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;
}
```

一共创建了$2^4 = 16$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;
}
```

对于函数 execlp()，若函数执行成功则不会返回，若函数执行失败则会返回-1，因此如果程序执行到”printf(“LINE J”)”，说明 execlp()函数执行失败，原因有可能是没有找到将要执行的程序”/bin/ls”

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.)

```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;
}
```

对于父进程，pid 的值为调用系统调用 fork()的返回值，即创建子进程的 pid，所以 pid=2603；pid1 的值为调用 getpid()的返回值，即为父进程的 pid，所以 pid1=2600
对于子进程，pid 的值为调用系统调用 fork()的返回值，所以 pid=0；pid1 的值为调用 getpid()的返回值，所以 pid1=2603。所以 A、B、C、D 的值分别为 0、2603、2603、2600

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

```c
#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;
}
```

生成的子进程将父进程的全局变量克隆过来，所以 LINE X 处输出的结果将是：
CHILD:0
CHILD:-1

CHILD:-4
CHILD:-9
CHILD:-16

由于是克隆，父进程的全局变量值没有受到子进程的影响，所以 LINE Y 处的输出结果为

PARENT:0
PARENT:1
PARENT:2
PARENT:3
PARENT: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;
}
```

如果 execl()函数执行成功则不会返回，也就不会执行到 LINE X，因为系统调用 execl()来执行程序，会执行到程序中”return”或”exit()”命令，使进程终止运行，也就无法返回到原来调用 execl()函数的程序了。但如果 execl()函数没能成功执行，那么将返回-1，也就可以执行到 LINE X 了。

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

终止状态标志进程结束运行，对于父进程而言，子进程的终止使父进程得以从 wait()挂起的状态转换为执行状态，保证了进程之间的通信要求；对于子进程而言，子进程结束运行进入终止状态向父进程发去 SIGCHLD 信号，使父进程将子进程占用的系统资源彻底释放，杀死处于僵尸状态的进程，有利于烯烃管理系统资源，防止系统资源被僵尸进程占用。

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

僵尸进程是当子进程比父进程先结束，而父进程又没有回收子进程，释放子进程占用的资源，此时子进程将成为一个僵尸进程。

僵尸进程的消除分为两种情况，第一种情况是子进程比父进程先结束，此时子进程结束运行进入终止状态向父进程发去 SIGCHLD 信号，父进程的 signal handler 接收到信号后，将发送该信号的子进程所占用的所有资源回收，完成僵尸进程的消除；第二种情况是子进程比父进程后结束，此时子进程被 init 接管，init 会周期性的调用 wait()来消除僵尸进

程，具体过程同第一种情况。

8.Explain what data will be stored in user-space and kernel-space memory for a process.

内核空间存储的数据包括：内核数据结构（PCB，PCB 又包括 Process state，Program counter，CPU register，CPU scheduling information，Memory-management information，I/O state information etc.）、内核代码、设备驱动

用户空间存储的数据包括：程序的代码段、数据段（全局变量、静态变量）、堆（动态分配的空间）、栈（参数、局部变量）、contents of register.

9.Explain the key differences between exec() system call and normal function call.

系统调用是内核访问资源的请求，而函数调用是程序执行特定任务的请求。当程序需要与内核通信时使用系统调用，而函数调用用于调用程序中的特定函数。