

Process Management

Slides were adopted from different sources:

- Operating System Concepts by Silberschats, Galvin, and Gagne
- Operating Systems by William Stallings
- OS course in UNC by Kevin Jeffay

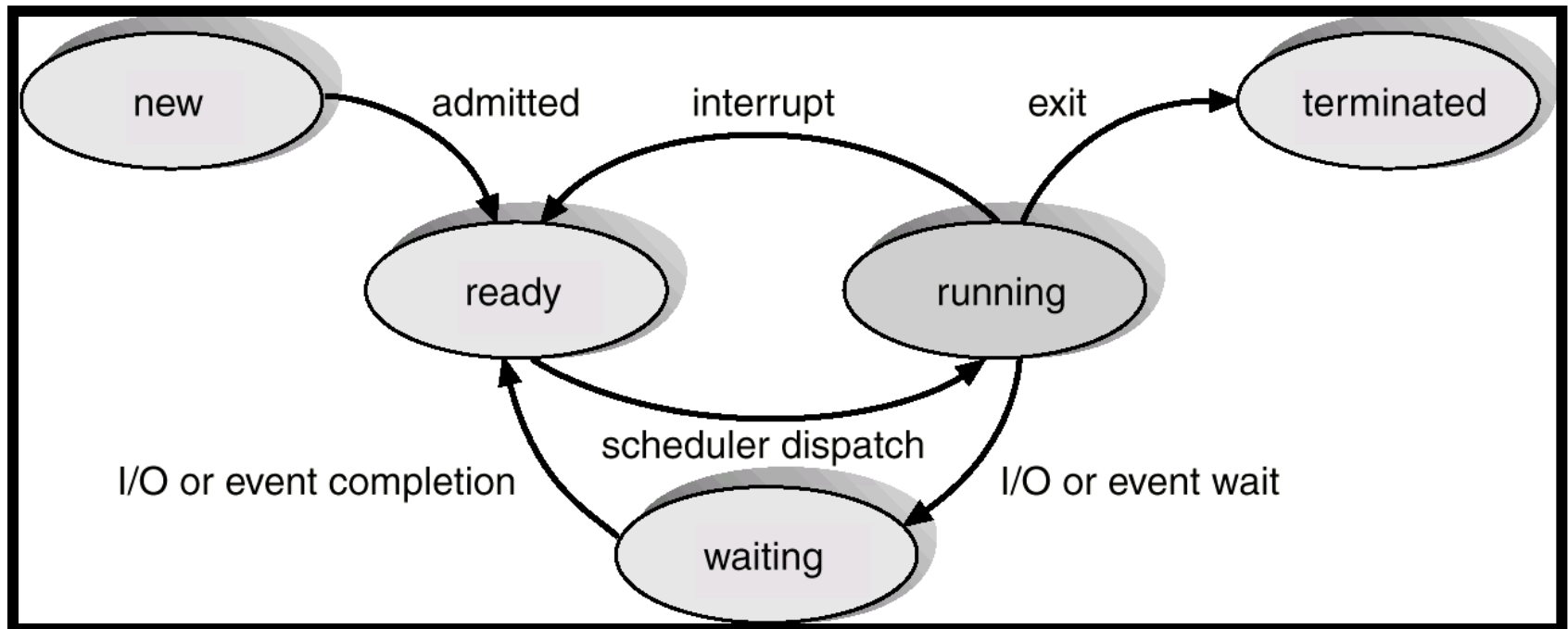
Process Concept

- **Process** – a program in execution
- It is the **basic active entity** in an OS
- An operating system **executes a variety of programs**:
 - Batch system – **jobs**
 - Time-shared systems – **tasks**
- process execution must progress in **sequential fashion**

Process State

- As a process executes, it changes state
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a process
 - **terminated**: The process has finished execution

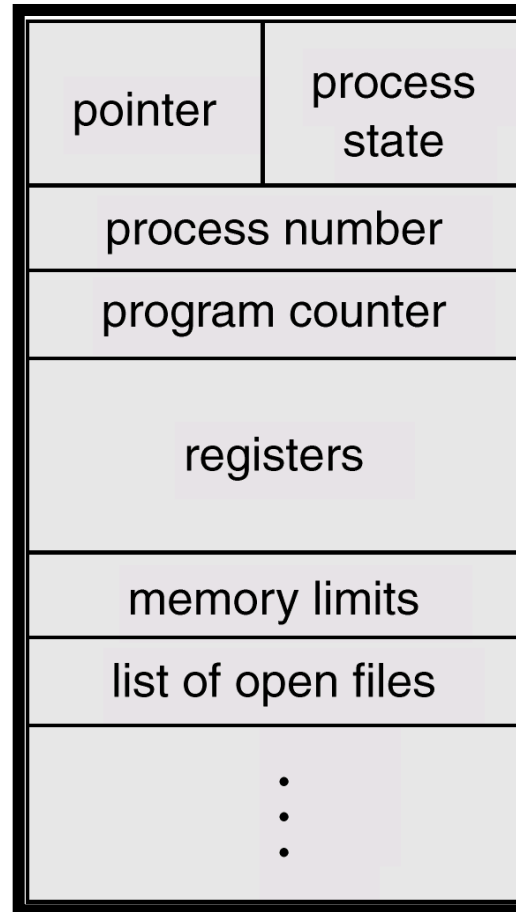
Process State



Process Creation

- OS **allocates** proper resources and **adds** appropriate **info** to the **kernel data structures**
- Information associated with each process:
 - Process state
 - Program counter
 - CPU registers
 - CPU scheduling information
 - Memory-management information
 - Accounting information
 - I/O status information

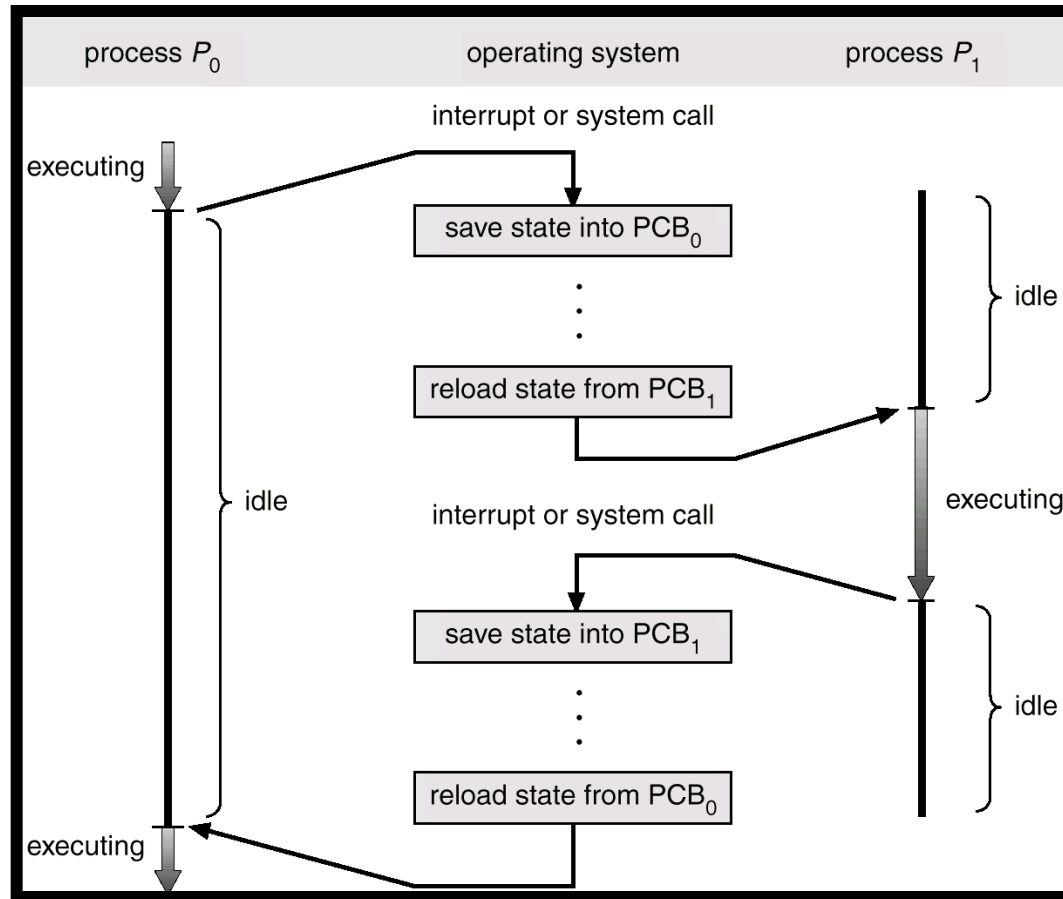
Process Control Block (PCB)



Context Switch

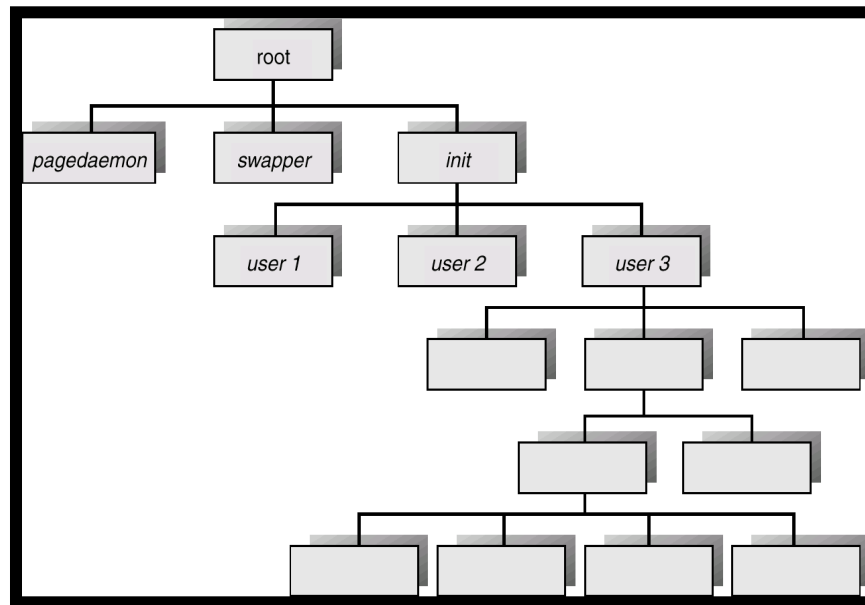
- When CPU switches to another process
 - OS **saves** the state of the old process
 - OS **loads** the saved state for the new process
- Context-switch time is **overhead**;
system does no useful work while switching
- Time dependent on **hardware support**

CPU Switch From Process to Process



Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes



A tree of processes on a typical UNIX system

Process Creation

- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate
- Address space
 - Child duplicate of parent (e.g. UNIX)
 - Child has a program loaded into it (e.g. VMS)

Process Termination

- Process **executes last statement** and asks the operating system to **delete it** (exit)
 - Output data from child to parent (via **wait**: parent collect child's **status and execution statistics**)
 - Process' resources are **deallocated** by operating system
- Parent may **terminate** execution of **children processes** (**abort**)
 - Child has **exceeded** allocated resources
 - **Task** assigned to child is **no longer required**
 - Parent is exiting
 - Operating system does not allow a child process to continue if its parent terminates (**cascading termination**)

Process Creation in Linux/UNIX

- A new process can be created using the system call `fork()`
- `fork()` returns two processes:
 - Parent (original)
 - Child (new)
- `fork()` makes a copy of the parent address space
- Both processes continue execution after `fork()` statement

Process Creation

```
#include <unistd.h>
pid_t fork(void); // pid_t: unsigned int
```

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    int x;
    x = 0;
    fork();
    x++;
    cout << "I am process " << getpid()
          << " and my x is \n" << x;
    return 0;
}
```

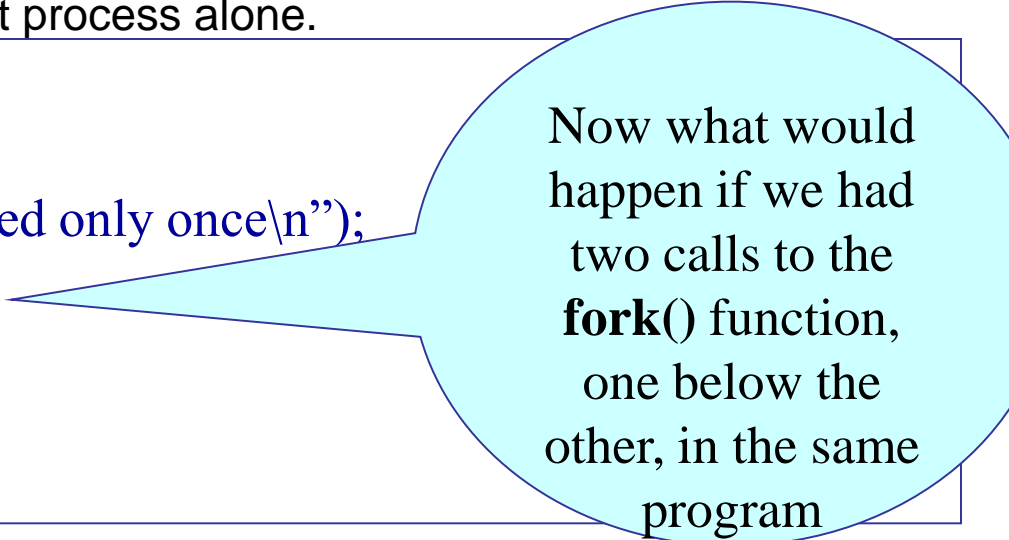
Process Creation !!!

- Because both may perform **different tasks**, we should be able to identify them
- The return value is:
 - **Child's process ID** (in parent process)
 - **Zero** (in the child process)
- Child and parent processes execute **different parts of the same program**

The 'fork()' System Call (continued....)

- The fork() creates a child that is a duplicate of the parent process .
- Since now there are two identical processes in memory, the Hello World is printed twice
- The child process begins from the 'fork()'
- All the statements after the call to 'fork()' will be executed twice. Once by the parent process and once by the child process
- But had there been any statements before the 'fork()' they would have been only executed by the parent process alone.

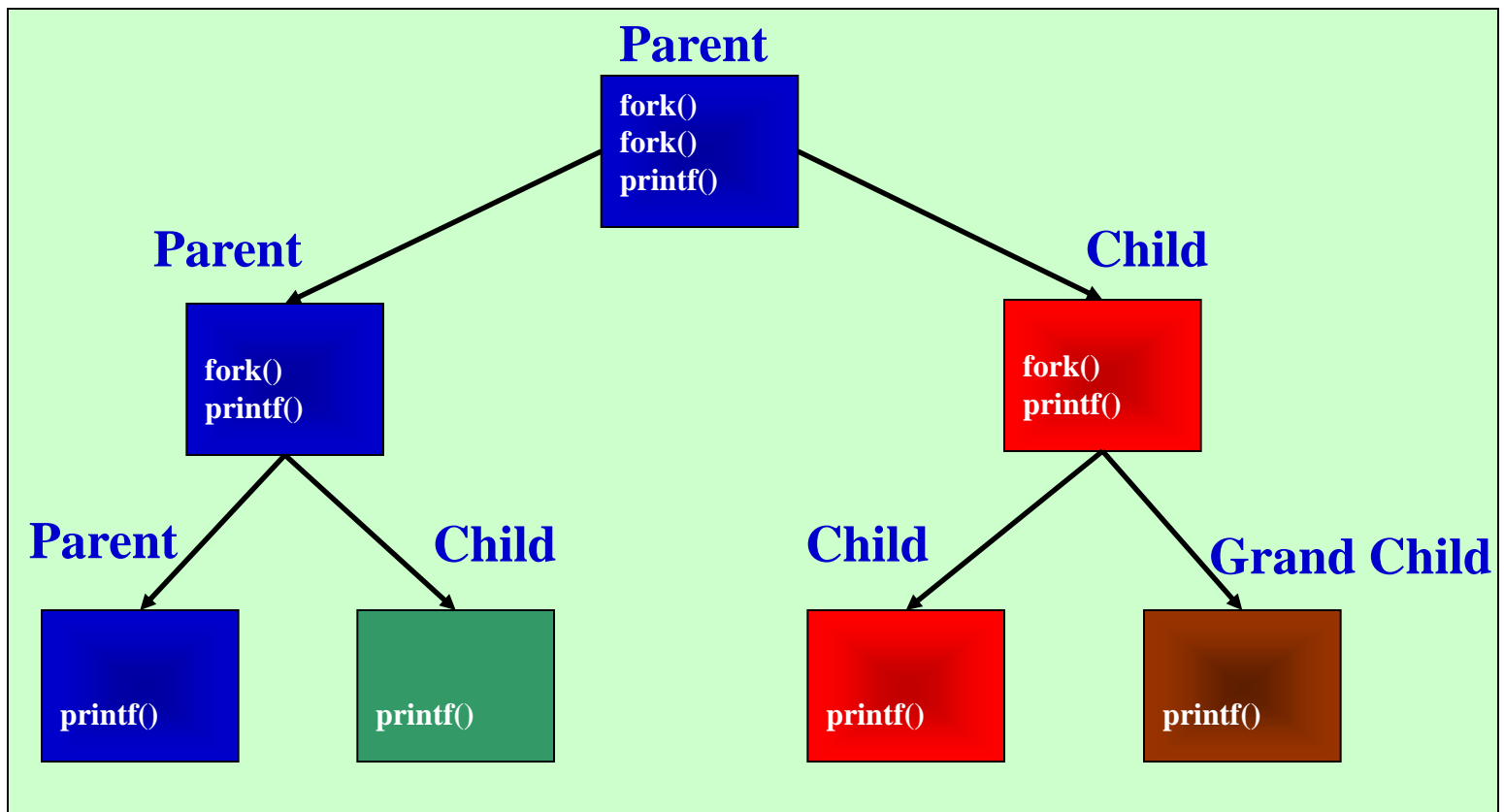
```
main()
{
    printf("This will be printed only once\n");
    fork();
    fork();
    printf("Hello World\n");
}
```



Now what would happen if we had two calls to the **fork()** function, one below the other, in the same program

The 'fork()' System Call (continued.....)

- Here instead of 2 processes, 3 processes will be created . This gives a total of 4 processes in memory: the parent , its 2 children and one grand child



The 'fork()' System Call (continued.....)

- The progression with each 'fork()' that is added is '2 raised to the power of n', where 'n' is the number of calls to fork().
- To the parent process the 'fork()' will return the value of the child's PID. Whereas in the child process the value of the PID will always be 0.

Orphan Process

- An orphan process is one whose parent process terminates before itself i.e. the parent is dead but the child process is still alive.
- Once a child process becomes an orphan the '**process dispatcher**' will immediately adopt it.
 - The parent process id of the child becomes 1
 - The following program will demonstrate an orphan process

```
main( )  
{  
    int pid;  
    pid = fork();  
  
    if (pid == 0)  
    {  
        printf("I am the child,my process id is %d\n",getpid( ));
```

Orphan Process (continued.....)

```
printf("the child's parent process id is %d\n",getpid( ));  
sleep(20);  
printf("I am the child,my process id is %d\n",getpid( ));  
printf("I am the child,parents process id is %d\n",getpid( ));  
}  
else  
{  
    printf("I am the parents,my process id is %d\n",getpid( ));  
    printf("the parents parent process id is %d\n",getpid( ));  
}  
}
```

Zombie process

- A zombie is a child process which has terminated but its parent is not informed about the termination.
- A zombie process is one that has terminated but has not been cleaned up yet i.e. its entry is still retained in the process table
- The following program will demonstrate a zombie process

```
main ( )  
{  
    if (fork ( ) > 0)  
    {  
        printf("parent\n");  
        sleep(50);  
    }  
}
```

- Run the program above as a background process. At the Linux prompt, type 'ps -el'. You will see in the 'process table' an entry with a 'Z' for zombie in the second column.

A "Sleeping Beauty" Process

Let's now see a **sleeping beauty**.

```
main ( )  
{  
    sleep(50);  
    printf("Hello World");  
}
```

- Run this as a background process and its PID will be outputted to the screen. Then do a 'ps-el' to see the 'process table' and go to the entry with the PID just displayed. In the second column of this entry you will see an 'S' indicating that the process is in a 'sleep state'

Changing Process Context

- Use `exec()` system call:
 - Causes the process to **replace its execution image** with that of a new program.
 - **Looses context** of the old program
 - Code space, data space, stack, heap
 - **Retains** process ID, parent, children, and open file descriptors.

exec ()

- Six different flavors, based on:
 - Program path name (relative or absolute)
 - Environment variables (inherited or explicit)
 - Command line arguments (explicit or vector)
- Easiest :
 - `execlp(char *filename, char *arg0, char arg1, , char *argn, (char *) 0);`
 - e.g. `execlp("sort", "sort", "-n", "foo", 0);`

exec () Flavors

Explicit list of arguments terminated by NULL pointer

- `int execl(char *pathname, char *arg0, char arg1,..... , char *argn, (char *) 0);`
Explicit pathname is provided
Environment variables are inherited from calling process
- `int execlp(char *filename, char *arg0, char arg1,..... , char *argn, (char *) 0);`
Filename has a '/' -> execl
Filename does not have a '/' -> PATH is searched
Environment variables are inherited from calling process
- `int execl(char *pathname, char *arg0, char arg1,..... , char *argn, (char *) 0, char *envp[]);`
Explicit pathname is provided
Environment is passed as a parameter

exec () Flavors

Passes the command-line arguments in an argument array

- `int execl(char *pathname, char **argv,);`
Explicit pathname is provided
Environment variables are inherited from calling process
- `int execlp(char *filename, char **argv);`
Filename has a '/' -> execl
Filename does not have a '/' -> PATH is searched
Environment variables are inherited from calling process
- `int execve(char *pathname, char **argv, char **envp);`
Explicit pathname is provided
Environment is passed as a parameter

The execl() System Call

- This function is passed three parameters, but the second parameter can be further subdivided to reflect many parameters (It means that we can pass some values(arguments) to the second program through the call to execl)
 - The path where the file will be found
 - The file name itself
 - A NULL to terminate, since all the parameters are taken as one string.

Consider the two programs below:

```
main()
{
    printf("Before exec my ID is %d \n",getpid());
    printf("My parent process's id is %d\n",getppid());
    printf("exec starts\n");
    execl("/usr/guest/ex2","ex2",(char *)0);
    printf("this will not print\n");
}
```

The execl() System Call (continued.....)

Wait don't run the above program till you have compiled the one below

```
main()
{
printf("After the exec my process id is %d\n",getpid());
printf("My parent process's id is %dn",getppid());
printf("exec ends\n");
}
```

- Now we will run the first program. Its PID will be printed on screen. And a call will be made to the second program, ie. 'ex2.c' through the execl() function.

Basic Control (`wait()`, `exit()`)

- `exit(exit_status)`
 - `exit_status = 0` (success)
 - `exit_status != 0` (failure)
- `wait()`
 - Returns
 - Exit status of the child
 - Process ID of child whose termination caused the `wait()` to wake up.
 - `wait(0)` : waits for the child to die

Synchronization between parent and child

`wait(0)`

- Used by the parent to wait for the child to die

`exit()`

- Used by the child upon completion

Example 2:

```
/* tinymenu.c */
#include ,stdio.h.

main() {
    char *cmd[]={ "who", "ls", "date" };
    int i;
    printf("0=who, 1=ls, 2=date :) ;
    scanf("%d", &i);
    execlp(cmd[i], cmd[i], 0);
    printf("command not found\n"); /*exec failed*/
}
```

- `execlp()` call never returns unless `execlp()` fails.
- If `execlp()` succeeds, control is transferred to the new program – all context within the old program is lost; there is no way back
- This limitation can be solved using the idea of `fork()` with child parent synchronization.

Example 3:

```
main() {
    char *cmd[]={ "who", "ls", "date" };
    int i;
    while(1){
        printf("0=who, 1=ls, 2=date :) ;
        scanf("%d", &i);
        if (fork() == 0) {
            execlp(cmd[i], cmd[i], 0);
            printf("command not found\n"); /*exec failed*/
            exit(1);
        }
        else {
            wait(0);
        }
    }
}
```

Example 3: (cont.)

```
main() {  
    char *cmd[]={ "who", "ls", "date"};  
    int i;  
    while(1){  
        printf("0=who, 1=ls, 2=date :) ;  
        scanf("%d", &i);
```

Parent

```
        else {  
            wait(0);  
        }  
    }  
}
```

Child

```
if (fork() == 0) {  
    execlp(cmd[i], cmd[i], 0);  
    printf("command not  
    found\n"); /*exec  
    failed*/  
    exit(1);  
}
```


Inheritance of Process Attributes

Attribute	Inherited by child?	Retained on exec?
Process ID	No	Yes
Static data	Copied	No
Stack	Copied	No
Heap	Copied	No
Code	Shared	No
Open file descriptors	Copied	Usually
Environment	Yes	Depends of version
Current directory	Yes	Yes