

# 第三次作业

## 题目一解答：

LINE A 输出为：“PARENT: value = 5”

因为使用 `fork()` 创建一个子进程，在子进程中，`pid` 的值为 0；父进程中 `pid` 的值为子进程的进程号 ( $>0$ )；所以将执行不同的 `if-else` 分支；而且，子进程和父进程均拥有独立的数据段，对于全局变量的修改不会影响到对方。尽管两个数据段在子进程刚刚创建的时候完全相同，但是在 `if-else` 分支后，子进程中 `value=20`、父进程 `value=5`；

## 题目二解答：

### 1) 同步和异步通信

同步通信：

执行效率比较低，更耗费时间，但是有利于对流程的控制，更容易捕获、处理异常；对于用户而言保证了同时同步性，提供更好的用户体验；

异步通信：

执行效率高，节省时间；对用户而言，可以在等待时间处理进行其他工作，但是占用更多资源，不利于对进程的控制；对用户而言，可能在发出信息之后无法立刻得到反馈，甚至由于某种突发错误而导致结果丢失；

### 2) 自动和显示缓冲

自动缓冲：提供了优先或者无限容量的队列，消息可以在队列中等待，在队列满之前不阻塞发送者；对于自动缓冲，存在浪费大量内存与系统资源的可能；但是对于用户，采用自动缓冲可以使得发送消息更及时流畅；

显示缓冲：也就是零容量队列缓冲，这样每次发送消息时都会被阻塞一段时间，但是这样避免了系统资源的浪费；

### 3) 复制传送和引用传送

复制传送：接收者不能够对传送参数进行修改，这样保证了参数的不可更改性，保持了系统间通信传输的一致性；对用户而言，不能修改参数可能会形成不便；

引用传送：接收者能够改变参数，比如，用户可以根据接收到的集中应用的分支版本；

### 4) 固定大小和可变大小信息

固定大小：如果进程间通信的信息是定长的，那么将会简化系统实现，降低难度；但对于用户变化的需求难以适应，使得编程难度增加；

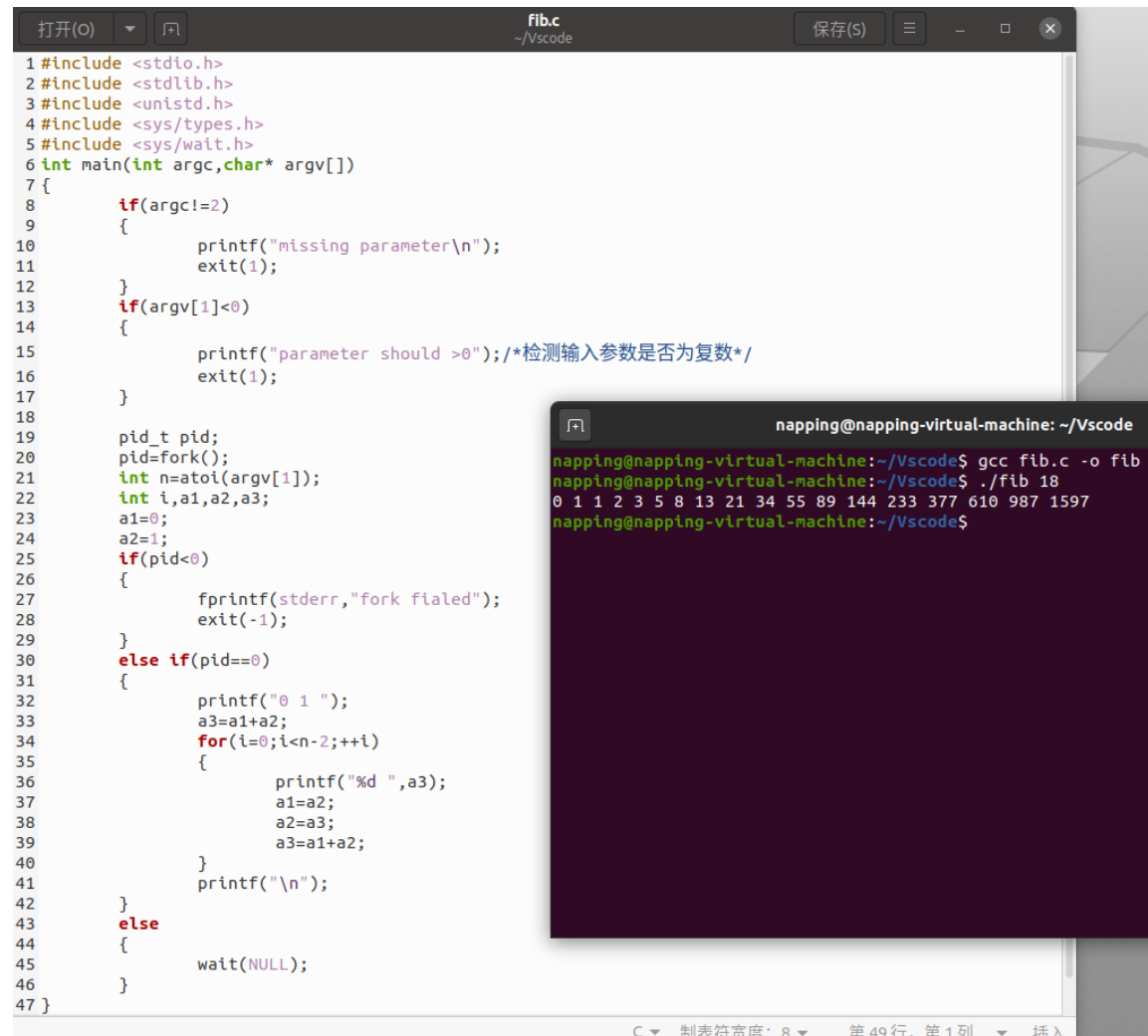
可变大小：可变大小信息要求更为复杂的系统级设计架构；但对于用户而言，使用更为方便了；

### 题目三解答：

首先进行参数数目是否合法，然后正确性检测，避免接收负数号码；

使用 atoi 函数自动将 char\* 指向的参数保存在 i 中，之后使用 fork 产生子进程，如果 pid<0，则说明 fork 失败；否则，在子进程中完成 Fibonacci 数列的输出，父进程中调用 wait 函数，由于使用了 wait，需要包含头文件<sys/wait.h>;

程序代码如下：



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 int main(int argc, char* argv[])
7 {
8     if(argc!=2)
9     {
10         printf("missing parameter\n");
11         exit(1);
12     }
13     if(argv[1]<0)
14     {
15         printf("parameter should >0");/*检测输入参数是否为复数*/
16         exit(1);
17     }
18     pid_t pid;
19     pid=fork();
20     int n=atoi(argv[1]);
21     int i,a1,a2,a3;
22     a1=0;
23     a2=1;
24     if(pid<0)
25     {
26         fprintf(stderr,"fork fialed");
27         exit(-1);
28     }
29     else if(pid==0)
30     {
31         printf("0 1 ");
32         a3=a1+a2;
33         for(i=0;i<n-2;++i)
34         {
35             printf("%d ",a3);
36             a1=a2;
37             a2=a3;
38             a3=a1+a2;
39         }
40         printf("\n");
41     }
42     else
43     {
44         wait(NULL);
45     }
46 }
47 }
```

```
napping@napping-virtual-machine: ~/Vscode
napping@napping-virtual-machine:~/Vscode$ gcc fib.c -o fib
napping@napping-virtual-machine:~/Vscode$ ./fib 18
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
napping@napping-virtual-machine:~/Vscode$
```

### 题目四解答：

#### Linux 系统中进程状态：

可运行态：

运行态和就绪态的合并，表示进程正在运行或准备运行，Linux 中使用 TASK\_RUNNING 宏表示此状态。

浅度睡眠态：

进程正在睡眠（被阻塞），等待资源到来是唤醒，也可以通过其他进程信号或时钟中断唤醒，进入运行队列。Linux 使用 TASK\_INTERRUPTIBLE 宏表示此状态。

深度睡眠态：

其与浅度睡眠基本类似，但有一点就是不可其他进程信号或时钟中断唤醒。Linux 使用 TASK\_UNINTERRUPTIBLE 宏表示此状态。

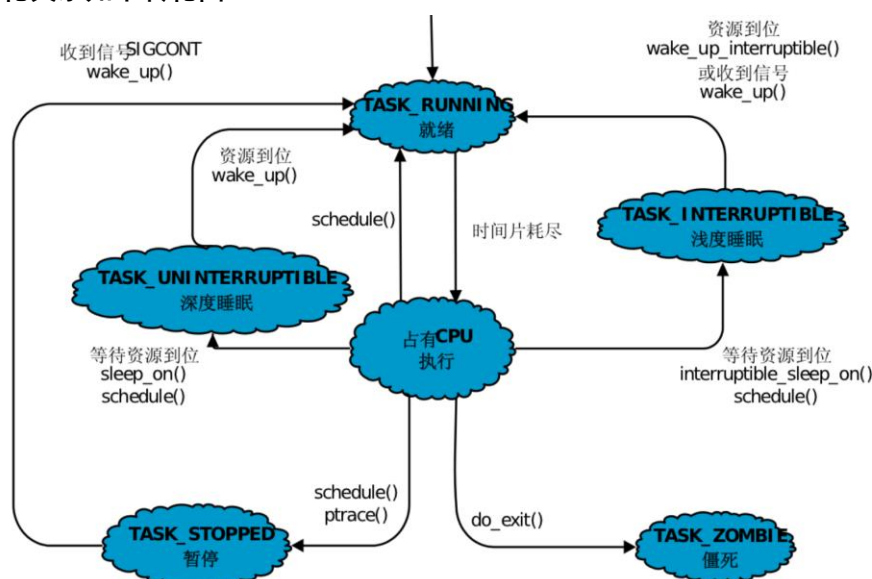
暂停状态：

进程暂停执行接受某种处理。如正在接受调试的进程处于这种状态，Linux 使用 TASK\_STOPPED 宏表示此状态。

僵死状态：

进程已经结束但未释放 PCB，Linux 使用 TASK\_ZOMBIE 宏表示此状态。

转化关系如下转化图：



对比基本流程状态图：

- 1) 减少了 ready 状态，将其与 running 合并为 TASK\_RUNNING；
- 2) 将 waiting 状态细分为浅度睡眠和深度睡眠，前者可被资源到来、时钟中断、其他进程信号唤醒、而后者只能被资源到来唤醒；
- 3) 增加暂停状态，表示正在接受某种处理；
- 4) 僵死状态对应 terminated，但是僵死状态并没有完全终止，因为进程还没释放 PCB；

题目五解答：

上下文切换的过程：

内核发起进程切换请求，开始进行上下文切换；首先保存现场，将程序计数器和 CPU 中的寄存器内的值保存，同时保存进程状态、编号、打开文件和相关虚拟地址；被中断进程的 PCB 保存好后，开始用新进程的 PCB 进行现场数据恢复，将保存的寄存器的值覆盖 CPU 原有寄存器的值和其他进程信息，根据 PC 地址开始执行新的进程，直到该进程也被切换，这是一个完整的上下文切换流程。

上下文切换的作用：

个人理解，通过上下文切换，通过调度算法给每个进程执行一个时间片的机会，让单核

CPU 在宏观上实现了“多任务”并发处理，在有些进程暂时阻塞时，可以切换到其他进程执行，使得系统资源得到充分利用；但是上下文切换本身也是有 CPU 周期消耗的，一次上下文切换约  $10^{-4} \sim 10^{-3}$  秒，但如果上下文切换很频繁的话也是很大的消耗。即便现代计算机正在逐渐降低上下文切换所占用的 CPU 时间，但这也仅是在 CPU 时钟周期降低，处理速度加快的情况下，而不是提升了上下文切换的效率。

## 题目六解答：

### 进程：

进程是允许某个并发执行的程序在某个数据集合上的运行过程，由代码段、用户数据段及 PCB 共同组成的执行环境，代码段存放被执行的机器指令，用户数据段存放进程在执行时直接进行操作的用户数据。在没有引入线程概念的操作系统中，进程是独立运行和资源调度的基本单位。

“Process is a running instance of a program”

### 与程序的联系：

进程是程序的一次执行，而进程总是对应至少一个特定的程序。一个程序可以对应多个进程，同一个程序可以在不同的数据集合上运行，因而构成若干个不同的进程。几个进程能并发地执行相同的程序代码，而同一个进程能顺序地执行几个程序。

### 与程序的区别：

- 1) 程序是静态的，进程是动态的，程序是存储在某种介质上的二进制代码，可以存在于任何位置；进程对应了程序的执行过程，只存在于内存，系统不需要为一个不执行的程序创建进程，一旦进程被创建，就处于不断变化的动态过程中，对应了一个不断变化的上下文环境。
- 2) 程序是永久的，进程是暂时的。程序的永久性是相对于进程而言的，只要不去删除它，它可以永久存储在介质当中，而进程只在一段时间内在内存中存在，运行完成后便消失。