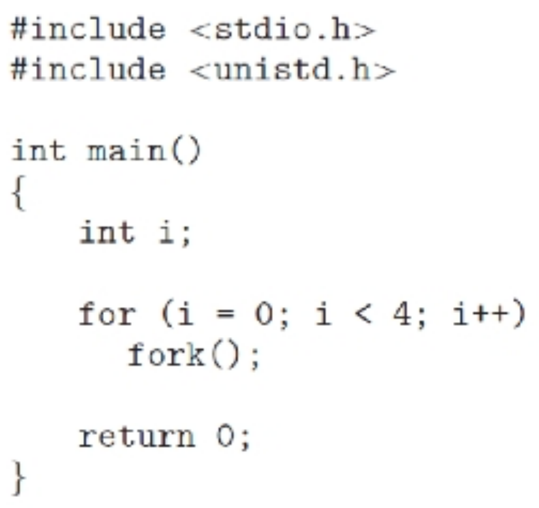
OS homework for week 4

by 艾语晨 PB18000227

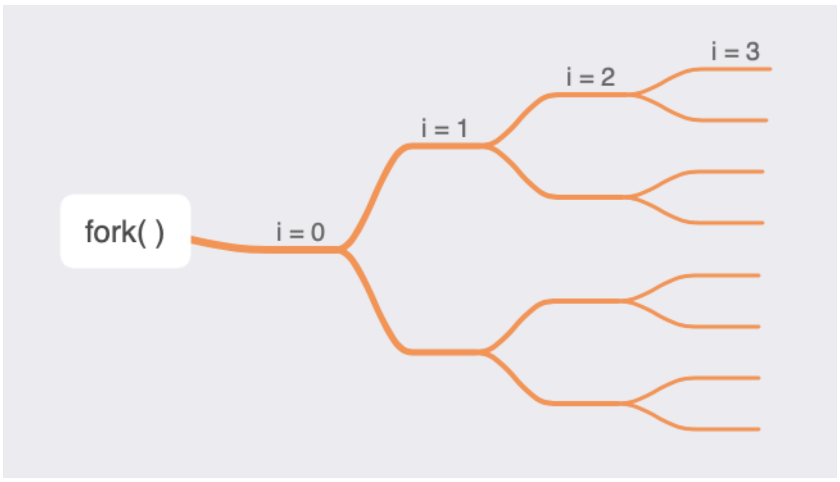
Problem 1

> Q : Including the initial parent process, how many processes are created by the program shown in Figure 1?



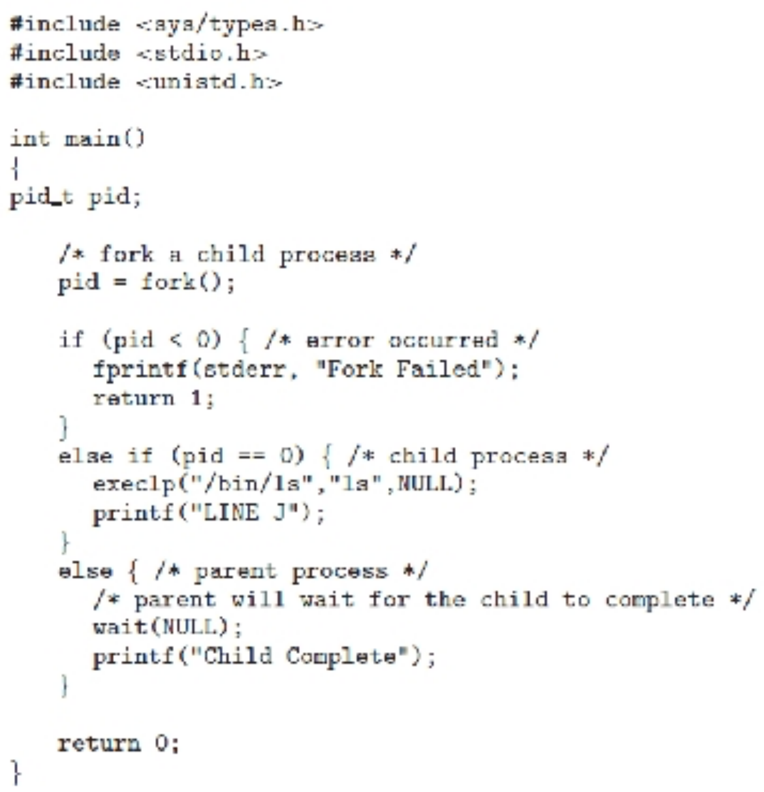
```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    for (i = 0; i < 4; i++)
      fork();

    return 0;
}
```

> A : 16 (Including the initial process of this program)
>
> Like the graph shows below, in each round of the circulation, both the parent(s) and the child(ren) calls `fork()`, thus doubles the process amount.



Problem 2

> Q : Explain the circumstances under which the line of code marked printf (“LINE J”) in Figure 2 will be reached.

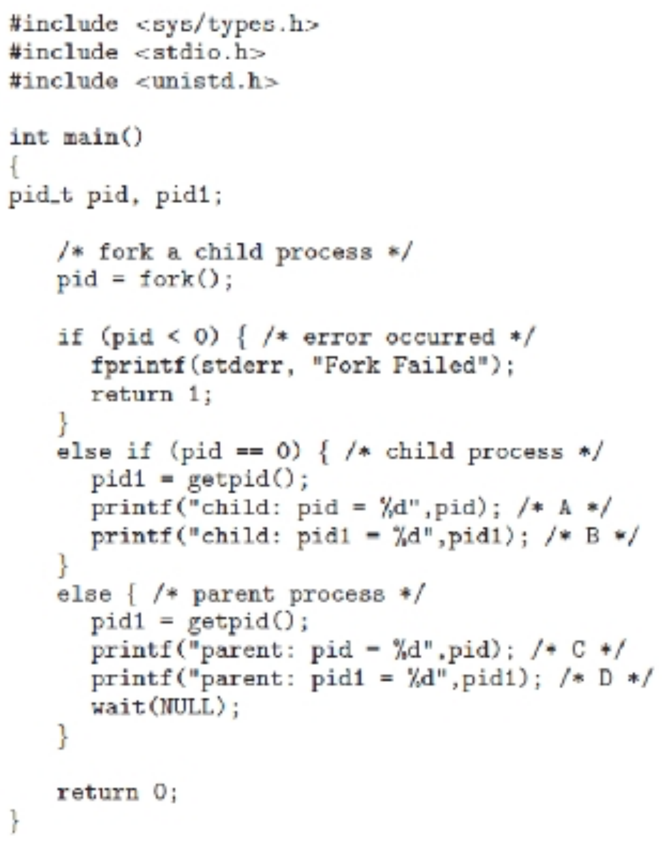


```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
       printf("LINE J");
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

> A : When the `execlp()` function call is failed to be called (probably due to some error), it will not reset the process but continue to execute the printf func, with a return value of `-1`.

Problem 3

> Q : Using the program in Figure 3, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)



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

int main()
{
pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
}
```

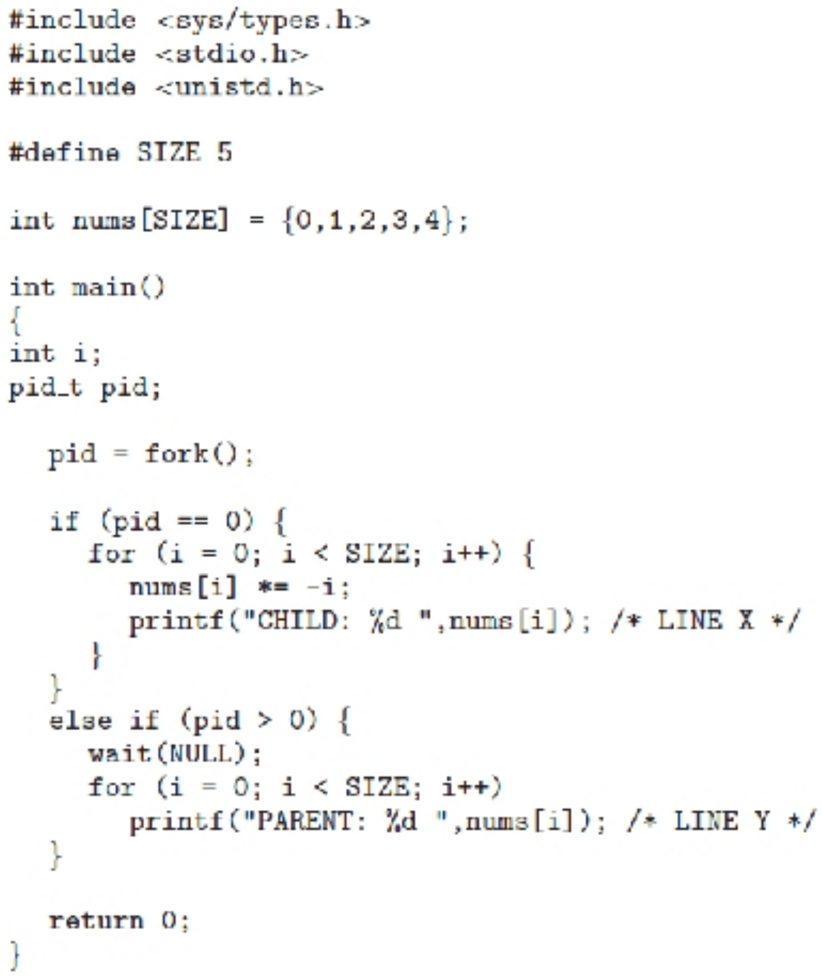
A :

The PID values are as the table below :

position	value
A	0
B	2603
C	2603
D	2600

Problem 4

Q : Using the program shown in Figure 4, explain what the output will be at lines X and Y.



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

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
int i;
pid_t pid;

   pid = fork();

   if (pid == 0) {
      for (i = 0; i < SIZE; i++) {
         nums[i] *= -i;
         printf("CHILD: %d ",nums[i]); /* LINE X */
      }
   }
   else if (pid > 0) {
      wait(NULL);
      for (i = 0; i < SIZE; i++)
         printf("PARENT: %d ",nums[i]); /* LINE Y */
   }

   return 0;
}
```

A :

```
CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16 # LINE X
CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16 # LINE Y
```

Note : Cause this program does *not* have a line break at its output...So the results were put altogether, yet it seems that there *is* a blank space between ':' and '%' and after 'd'.

> Explaination : The 'if' case for parent process has a `wait()` command, which means the child process would be executed first, changing the global variable `nums[]`, therefore the parent process would print out the same results as its child.

Problem 5

> Q : For the program in Figure 5, will LINE X be executed, and explain why.

```c
int main(void) {
    printf("before execl ...\n");
    execl("/bin/ls", "/bin/ls", NULL);
    printf("after execl ...\n");   /*LINE: X*/
    return 0;
}
```

> A : LINE X will *not* be actually executed (except for the condition that the function call `execl` is not executed cause an error has occurred).
>
> The reason is that the system call of `exec()` replaces the current process image with a new process image. While the new one is there to execute, the original process code or statistics are covered and will not be executed.

Problem 6

> Q : Explain why “terminated state” is necessary for processes.

> A : In a `fork(), exec(), wait()` , a parent process is blocked when using a `wait()` call, and only can wake up again only when receiving a SIGCHLD *from an `exit()` call made by the dying child process*, destroying the zombie process, and move on to its former execution.
>
> I mean, when the child process is terminated, the kernel notifies the parent of the child process about the termination of its child.
>
> So, if there isn't a "terminated state", the parent will never be waken up, and the zombie will never be eliminated.

Problem 7

> Q : Explain what a zombie process is and when a zombie process will be eliminated (i.e., its PCB entry is removed from kernel).

> A : When a process ends its execution and invokes `exit()`, or returns from `main()`, or terminates abnormally, it changes into terminated state, keeping its storage in the kernel-space memory to a minimum. Then it's called a *zombie process* before it's given a clean death by its parent's `wait()` call.
>
> A zombie process is eliminated either when its SIGCHLD is picked up by its parent process's signal handling routine, or become an orphan and be adopted by process 'init', before being destroyed by the periodically called `wait()`.