Operating Systems

Acknowledgement

Some slides and pictures are adapted from Lecture slides / Books of

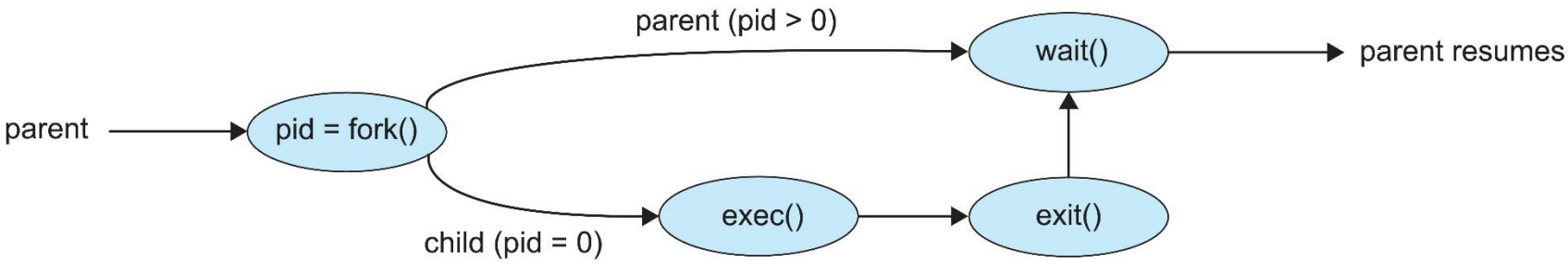
- Dr. Syed Mansoor Sarwar.
- Muhammad Adeel Nisar
- Text Book - OS Concepts by Silberschatz, Galvin, and Gagne.

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

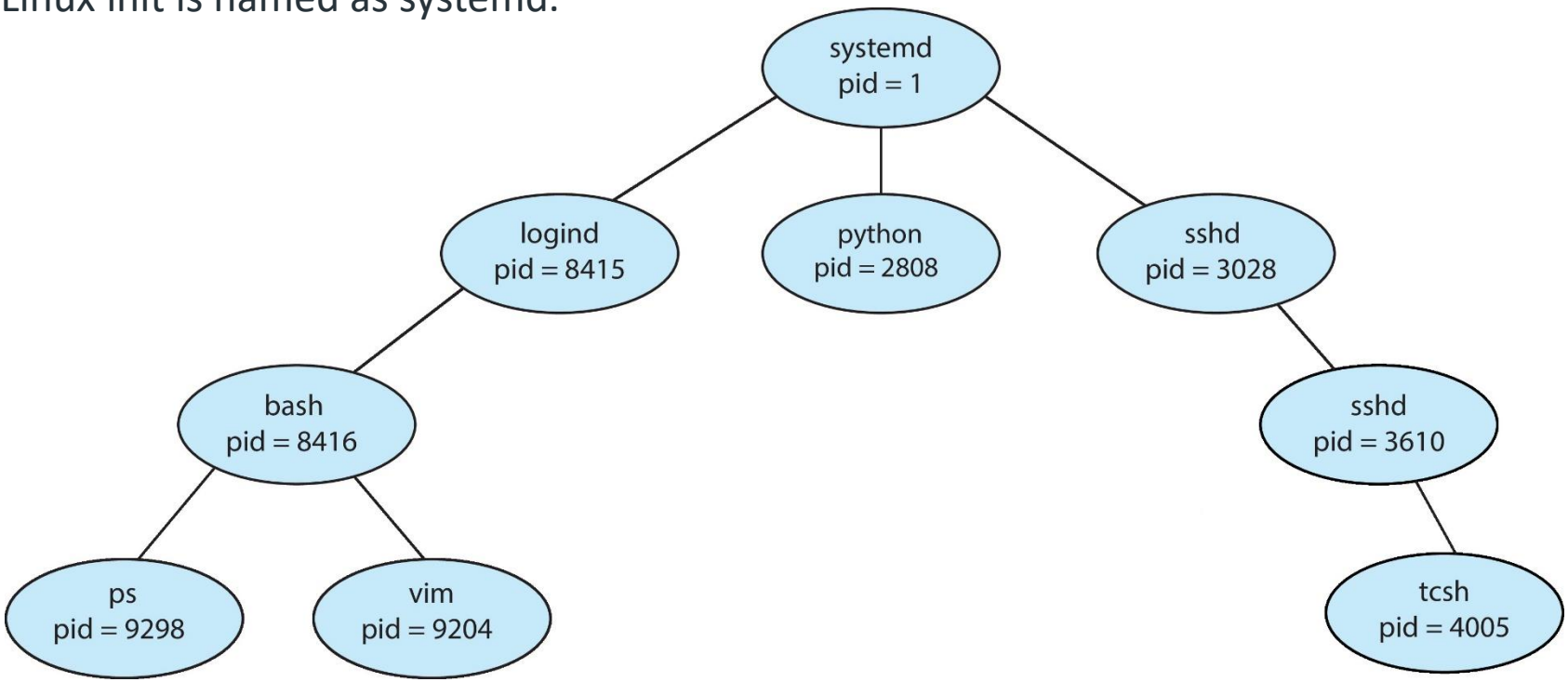
Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate



A Tree of Processes in Linux

- **init** is parent of all Linux processes with PID or process ID of 1.
- It is the first process to start when a computer boots up and runs until the system shuts down.
- init stands for initialization.
- In simple words the role of init is to create processes from script stored in the file /etc/inittab which is a configuration file which is to be used by initialization system. It is the last step of the kernel boot sequence.
- In Linux init is named as systemd.



Process Creation

- **Reasons for Process Creation**
  - In batch environment a process is created in response to the submission of a job
  - In interactive environment a process is created when a new user attempts to log on
  - OS can create a process to perform a function on behalf of a user program. (e.g. printing)
  - Spawning: When a process is created by OS at the explicit request of another process.

Process Termination

- Process executes the last statement and requests the operating system to terminate it (`exit`).
  - Output data from child to parent (via `wait`).
  - Process resources are deallocated by the operating system, to be recycled later.
  - A process may terminate due to Normal completion, Memory unavailable, Protection error, Mathematical error, I/O failure, Cascading termination (by OS), Operator intervention.

Process Termination (cont…)

- A **parent** may terminate execution of one of its children for a variety of reasons such as:
  - Child has exceeded allocated resources (main memory, execution time, etc.)
  - Parent needs to create another child but has reached its maximum children limit
  - Task performed by the child is no longer required
  - Parent exits
    - OS does not allow child to continue if its parent terminates
    - Cascaded termination

Process Management in UNIX/Linux

- Important process-related UNIX/Linux system calls
  - fork()
  - exit()
  - wait()
  - exec()

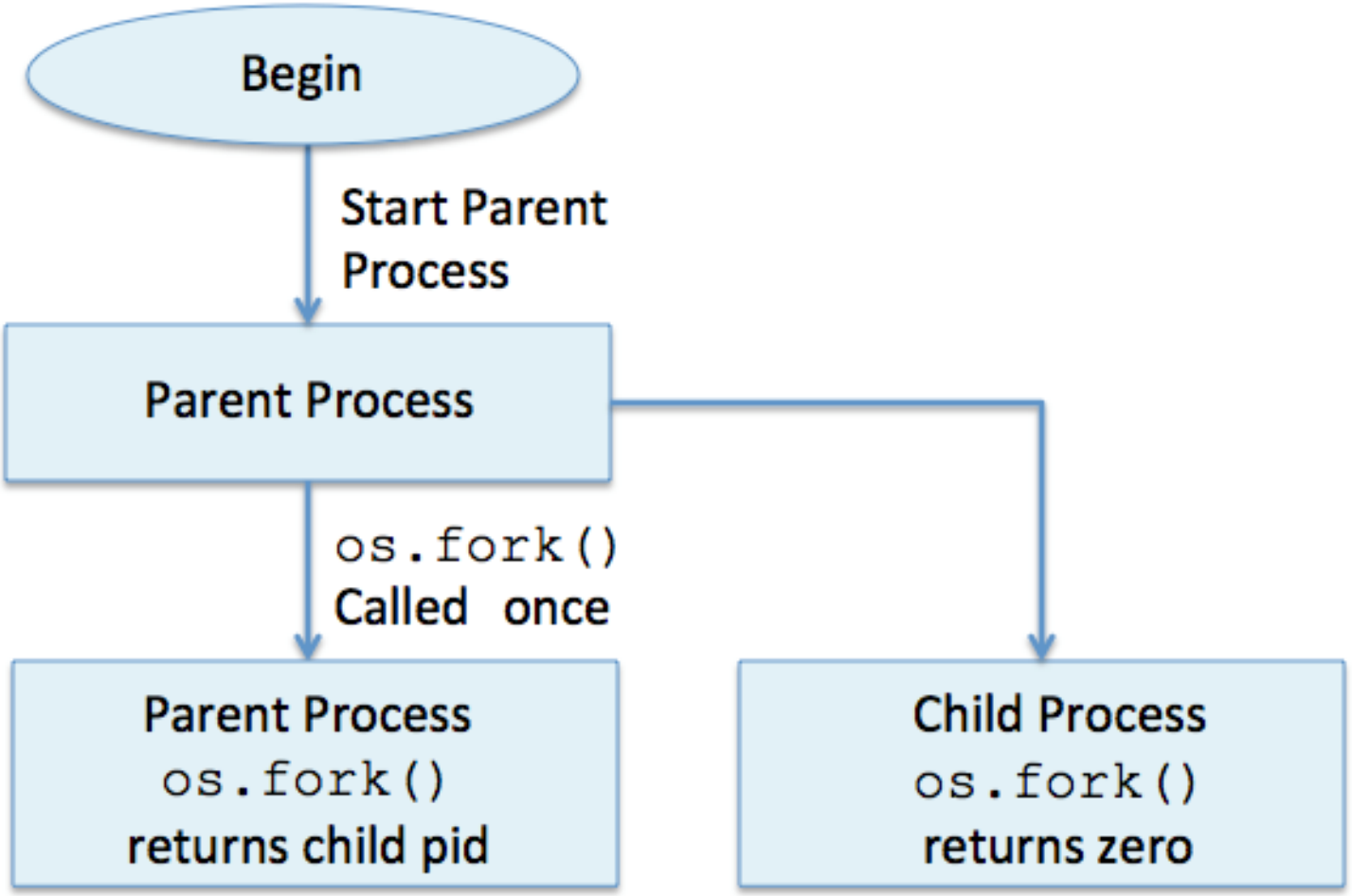
System Call - fork()

- When the fork system call is executed, a new process is created which consists of a copy of the address space of the parent
- An exact copy of the parent program is created
- This mechanism allows the parent process to communicate easily with the child process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

System Call - fork()



System Call - fork() ...

- **On success:** (Child process is created)
  - The return code for fork is zero for the child process and the child process ID is returned to the parent process
  - Both processes continue execution at the instruction after the fork call
- **On failure:** (No child process is created)
  - A -1 is returned to the parent process
  - Variable **errno** is set appropriately to indicate the reason of failure

Using fork() & exit() system call

- **Parent forks**

PID: 597 **Parent**

```
1. //fork1.cpp
2. int main()
3. {
4.       int i = 54, cpid = -1;
5.       cpid = fork();
6.       if (cpid == -1)
7.       {
8.                cout <<"\nFork failed\n";
9.                exit (1);
10. }
11. if (cpid == 0)    //child code
12.    cout <<"\n Hello I am child \n";
13. else                    //parent code
14.    cout <<("\n Hello I am parent \n";
15. }
```

DATA

i = 54

cpid = -1

Using fork() & exit() system call

PID: 597 **Parent**

```
1. //fork1.c
2.int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6. if (cpid == -1)
7.  {
8.        cout <<"\nFork failed\n";
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       cout <<"\n Hello I am child \n";
13. else              //parent code
14.       cout <<"\n Hello I am parent \n";
15. }
```

DATA

i = 54
cpid = 598

PID: 598 **Child**

```
1. //fork1.c
2.int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6. if (cpid == -1)
7.  {
8.        cout <<"\nFork failed\n";
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       cout <<"\n Hello I am child \n";
13. else              //parent code
14.       cout <<"\n Hello I am parent \n";
15. }
```

DATA

i = 54
cpid = 0

- After fork parent and child are identical except for the return value of fork (and of course the PIDs).
- Because data are different therefore program execution differs.
- They are free to execute on their own from now onwards, i.e. after a successful or unsuccessful fork() system call both will start their execution from line#6.

Using fork() & exit() system call

PID: 597 **Parent**

```
1. //fork1.c
2.int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6. if (cpid == -1)
7.  {
8.        cout <<"\nFork failed\n";
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       cout <<"\n Hello I am child \n";
13. else              //parent code
14.       cout <<"\n Hello I am parent \n";
15. }
```

DATA

i = 54
cpid = 598

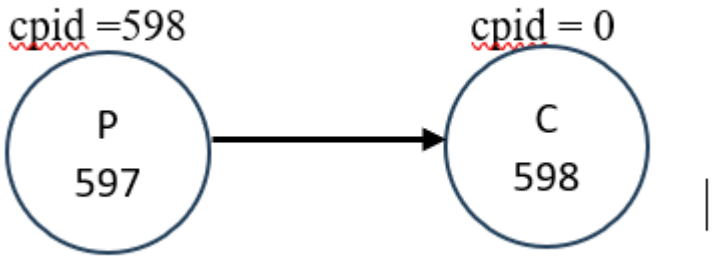
PID: 598 **Child**

```
1. //fork1.c
2.int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6. if (cpid == -1)
7.  {
8.        cout <<"\nFork failed\n";
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       cout <<"\n Hello I am child \n";
13. else              //parent code
14.       cout <<"\n Hello I am parent \n";
15. }
```

DATA

i = 54
cpid = 0

- Process Tree



Using fork() & exit() system call

PID: 597 **Parent**

```
1. //fork1.c
2.int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6.if (cpid == -1)
7.  {
8.        cout <<"\nFork failed\n";
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       cout <<"\n Hello I am child \n";
13. else              //parent code
14.       cout << "\n Hello I am parent \n";
15. }
```

DATA

i – 54
cpid = 598

PID: 598 **Child**

```
1. //fork1.c
2.int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6.if (cpid == -1)
7.  {
8.        cout <<"\nFork failed\n";
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       cout <<"\n Hello I am child \n";
13. else              //parent code
14.       cout << "\n Hello I am parent \n";
15. }
```

DATA

i – 54
cpid = 0

- When both will execute line 11, parent will now execute line 14 while child will execute line 12.

Example 1 -fork() & exit()

```
//fork1.c
int main()
{
  int  cpid;
  cpid = fork();
  if (cpid == -1)
  {
     cout <<"\nFork failed\n";
     exit (1);
  }
  if (cpid == 0)   //child code
     cout <<"\n Hello I am child \n";
  else                //parent code
     cout <<"\n Hello I am parent \n";
}
```

What will be the output of the code?

Output- 1

Hello I am child

Hello I am parent

OR

Output- 2

Hello I am parent

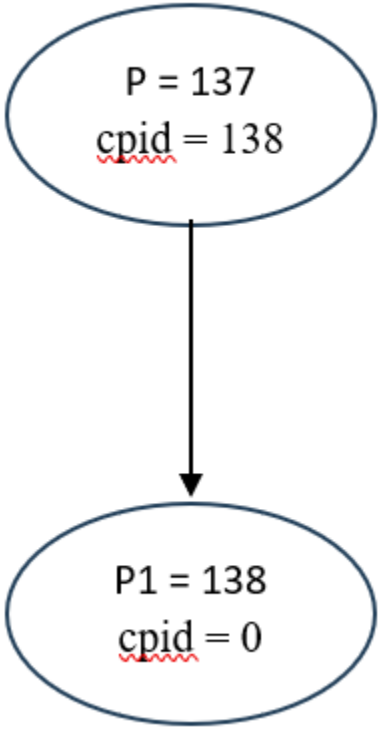
Hello I am child

- If child process executes first and then CPU executes the parent; we will get **Output-1**

•If parent process executes first, it terminates and the child process become Zombie, process **init** takes over control and execute the child process as its own child and we get output similar to **Output-2**

Example 2

```cpp
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

using namespace std;

int main() {
    pid_t opid, pid, mypid, myppid;
    opid = getpid();
    cout << "Before fork: Origional Process id is " << opid << endl;
    pid = fork();

    if (pid < 0) {
        perror("fork() failure\n");
        return 1;
    }

    // Child process
    if (pid == 0) {
        cout << "This is child process" << endl;
        mypid = getpid();
        myppid = getppid();
        cout << "Process id is " << mypid << " and PPID is " << myppid << endl;
    } else { // Parent process
        sleep(2);
        cout << "This is parent process" << endl;
        mypid = getpid();
        myppid = getppid();
        cout << "Process id is " << mypid << " and PPID is " << myppid << endl;
        cout << "Newly created process id or child pid is " << pid << endl;
    }
    return 0;
}
```



Output:

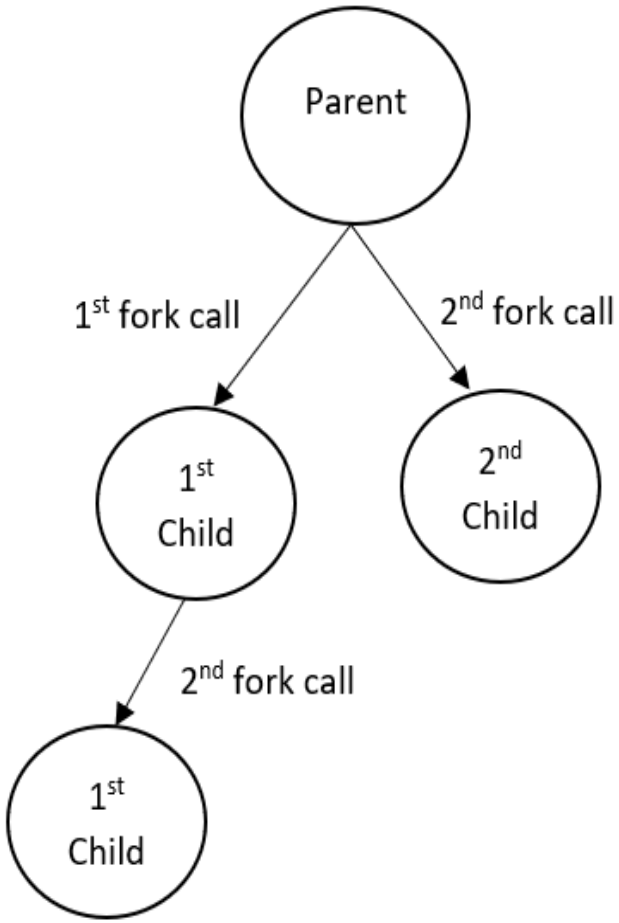
Before fork: Process id is 137
This is child process
Process id is 138 and PPID is 137
This is parent process
Process id is 137 and PPID is 116
Newly created process id or child pid is 138

Output:

Before fork: Process id is 199
This is parent process
Process id is 199 and PPID is 116
Newly created process id or child pid is 200
This is child process
Process id is 200 and PPID is 199

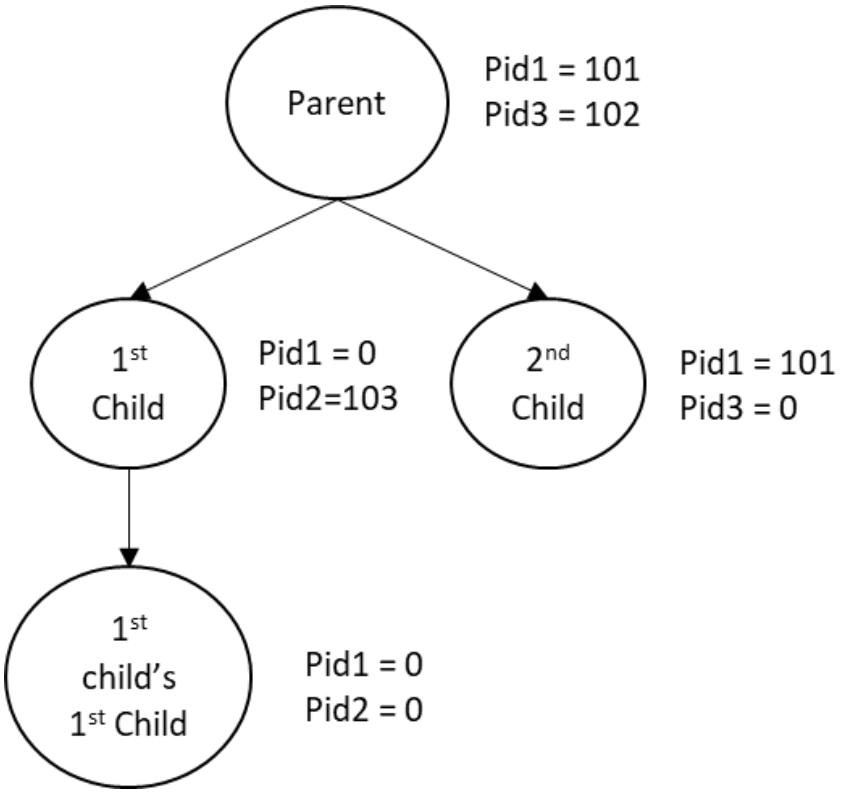
Example 3

```
int main () {

    int pid1 = fork ();
    int pid2 = fork ();
    if (pid2 == 0)
    {
        Block A
    }
    if (pid2 > 0)
    {
        Block B
     }
     if (pid1 > 0)
     {
        Block C
      }
      if (pid1 == 0)
      {
         Block D
      }
    }
  return 0;
```



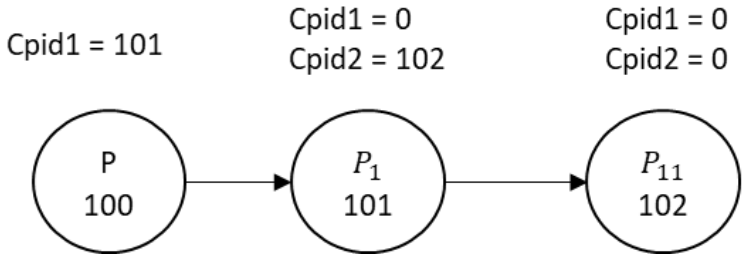
Example 4

```
int main ()
{
   int pid1 = fork ();
   if (pid1 == 0)
   {
      int pid2 = fork ();
      if (pid2>0)
         cout << wait (NULL) << endl;
    }
    else if (pid1>0)
   {
        int pid3 = fork ();
        if (pid3==0)
          cout << wait (NULL) << endl;
   }
   return 0;
}
```



Example 5

```
1   int main()
2   {
3      int cpid1 = fork();
4      if(cpid1>0)
5      {
6         cout<<wait(NULL)<< endl;
7      }
8      else if (cpid1==0)
9      {
10        int cpid2 = fork();
11        if(cpid2==0) {
12           cout << wait(NULL) << endl ;
13        }
14        if(cpid2>0) {
15           cout << wait(NULL) << endl ;
16        }
17     }
18     cout << "Hello Students!!!" << endl;
19     return 0;
20  }
```



fork() – Child inherits from the Parent

- The child process inherits the following attributes form the parent:
  - Environment
  - Open File Descriptor table
  - Signal handling settings
  - Nice value
  - Current working directory
  - Root directory
  - File mode creation mask (umask)

fork() – Child Differs from the Parent

- The child process differs from the parent process:
  - Different process ID (PID).
  - Different parent process ID (PPID).
  - Child has its own copy of parent's file descriptors

fork() – Reasons for Failure

- Maximum number of processes allowed to execute under one user has exceeded
- Maximum number of processes allowed on the system has exceeded
- Not enough swap space

System Call - wait()

- The wait system call suspends the calling process until one of its immediate children terminates, or until a child that is being traced stops because it has hit an event of interest.
- wait returns prematurely if a signal is received. If all children processes stopped or terminated prior to the call on wait, return is immediate.

```
SYNOPSIS
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

System Call - wait()

- If the call is successful, the process ID of the terminating child is returned
- If parent terminates, all its children are assigned process **init** as new parent. Thus the children still have a parent to collect their status and execution statistics
- **Zombie** process—a process that has terminated but whose exit status has not yet been received by its parent process or by **init**. (A process that has died but has yet not been reaped)
- **Orphan** process – a process that is still executing but whose parent has died. They do not become zombie rather are adopted by **init** process

Example - wait() system call

```
//fork2.cpp
int main()
{
   int cpid, status;
   cpid = fork();
   if (cpid == -1)
   {
        cout << "\n Fork failed\n";
        exit (1);
   }
   if (cpid == 0){
        cout<< "\n Hello I am child \n";
        exit(0);
        }
   else {
        wait(&status);
        cout << "\n Hello I am parent \n";
}
```

Example - wait() system call

```
1 #include <iostream>
2 #include <sys/types.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 using namespace std;
6 int main ()
7 {
8     int pid = fork();
9     if(pid==0){
10        cout << "I am a child process and my PID is : " << getpid();
11        cout << endl;
12        exit(0);
13    }else if (pid > 0){
14        int cpid = wait(NULL);
15        cout << "I am a parent process and my PID is: " << getpid();
16        cout << endl << "PID : " << pid << endl;
17        cout << "My Child PID is: " << cpid << endl;
18    }else{
19        cout << "Fork Failed";
20        cout << endl;
21    }
22    return 0;
23 }
24
```

Output

I am a child process and my PID is : 3701
I am a parent process and my PID is: 3700
PID : 3701
My Child PID is: 3701

Example - Zombie Process

```
1 #include <iostream>
2 #include <sys/types.h>
3 #include <unistd.h>
4 using namespace std;
5 int main()
6 {
7     int pid = fork();
8     if(pid>0)
9     {
10        sleep(10);
11        cout << "I am parent and my pid is : "<<getpid()<< endl;
12    }
13    else
14    {
15        cout << "Hey, I am a child and my pid is : "<<getpid()<< endl;
16        exit(0);
17    }
18    return 0;
19 }
20
21
```

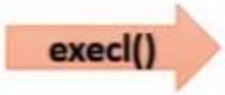
```
waqar@waqar-virtual-machine:~/Desktop/OS$ ./zomb &
[1] 5433
waqar@waqar-virtual-machine:~/Desktop/OS$ Hey, I am a child and my pid is : 5434
ps
    PID TTY          TIME CMD
   3006 pts/0    00:00:00 bash
   5433 pts/0    00:00:00 zomb
   5434 pts/0    00:00:00 zomb <defunct>
   5435 pts/0    00:00:00 ps
waqar@waqar-virtual-machine:~/Desktop/OS$ I am parent and my pid is : 5433
waqar@waqar-virtual-machine:~/Desktop/OS$
```

Example - Orphan Process

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

using namespace std;

int main() {
    pid_t p;
    p = fork();

    if (p == 0)
    {
        sleep(5); // Child goes to sleep, and in the meantime, the parent terminates.
        cout << "I am child having PID " << getpid() << endl;
        cout << "My parent PID is " << getppid() << endl;
    }
    else {
        cout << "I am parent having PID " << getpid() << endl;
        cout << "My child PID is " << p << endl;
    }

    return 0;
}
```

I am parent having PID 4192
My child PID is 4193
waqar@waqar-virtual-machine:~/Desktop/OS$ I am child having PID 4193
My parent PID is 1560

System Call exec()




```
int main()
{
pid n;
n=fork ();
if(n==0)
{
printf("child");
}
else
{
printf("Parent");
}
```

```
int main()
{
pid n;
n=fork ();
if(n==0)
{
execl("/bin/ps","ps",NULL)
}
else
{
printf("Parent");
}
```

System Call exec()

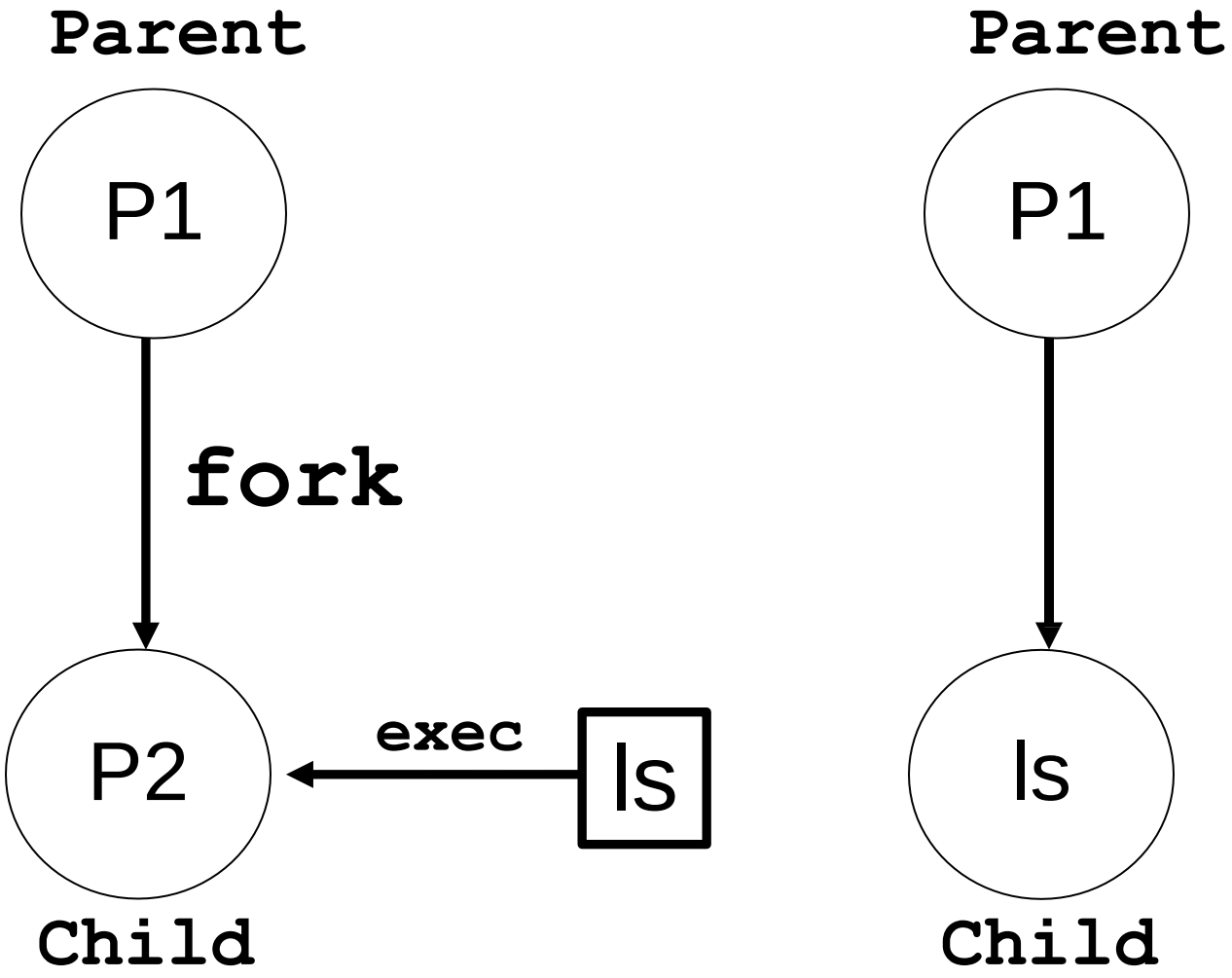
- Typically the exec() system call is used after a fork() system call by one of the two processes to replace the process memory space with a new executable program.
- The new process image is constructed from an ordinary, executable file.
- There can be no return from a successful exec() because the calling process image is overlaid by the new process image. Various Functions: *execl, execlp, execle, execv* and *execvp*

#include <unistd.h>

int execlp(const char *file,const char *arg0,...,const char *argn,(char *)0);

- `file` **is the path name of executable file which is going to override the caller process**
- **arg0 is the name given to child process.**
- **arg1 may be the options passed to the process.**
- **Last argument is null pointer NULL**

System Call exec()



Example - execlp() system call

```
//fork3.c
int main()
{
   int cpid, status;
   cpid  =  fork();
   if (cpid == -1)
   {
        printf ("\nFork failed\n");
        exit (1);
   }
   if (cpid == 0){
        execl("/bin/ls", "ls", NULL);
        //execl("/bin/ls", "ls", "-la", NULL);
        //execl("/bin/cal", "cal", NULL);
        exit(0);
   }
   else {
        wait(&status);
        printf ("\n Hello I am parent \n");
    }
```

Example - execlp() system call

```
#include <iostream>
#include <unistd.h>
using namespace std;
int main ()
{
    cout << "\nI prints 5 natural numbers" << endl;
    cout <<  "my pid is : " << getpid() << endl;
    for (int i  = 0 ; i <5 ; i++)
    {
        cout << i+1 << "    ";
    }
    cout << endl;
}
```

Compile this program as

g++ naturalNumbers.cpp –o nn

Example - execlp() system call

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
using namespace std;
int main ()
{
   int pid = fork();
   if (pid > 0)
   {
      cout << "I am a parent process and my PID is: " << getpid();
      cout << endl;
   }
   else if(pid==0)
   {
      cout << "I am a child process and my PID is : " << getpid();
      cout << endl;
      const char * BinaryPath = "./nn";
      execl(BinaryPath,BinaryPath,NULL);
   }
   return 0;
}
```

```
romana@ubuntu:~/Desktop/Codes$ g++ ExecE3.cpp -o e3
romana@ubuntu:~/Desktop/Codes$ ./e3
I am a parent process and my PID is: 5245
I am a child process and my PID is : 5246
romana@ubuntu:~/Desktop/Codes$
I prints 5 natural numbers
my pid is : 5246
1    2    3    4    5
```