操作系统作业 2

1. 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;
}
```

Figure 1: Program for Question 1.

解：① $i = 0$，父进程创建一个子进程，共 2 个进程。之后这两个进程分别对i加一，这一步结束时$i = 1$

② $i = 1$，上一步的进程分别创建子进程，得到 4 个$i = 1$的进程。结束时$i = 2$

③ $i = 2$，上一步的进程分别创建子进程，得到 8 个$i = 2$的进程。结束时$i = 3$

④ $i = 3$，上一步的进程分别创建子进程，得到 16 个$i = 3$的进程。结束时$i = 4$

⑤ $i = 4$，停止执行。不考虑进程的消亡，最后总共存在的进程数就是创建的总进程数。全过程系统一共创建 16 个进程。

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

```
#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;
}
```

Figure 2: Program for Question 2.

解：printf（“LINE J”）；该语句若能被执行，必须满足$pid == 0$. 而题中变量pid的来源是$fork()$函数，该函数用于创建子进程：对于父进程，返回子进程的真实pid（正整数），在子进程中返回 0. 因此父进程中不可能被执行；同时该语句上方还有函数$execlp()$，若该函数执行成功则不会返回，自然也无法执行printf语句。因此。题中所给语句被执行的条件是$fork()$函数成功生成子进程，并且子进程执行$execlp()$函数失败。

3. 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.)

```
#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;
}
```

Figure 3: Program for Question 3.

解：

A: $child{:}\,pid = 0$
B: $child{:}\,pid1 = 2603$
C: $parent{:}\,pid = 2603$
D: $parent{:}\,pid1 = 2600$

4. 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;
}
```

Figure 4: Program for Question 4.

解：子进程的全局变量改变不影响父进程。因此输出为：

$LINE\ X$: $CHILD{:}0$ $CHILD{:}-1$ $CHILD{:}-4$ $CHILD{:}-9$ $CHILD{:}-16$

$LINE\ Y$: $CHILD{:}0$ $CHILD{:}1$ $CHILD{:}2$ $CHILD{:}3$ $CHILD{:}4$

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

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

Figure 5: Program for Question 5.

解：*execl*()函数如果成功执行，那么就不会返回该程序。所以若行X会被执行，就必须有*execl*()函数执行失败。

6. Explain why “terminated state” is necessary for processes.

解：进程终止状态是指：当进程完成执行最后语句并且调用exit()函数请求操作系统删除自身时，进程处于的一种状态。

当一个进程终止时，需要释放它所占用的资源和空间。对于子进程来说，在进程终止状态可以返回状态值到父进程表述结束状态是否正常。除此之外，对于父进程和操作系统来说，终止状态的返回值能够描述进程终止的原因（子进程使用了过多的资源，分配给子进程的任务不再需要，父进程已消亡）。操作系统需要据此来判断进程的运行状态和运行是否出错，并决定接下来的任务分配。因此，进程的结束状态非常必要。

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

解：僵尸进程是指某些子进程，它们已经运行结束，操作系统已经释放了其占用资源，但是它们的父进程在此时还尚未调用*wait*()函数，因此这些进程的进程标识符和它在进程表中的条目仍然存在。这些僵尸进程直到父进程调用*wait*()函数之后，它们的进程标识符和它在进程表中的条目才会被释放。

如果父进程没有调用*wait*()函数就终止了，这时子进程就成为孤儿进程。特别的，在Linux和UNIX系统中，init进程会定期调用*wait*()函数，收集任何孤儿进程的退出状态，并释放孤儿进程的进程标识符和它在进程表中的条目。