#include <stdio.h>

#include <sys/types.h>

#define MAX\_COUNT 200

void ChildProcess(void); /\* child process prototype \*/

void ParentProcess(void); /\* parent process prototype \*/

void main(void)

{

pid\_t pid;

pid = fork();

if (pid == 0)

ChildProcess();

else

ParentProcess();

}

void ChildProcess(void)

{

int i;

for (i = 1; i <= MAX\_COUNT; i++)

printf(" This line is from child, value = %d\n", i);

printf(" \*\*\* Child process is done \*\*\*\n");

}

void ParentProcess(void)

{

int i;

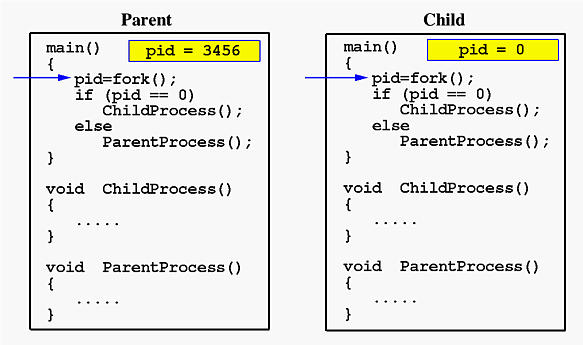
for (i = 1; i <= MAX\_COUNT; i++)

printf("This line is from parent, value = %d\n", i);

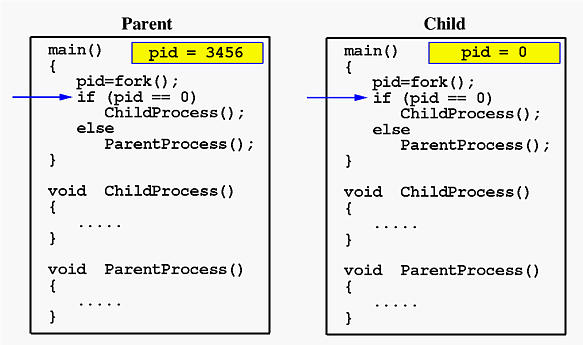
printf("\*\*\* Parent is done \*\*\*\n");

}

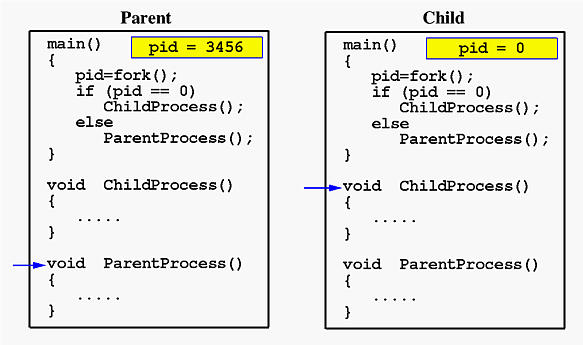
When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created. System call **fork()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the parent, since pid is non-zero, it calls function ParentProcess(). On the other hand, the child has a zero pid and calls ChildProcess() as shown below:



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of **MAX\_COUNT** should be large enough so that both processes will run for at least two or more time quanta. If the value of **MAX\_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

**BASIC FORK PROGRAM:**

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

int main ()

{

fprintf(stderr,"hello");

fork();

printf("hi \n");

}

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

int main (int argc , char \*argv[])

{

char str[] = "";

char ptr ='\0';

//char \*piece = strtok(str, "#");

char \*piece = strtok(str, &ptr);

//printf("%s \n ",piece); //This

while (piece !=NULL)

{

printf("%s \n ",piece);

piece =strtok(NULL,&ptr);

}

/\*

for (int i =0;i<20;i++) {

if (str[i]=='\0')

printf("\\0");

else

printf("%c",str[i]);

}

printf("\n");

\*/

return 0;

}

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

int main(int argc, char \*argv[], char\* env[],int a, int b)

{

int i;

for(i=0;i<argc;i++)

printf("argv[%d]: %s\n", i, argv[i]);

return 0;

}

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

int main()

{

pid\_t p;

printf("before fork\n");

p=fork();

if(p==0)

{

printf("I am child having id %d\n", getpid());

printf("My parent's id is %d\n", getppid());

}

else{

printf("My child's id is %d\n", p);

printf("I am parent having id %d\n", getpid());

}

printf("Common\n");

}

**ORPHAN**

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

int main()

{

pid\_t p;

p=fork();

if(p==0)

{

//sleep(20); //child goes to sleep and in the mean time parent terminates

printf("I am child having PID %d\n",getpid());

printf("My parent PID is %d\n",getppid());

}

else

{

//sleep(10);

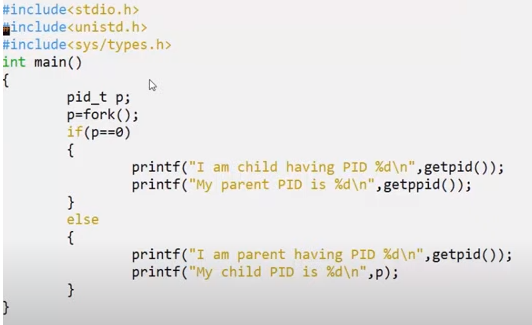
printf("I am parent having PID %d\n",getpid());

printf("My child PID is %d\n",p);

}

}

DexTutor

https://www.youtube.com/watch?v=DYDHNL\_AImo

**ZOMBIE**

//zombie.c

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<sys/types.h>

int main()

{

pid\_t t;

t=fork();

if(t==0)

{

printf("Child having id %d\n",getpid());

printf("Parent having id %d\n",getppid());

}

else

{

wait(NULL);

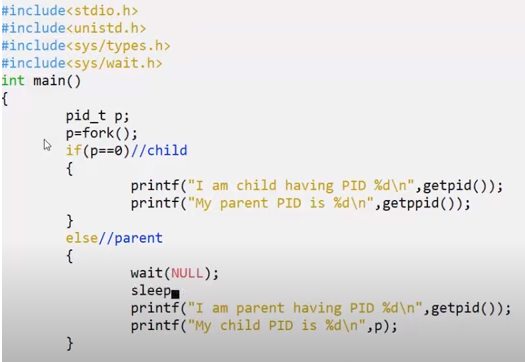
// sleep(15); // Parent sleeps. Run the ps command during this time

printf("Parent having id %d\n",getpid());

printf("My child PID is %d\n",t);

}

}



**FORK WITH SIGNAL**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <signal.h>

// Signal handler function

void signal\_handler(int signo) {

if (signo == SIGUSR1) {

printf("Child received SIGUSR1 signal.\n");

}

}

int main() {

// Install signal handler for SIGUSR1

signal(SIGUSR1, signal\_handler);

pid\_t child\_pid;

// Fork the first child

if ((child\_pid = fork()) == -1) {

perror("Error in fork");

exit(EXIT\_FAILURE);

}

if (child\_pid == 0) {

// Code for the first child process

// Child-specific code here

printf("First Child Process ID: %d\n", getpid());

// Send a signal to the parent process

kill(getppid(), SIGUSR1);

// Child exits

exit(EXIT\_SUCCESS);

} else {

// Code for the parent process

// Parent-specific code here

printf("Parent Process ID: %d\n", getpid());

// Wait for the first child to finish

wait(NULL);

// Fork the second child

if ((child\_pid = fork()) == -1) {

perror("Error in fork");

exit(EXIT\_FAILURE);

}

if (child\_pid == 0) {

// Code for the second child process

// Child-specific code here

printf("Second Child Process ID: %d\n", getpid());

// Child exits

exit(EXIT\_SUCCESS);

} else {

// Code for the parent process after forking the second child

// Parent-specific code here

// Wait for the second child to finish

wait(NULL);

printf("Parent Process after both children have exited.\n");

}

}

return 0;

}

2. BASIC FORK

#include <stdio.h>  
#include <sys/types.h>  
#include <unistd.h>  
int main() {  
pid\_t child\_pid;  
child\_pid = fork();  
if (child\_pid < 0) {  
// Fork failed  
 perror("Fork failed");  
 return 1;  
} else if (child\_pid == 0) {  
// Child process  
 printf("Child process: My PID is %d\n", getpid());  
} else {  
// Parent process  
 printf("Parent process: My PID is %d\n", getpid());  
 printf("Parent process: Child process ID is %d\n", child\_pid);  
}  
return 0;  
}