UNIX multi-process programming

using fork

PID related Functions

- **getpid();**
- **getppid();**

- Use pid_t for variable declarations.

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

fork(): creating new process

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

pid_t
fork(void);
```

fork returns

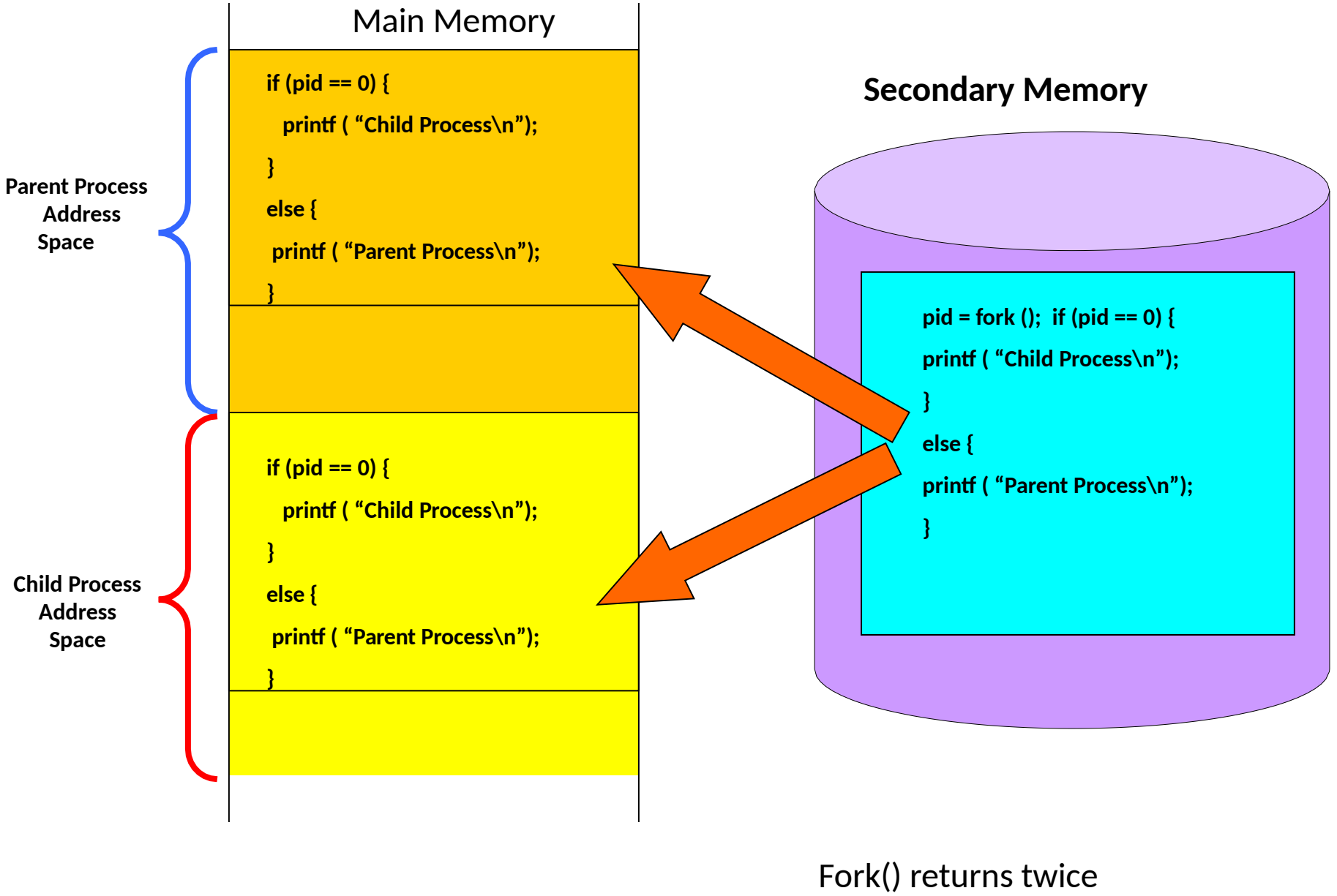
- **In parent : pid of the child**
- **In child : 0**
- **-1 : on errors**

wait() function

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

pid_t wait ( int
*status );
```

- Status receive the child termination status.
- Return value from wait is the pid that matches the return termination status.
- - 1 on error.



Fork() returns twice

fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{   fork();
    printf("Hello world!\n");
    return 0;
}
```

fork()

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

fork()

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
        fork();
        fork();
        fork();
        printf("hello\n");
        return 0;
}
```

Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

here n = 3, $2^3 = 8$

If we want to represent the relationship between the processes as a tree hierarchy it would be the following:

The main process: P0

Processes created by the 1st fork: P1

Processes created by the 2nd fork: P2, P3

Processes created by the 3rd fork: P4, P5, P6, P7

fork()

```
void forkexample()
{    if (fork() == 0)
                printf("Hello from Child!\n");
     else
                printf("Hello from Parent!\n");
}
int main()
{
     forkexample();
     return 0;
}
```

fork()

```
void forkexample()
{    int x = 1;
     if (fork() == 0)
                printf("Child has x = %d\n", ++x);
     else
                printf("Parent has x = %d\n", --x);
}
int main()
{    forkexample();
     return 0;
}
```

```
void dsc(int *a, int sz)
{
	int i,j,temp;
	for(i = 0; i < sz; i++)
	{
		for(j = i + 1; j < sz; j++)
		{
			if (a[i] < a[j])
			{
				temp = a[i];
				a[i] = a[j];
				a[j] = temp;
			}//if
		}//for_j
	}//for_i
	printf("Sorted array elements in descending order are\n");
	for(i = 0; i < sz; i++)
	{
		printf("%d\t", a[i]);
	}//for
printf("\n");
}//dsc
```

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
/* pid_t : is a long integer type data type...prototype in types.h */

pid_t num_pid,cpid;
int i;
void main() {

num_pid=fork(); /* return value of fork */
if(num_pid<0)
        printf("Error in fork execution");
else
if(num_pid==0) /* this is child process */  {
         printf("this is the child process id %d\n",getpid());
        printf("child completed\n"); }
```

```
else /* this is parent process */
{
printf("this is the parent id %d\n",getpid());
wait(NULL);
printf("********parent exiting after child completed\n");
}
exit(0);
}
```

Assignment –

- **Problem Statement – I :** Program where parent process sorts array elements in descending order and child process sorts array elements in ascending order.

```
        pid = fork();
        switch(pid)
        {
                case -1:
                        printf("Error in fork\n");
                exit(-1);
                case 0:
                        printf("Child Process\n");
                        asc(a,sz);
                exit(0);

                default:
                        wait(NULL);
                        printf("Parent Process\n");
                        dsc(a,sz);
                exit(0);
        }

return 0;
}
```

```
#include<unistd.h>
#include<stdio.h>
#include<sys/types.h>
#include<stdlib.h> //for exit

void asc(int *, int sz);
void dsc(int *, int sz);

int main()
{
        pid_t pid;
        int *a,sz,i;
        printf("Enter the size of the array\n");
        scanf("%d",&sz);
        a = (int*)malloc(sz * sizeof(int));
        printf("Enter elemment of %d size array\n",sz);
        for(i = 0; i < sz; i++)
        {
                printf("a[%d]:",i);
                scanf("%d",&a[i]);
        }
```

```
        pid = fork();
        switch(pid)
        {
                case -1:
                        printf("Error in fork\n");
                exit(-1);
                case 0:
                        printf("Child Process\n");
                        asc(a,sz);
                exit(0);

                default:
                        wait(NULL);
                        printf("Parent Process\n");
                        dsc(a,sz);
                exit(0);
        }

return 0;
}
```

```
void asc(int *a, int sz)
{
	int i,j,temp;
	for(i = 0; i < sz; i++)
	{
		for(j = i + 1; j < sz; j++)
		{
			if (a[i] > a[j])
			{
				temp = a[i];
				a[i] = a[j];
				a[j] = temp;
			}//if
		}//for_j
	}//for_i
	printf("Sorted array elements in ascending order are\n");
	for(i = 0; i < sz; i++)
	{
		printf("%d\t", a[i]);
	}//for
printf("\n");
}//asc
```

Zombie processes

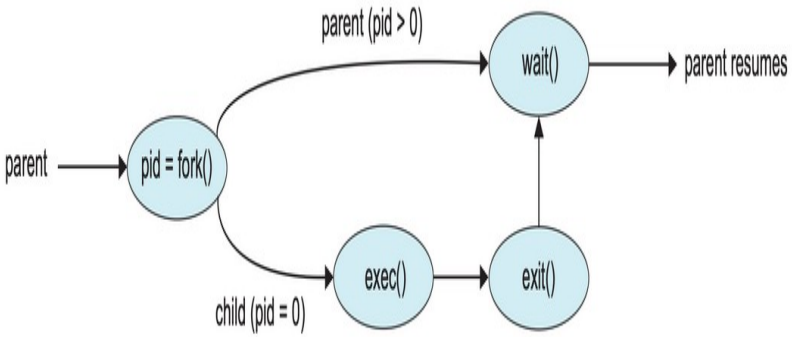


Figure 3.10 Process creation using the `fork()` system call.

A zombie process is a process whose execution is completed but it still has an entry in the process table.

Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status.

Once this is done using the wait system call, the zombie process is eliminated from the process table.

Zombie processes don't use any system resources but they do retain their process ID. .

The original process calls fork, **which creates a child process**. The child process then uses exec to start the execution of a new program. Meanwhile, the parent uses wait() to wait for the child process to finish.

Zombie processes

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main(void)
 {
   pid_t pid;
   if((pid=fork()) < 0)
                printf("\tfork error\n");
  else
  if(pid==0) {
                printf(" ");
                printf("child process id is %d\n",getpid());
  }
 else
   {
              sleep(2);
              printf("*****parent\n");
               system("ps -axj | tail");
    }
exit(0);
}
```

Child becomes Zombie as parent is sleeping ,when child process exits

Zombie processes output

```
/*****OUTPUT
child id is 3925
*****parent
   2  3801    0    0 ?        -1 S      0   0:00 [kworker/0:2]
   2  3804    0    0 ?        -1 S      0   0:00 [kworker/3:0]
   2  3847    0    0 ?        -1 S      0   0:00 [kworker/1:0]
   2  3894    0    0 ?        -1 S      0   0:00 [kworker/u8:1]
   2  3914    0    0 ?        -1 S      0   0:00 [kworker/3:2]
 2471  3924  3924  2471 pts/4    3924 S+    1001   0:00 ./a.out
 3924  3925  3924  2471 pts/4    3924 Z+    1001   0:00 [a.out] <defunct>
 3924  3926  3924  2471 pts/4    3924 S+    1001   0:00 sh -c ps -axj | tail
 3926  3927  3924  2471 pts/4    3924 R+    1001   0:00 ps -axj
 3926  3928  3924  2471 pts/4    3924 D+    1001   0:00 sh -c ps -axj | tail
 **/
```

Orphan Processes

- Orphan processes are those processes that are still running even though their parent process has terminated or finished.
- A process can be orphaned intentionally or unintentionally.
- in this case, **init** will adopt it and will wait for it.

In the following code, parent finishes execution and exits while the child process is still executing and is called an orphan process now.
However, the orphan process is soon adopted by init process, once its parent process dies.

```
#include<stdio.h> #include <sys/types.h> #include <unistd.h>
  int main()
{       int pid = fork();
      if (pid > 0){
         printf("in child process id %d\n“, getpid());
         printf(“parent  process id%d\n“, getppid());
}
        else if (pid == 0)
         {  sleep(10);
         printf(“child process id %d\n“, getpid());
         printf(“parent process  id %d\n“, getppid());
   }
      return 0;
}
```

Orphan processes output

```
/**output
child process id 4061
parent process id 4060

student@c04l0324:~$
child process id 4061
parent process id 1
**/
```