt changes in the control flow. It is refered as exceptional control flow.
- It exists all levels (hardware, operating system, application)

trong TH đặc biệt, sự cần thiết chuyển địa chỉ thực thi ngay lập tức(được tác động bởi các tác động bên ngoài(!= jump, call, return)) làm thay đổi luồng điều khiển => gọi nó là Luồng ĐK Ngoại Lệ(Exceptional CF)

Exceptions Ngoại lệ

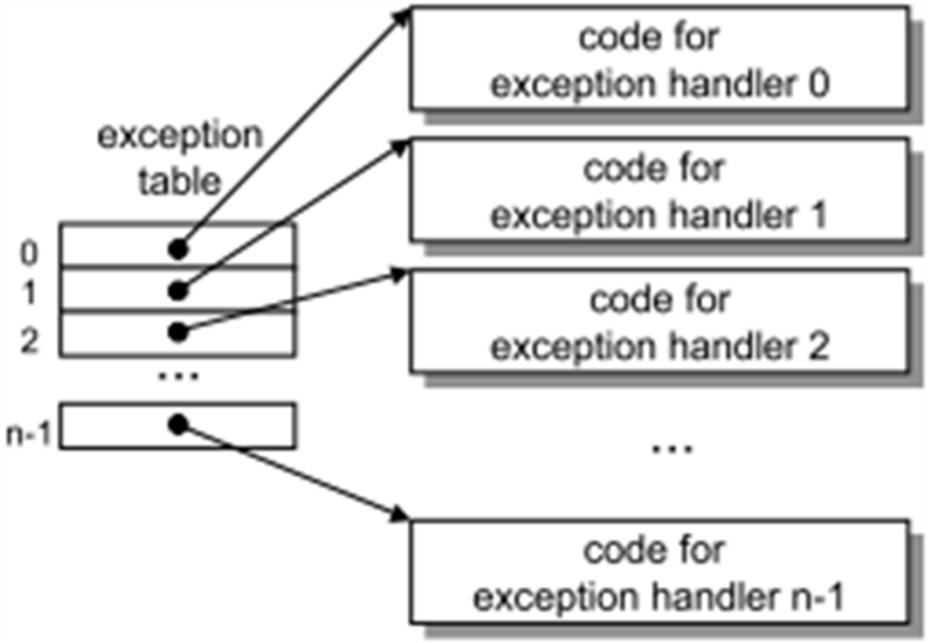
- A form of exceptional control flow that are implemented partly by the hardware and partly by the operating system.
- An exception is an abrupt thay đổi đột ngột change in the control flow in response đáp ứng to some change in the processor's state

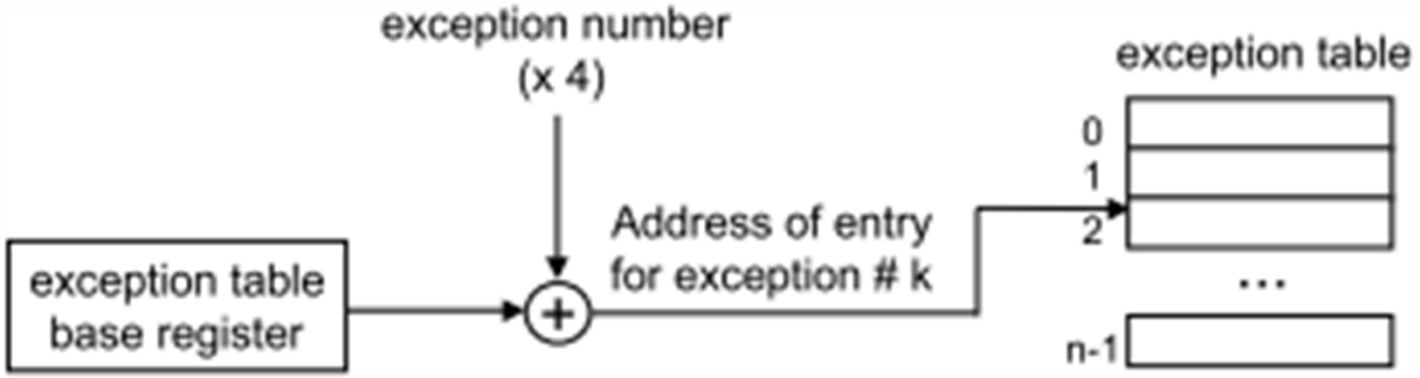


được viết và nạp sẵn ở trong máy tính. Khi có ECF thì nó sẽ được đưa và thanh ghi PC thực hiện ngay tức thì.

Exception Handling

- Each type of possible exception in a system is assigned a unique non-negative integer exception number by the designers of the processor and designers of the operating system kernel
- At system boot time the operating system allocates and initializes a jump table called an exception table, so that entry k contains the address of the handler for exception k.


exception
table
0
1
2
...
n-1
code for
exception handler 0
code for
exception handler 1
code for
exception handler 2
...
code for
exception handler n-1


exception number
(x 4)
exception table
base register
Address of entry
for exception # k
0
1
2
...
n-1

Phân Loại

Classes of Exceptions

- Four classes: Interrupts, Traps, Faults, and Aborts

nguồn phát sinh

bất đồng bộ/ đồng bộ

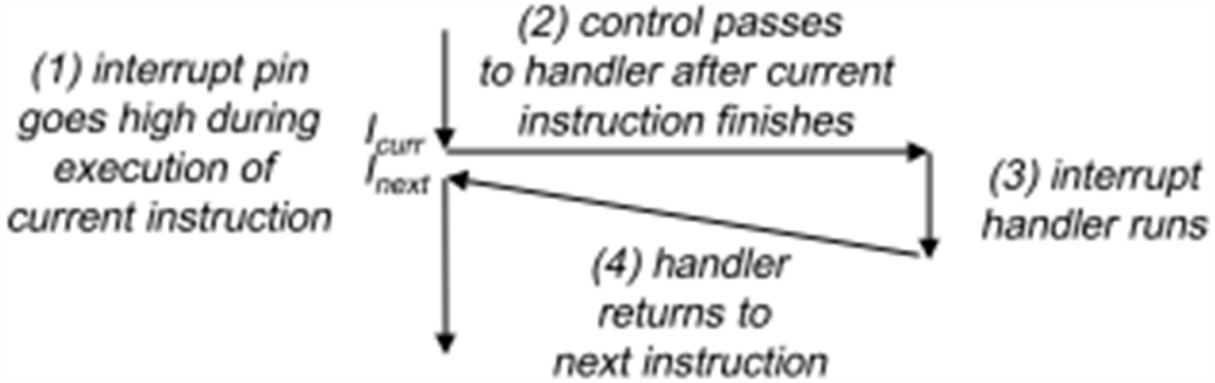
Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

lỗi có thể khắc phục

lỗi không thể khắc phục

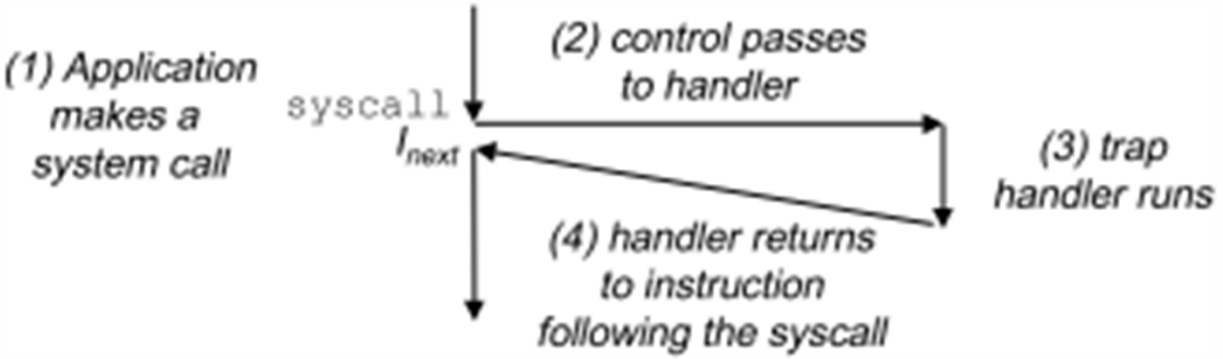
Interrupts

- Interrupts occur asynchronously as a result of signals from I/O devices



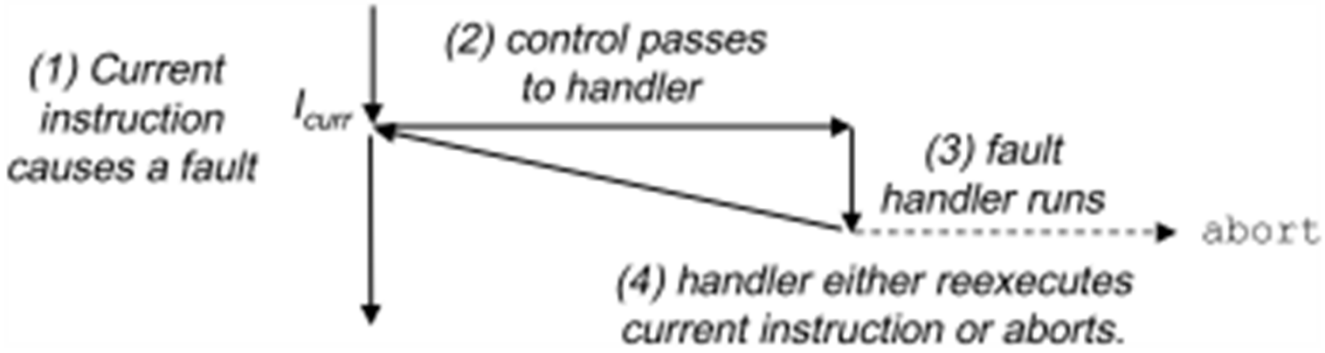
Traps

- Traps are intentional exceptions that occur as a result of executing an instruction
- The most important use of traps is to provide a procedure like interface between user programs and the kernel known as a **system call**
- Executing the syscall instruction causes a trap to an exception handler that decodes the argument and calls the appropriate kernel routine



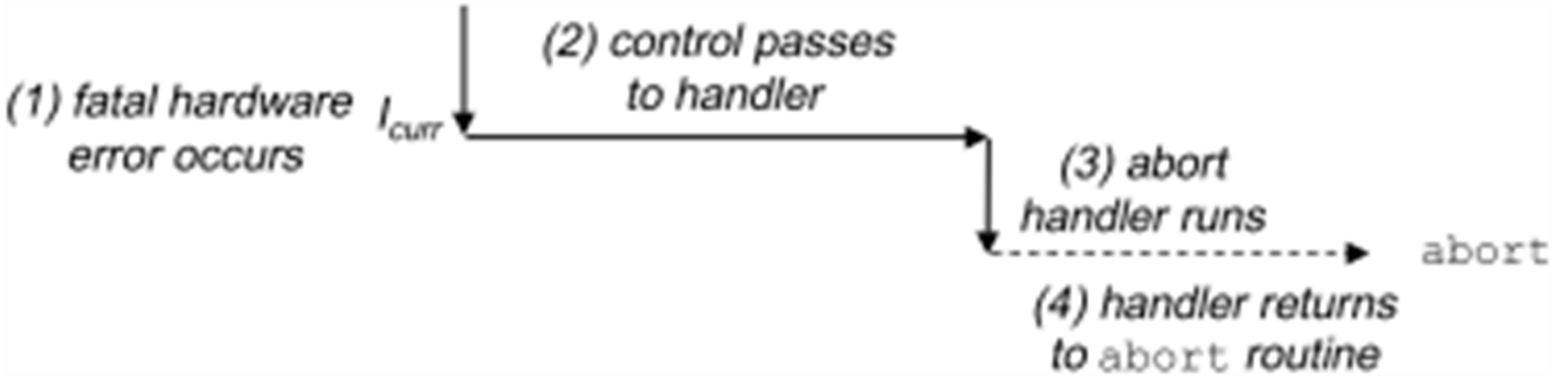
Faults

- When a fault occurs, the processor transfers control to the fault handler.
- If the handler is able to correct the error condition, it returns control to the faulting instruction, thereby reexecuting it.
- Otherwise, the handler returns to an abort routine in the kernel that terminates the application program that caused the fault.



Aborts

- Result from unrecoverable fatal errors
- Abort handlers never return control to the application program.The handler returns control to an abort routine that terminates the application program.



Exceptions in Intel Processors

- Up to 256 different exception types
- Numbers in the range 0 to 31 correspond to exceptions that are defined by the Pentium architecture.
- Numbers in the range 32 to 255 correspond to interrupts and traps that are defined by the operating system.
- System calls are provided on IA32 systems via a trapping instruction called INT n, where n can be the index of any of the 256 entries in the exception table. Historically, system calls are provided through exception 128 (0x80)

Exception Number	Description	Exception Class
0	divide error	fault
13	general protection fault	fault
14	page fault	fault
18	machine check	abort
32–127	OS-defined exceptions	interrupt or trap
128 (0x80)	system call	trap
129–255	OS-defined exceptions	interrupt or trap

Processes

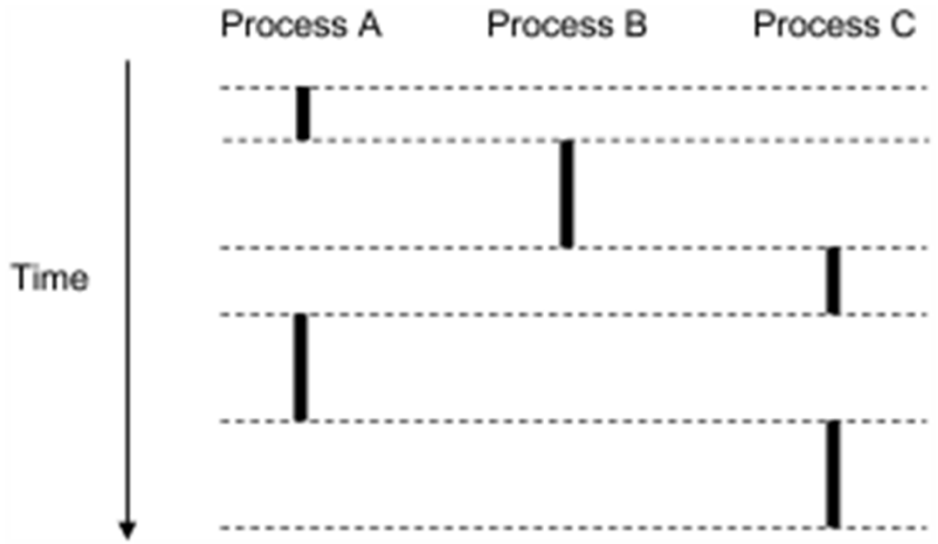
- An instance of a program in execution.
- State includes the program's code and data stored in memory, its stack, the contents of its general-purpose registers, its program counter, environment variables, and the set of open file descriptors
- Each program in the system runs in the context of some process. The context consists of the state that the program needs to run correctly

Key abstractions that a process provides to the application

- An independent logical control flow that provides the illusion that our program has exclusive use of the processor
- A private address space that provides the illusion that our program has exclusive use of the memory system.

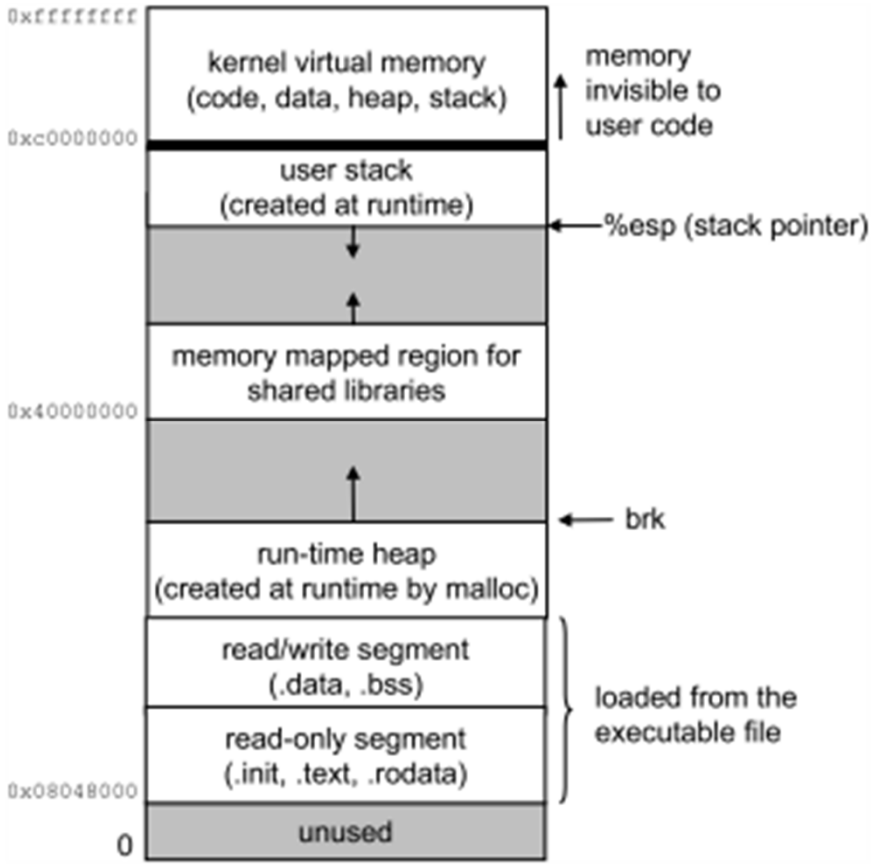
Logical Control Flow

- A process provides each program with the illusion that it has exclusive use (độc quyền sử dụng) of the processor
- Sequence of PC values is known as a logical control flow (PC: Program Counter)
- Concurrent process ; multitasking; time slice



Private Address Space

- A process also provides each program with the illusion that it has exclusive use of the system's address space



User and Kernel Modes

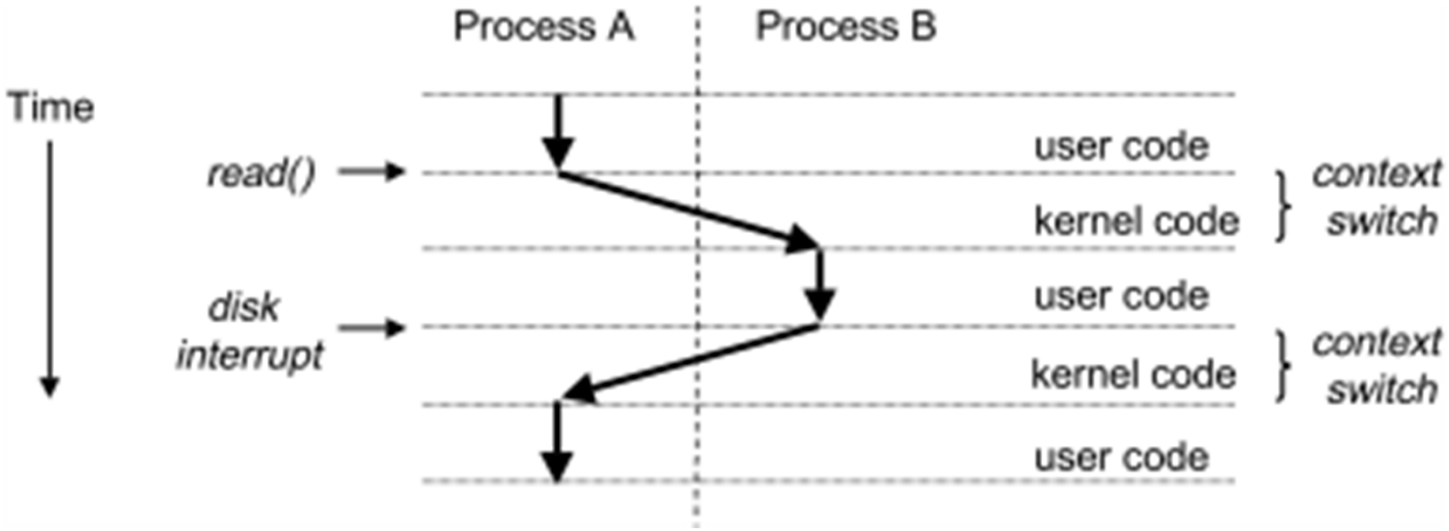
- In order for the kernel to provide an airtight process abstraction
- Use a mechanism that restricts the instructions that an application can execute, as well as the portions of the address space that it can access.
- Controlled by using the mode bit in some control register
- Mode bit is set ➔ process in kernel mode (supersisor mode)
- User mode: process **is not allowed** to execute privileged instructions that do things such as halt the processor, change the mode bit, or initiate an I/O operation, and to directly reference code or data in the kernel area of the address space
- User programs must access kernel code and data indirectly via the system call interface
- The only way for the process to change from user mode to kernel mode is via an exception such as an Interrupt, a fault, or a trapping system call

Context Switches (1/2)

- Context consists of the values of objects such as the general-purpose registers, the floating-point registers, the program counter, user's stack, status registers, kernel's stack, and various kernel data structures such as a page table that characterizes the address space, a process table that contains information about the current process, and a file table that contains information about the files that the process has opened.
- The kernel maintains a context for each process. The context is the state that the kernel needs to restart a preempted process
- The kernel has scheduled a new process to run, it preempts the current process and transfers control to the new process using a mechanism called a **context switch**
- Context switch mechanism is built on top of the lower-level exception mechanism

Context Switches (2/2)

- A context switch can occur while the kernel is executing a system call on behalf of the user.
- A context switch can also occur as a result of an interrupt



Process Control: Obtaining Process ID

- The getpid function returns the PID of the calling process.
- The getppid function returns the PID of its parent (i.e., the process that created the calling process)

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```

returns: PID of either the caller or the parent

Process Control: Creating Processes (Linux)

- From a programmer's perspective, we can think of a process as being in one of three states:
  - Running: process is either executing on the CPU, or is waiting to be executed and will eventually be scheduled
  - Stopped: A process stops as a result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal, and it remains stopped until it receives a SIGCONT signal, at which point is becomes running again
  - Terminated

Process Control: Creating Processes (Linux)

- A parent process creates a new running child process by calling the fork() function
- The newly created child process is almost, but not quite, identical to the parent
- The most significant difference between the parent and the newly created child is that they have different PIDs

Example

```
 1 #include "csapp.h"
 2
 3 int main()
 4 {
 5     pid_t pid;
 6     int x = 1;
 7
 8     pid = Fork();
 9     if (pid == 0) {  /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

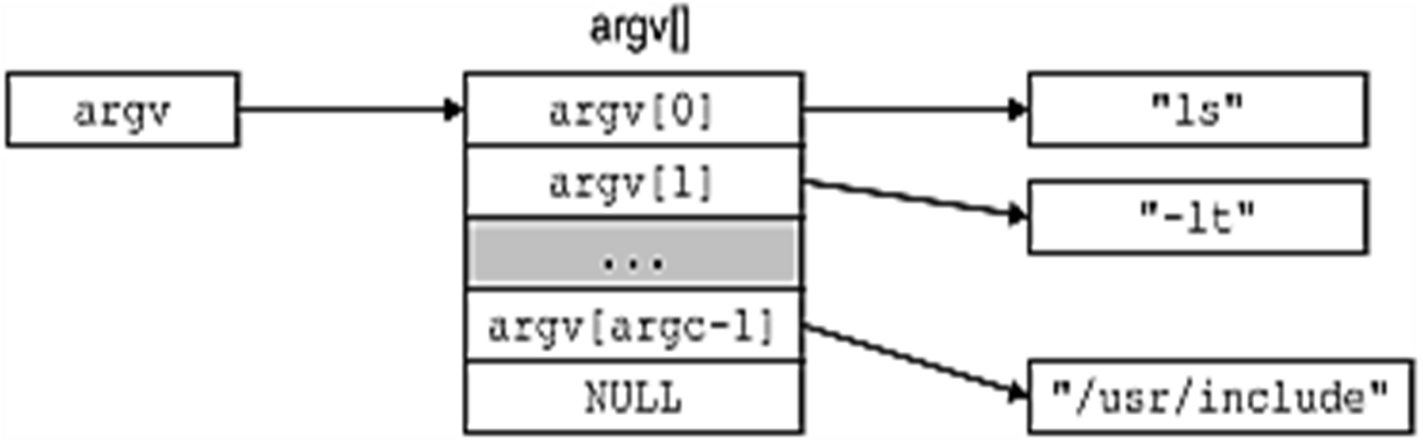
Loading and Running Programs

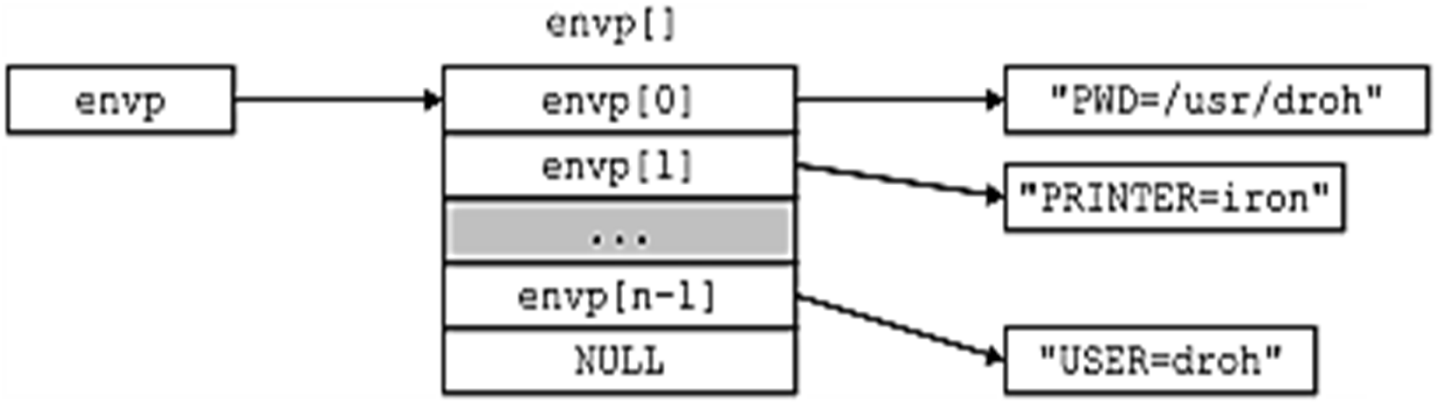
- The execve function loads and runs a new program in the context of the current process

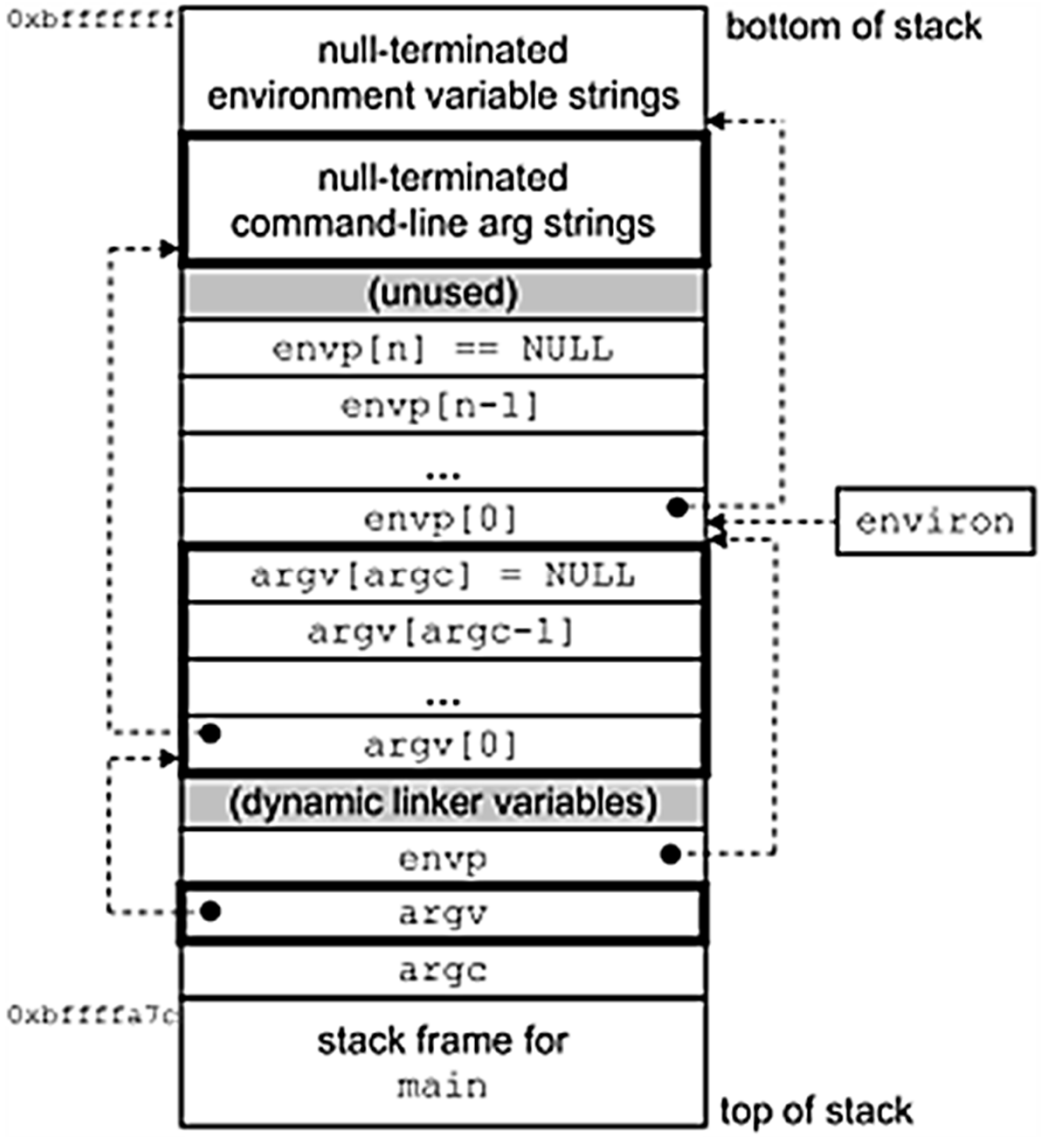
```
# include <unistd.h>

int execve(char *filename, char *argv[], char *envp);
```

(*does not return if OK, returns -1 on error*)

- The execve function loads and runs the executable object file filename with the argument list argv and the environment variable list envp.


argv[]
argv
argv[0]
argv[1]
...
argv[argc-1]
NULL
"ls"
"-lt"
"/usr/include"


envp[]
envp
envp[0]
envp[1]
...
envp[n-1]
NULL
"PWD=/usr/droh"
"PRINTER=iron"
"USER=droh"


0xbfffffff
null-terminated
environment variable strings
bottom of stack
null-terminated
command-line arg strings
(unused)
envp[n] == NULL
envp[n-1]
...
envp[0]
environ
argv[argc] = NULL
argv[argc-1]
...
argv[0]
(dynamic linker variables)
envp
argv
argc
0xbffffa7c
stack frame for
main
top of stack

Using fork() and execve() to Run Programs

- A shell is an interactive application-level program that runs other programs on behalf of the user.
- The original shell was the sh program, which was followed by variants such as csh, tcsh, ksh, and bash.
- A shell performs a sequence of read/evaluate steps, and then terminates.
- The read step reads a command line from the user. The evaluate step parses the command line and runs programs on behalf of the user

Using fork() and execve() to Run Programs

The main routine for a simple shell program.

```
#include "csapp.h"
#define MAXARGS   128
/* function prototypes */
void eval(char *cmdline);
int parseline(const char *cmdline, char **argv);
int builtin_command(char **argv);
 int main()  {
   char cmdline[MAXLINE]; /* command line */
     while (1) {
   /* read */
                printf("> ");
                Fgets(cmdline, MAXLINE, stdin);
                if (feof(stdin))
                exit(0);
  /* evaluate */
                 eval(cmdline);
    } }
```

```
1  /* eval - evaluate a command line */
2  void eval(char *cmdline)
3  {
4      char *argv[MAXARGS]; /* argv for execve() */
5      int bg;              /* should the job run in bg or fg? */
6      pid_t pid;           /* process id */
7
8      bg = parseline(cmdline, argv);
9      if (argv[0] == NULL)
10         return;   /* ignore empty lines */
11
12     if (!builtin_command(argv)) {
13         if ((pid = Fork()) == 0) {   /* child runs user job */
14             if (execve(argv[0], argv, environ) < 0) {
15                 printf("%s: Command not found.\n", argv[0]);
16                 exit(0);
17             }
18         }
19
20         /* parent waits for foreground job to terminate */
21         if (!bg) {
22             int status;
23             if (waitpid(pid, &status, 0) < 0)
24                 unix_error("waitfg: waitpid error");
25         }
26         else
27             printf("%d %s", pid, cmdline);
28     }
29     return;
30 }
31
32 /* if first arg is a builtin command, run it and return true */
33 int builtin_command(char **argv)
34 {
35     if (!strcmp(argv[0], "quit")) /* quit command */
36         exit(0);
37     if (!strcmp(argv[0], "&"))    /* ignore singleton & */
38         return 1;
39     return 0;                     /* not a builtin command */
40 }
```

```
1  /* parseline - parse the command line and build the argv array */
2  int parseline(const char *cmdline, char **argv)
3  {
4      char array[MAXLINE]; /* holds local copy of command line */
5      char *buf = array;   /* ptr that traverses command line */
6      char *delim;         /* points to first space delimiter */
7      int argc;            /* number of args */
8      int bg;              /* background job? */
9
10     strcpy(buf, cmdline);
11     buf[strlen(buf)-1] = ' ';  /* replace trailing '\n' with space */
12     while (*buf && (*buf == ' ')) /* ignore leading spaces */
13         buf++;
14
15     /* build the argv list */
16     argc = 0;
17     while ((delim = strchr(buf, ' '))) {
18         argv[argc++] = buf;
19         *delim = '\0';
20         buf = delim + 1;
21         while (*buf && (*buf == ' ')) /* ignore spaces */
22                buf++;
23     }
24     argv[argc] = NULL;
25
26     if (argc == 0)  /* ignore blank line */
27         return 1;
28
29     /* should the job run in the background? */
30     if ((bg = (*argv[argc-1] == '&')) != 0)
31         argv[--argc] = NULL;
32
33     return bg;
34 }
```

The End