习题3.3

a.

新建态 → 就绪/挂起态

新建态 → 就绪态

创建一个新进程A时，该进程要么加入就绪队列，要么加入就绪/挂起队列。

就绪/挂起态 → 就绪态

若内存中没有就绪进程，则操作系统需要调入一个进程继续执行。此外，处于就绪/挂起态的进程比处于就绪态的任何进程相比优先级更高时也可以进行这种转换，例如进程B处于就绪态，若处于就绪/挂起态的进程A的优先级高于它，则进程B将被换出，进程A转换为就绪态。

就绪态 → 就绪/挂起态

通常，操作系统更倾向于挂起阻塞态进程而非就绪态进程，但若释放内存来得到足够空间的唯一方法是挂起一个就绪态进程时，这种转换也是必须的。因此在一些情况下，不得不挂起一个就绪态的进程来获得足够的内存空间。

就绪态 → 运行态

需要选择一个新进程运行时，操作系统选择一个处于就绪态的进程运行。

运行态 → 就绪/挂起态

通常，当一个运行进程的分配空间到期后，它将转换到就绪态，但在阻塞/挂起队列中具有较高优先级的进程不再被阻塞时，操作系统会抢占这个进程，直接把这个运行进程转换到就绪/挂起队列中，并释放一些内存空间。

运行态 → 就绪态

当正在运行的进程已到达“允许不中断执行”的最大时间段时，或操作系统给不同的进程分配不同的优先级，例如进程A以一个给定优先级在运行，具有更高优先级的进程B处于阻塞状态，操作系统知道进程B等待的事件已经发生时将进程B转换为就绪态，进程A将被中断。还有一种情况是进程自愿释放对处理器的控制，例如一个周期性进行记账和维护的后台进程。

运行态 → 阻塞态

进程请求其必须等待某些事件时，它将会进入阻塞态。例如进程可能请求操作系统的一个服务，但操作系统无法立即执行，或请求一个无法立即得到的资源，或是I/O操作等。

运行态 → 退出态

若当前正在运行的进程表示自身已完成或取消时，它将被操作系统终止。

阻塞/挂起态 → 就绪/挂起态

当处于阻塞/挂起态的进程等待的事件发生时，它就可以转换到就绪/挂起态。

阻塞/挂起态 → 阻塞态

一个进程终止后，释放掉了一些内存空间，若阻塞/挂起队列中有一个进程的优先级比就绪/挂起队列中任何进程的优先级都要高，且操作系统有理由相信该阻塞/挂起的进程等待的事件很快就会发生，这时候就有可能将其调入内存。

阻塞态 → 就绪态

处于阻塞态的进程所等待的事件发生时，它将会转换为就绪态。

阻塞态 → 阻塞/挂起态

若没有就绪进程，则至少需要换出一个阻塞进程，以便为另一个未阻塞进程腾出空间。当操作系统需要确定当前正在运行的进程，或就绪进程，为了维护基本的性能而需要更多的内存空间，则会挂起一个处于阻塞态的进程。

各种状态 → 退出态

典型情况下，一个进程的运行终止，要么是它已完成运行，要么是出现了一些错误条件。在一些操作系统中，进程可被父进程终止，或在父进程终止时终止。若这种情况运行，进程在任何状态下都有可能转换为退出态。

运行态 → 阻塞/挂起态

一般情况下，正在运行的程序遇到需要等待的事件时会进入阻塞态，但若是操作系统有理由相信它等待的事件不会马上发生且需要挂起一些程序来获得足够的内存空间时会将其挂起。

b.

任何其他6个状态都不可能转换为新建态。

退出态不能转换为任何其他状态。

新建态 → 退出态

新建一个进程操作系统便分配了足够的内存，除非该进程马上终止，否则不会立马退出。

新建态 → 运行态

新建态需要进行一定的调度做好准备后才能进入运行态。

新建态 → 阻塞/挂起态

新建态 → 阻塞态

新创建的进程需要运行且遇到需要等待的事件才能进入阻塞（阻塞/挂起）态。

就绪/挂起态 → 阻塞/挂起态

就绪/挂起态 → 阻塞态

只有运行的程序遇到需要等待的事件时它才会被阻塞。

就绪/挂起态 → 运行态

就绪/挂起态的进程是已在外存中的，需要载入内存才能运行，因此它必须先转换为就绪态。

就绪态 → 阻塞/挂起态

就绪态 → 阻塞态

只有运行的程序遇到需要等待的事件时它才会被阻塞，处于就绪态的程序必须先进入运行态才有可能进入阻塞态。

阻塞/挂起态 → 就绪态

处于阻塞挂起态的进程进入就绪态需要两个条件：等待的事件发生和换入内存。因此它需要先转为就绪/挂起态或阻塞态，不能直接转换为就绪态。（一般来说这两种情况总是先后发生）

阻塞/挂起态 → 运行态

阻塞/挂起态的程序只有先进入就绪态才能开始运行。

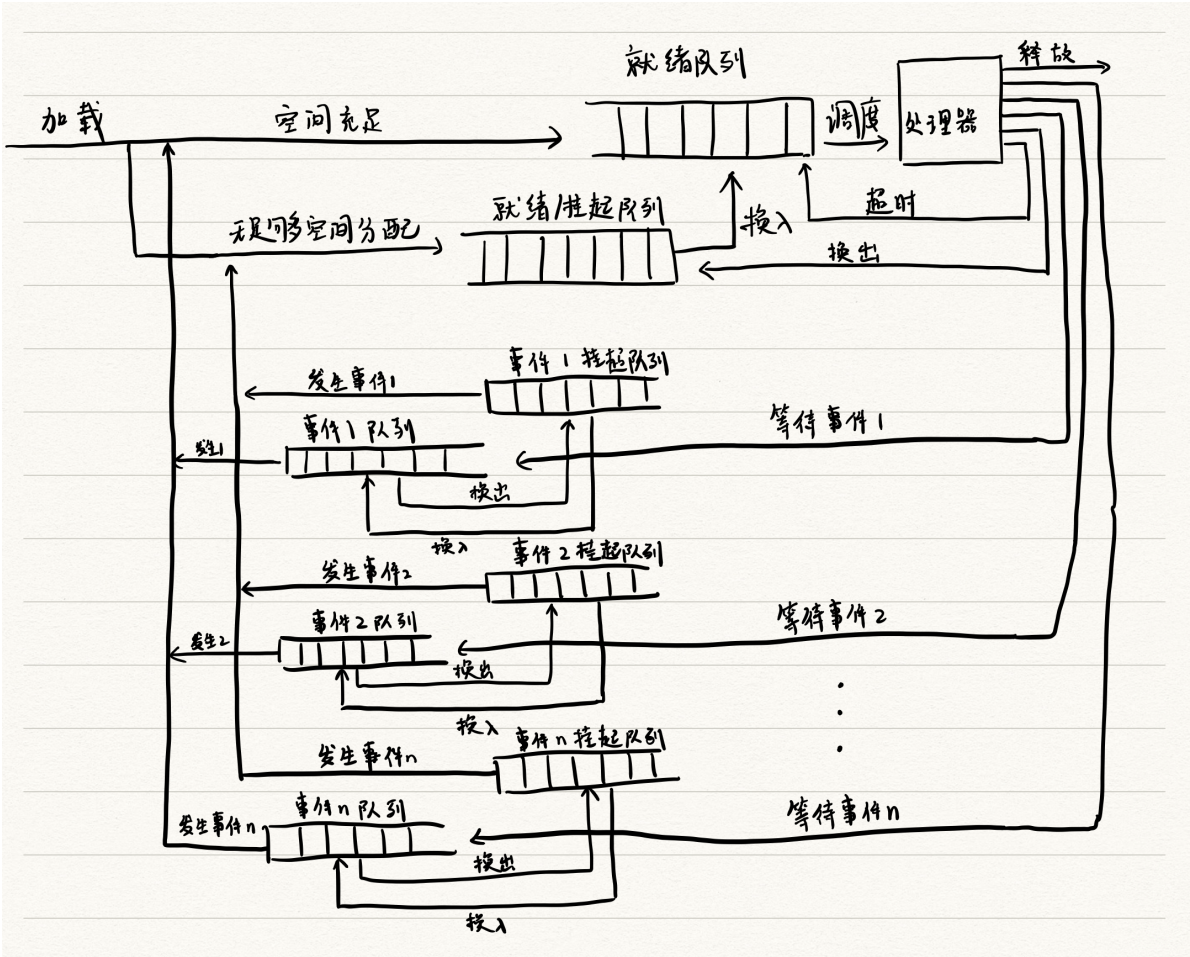
阻塞态 → 就绪/挂起态

在内存中阻塞的程序等待的事件发生时会进入就绪态，当需要挂起它来获得一定内存空间时会进入阻塞/挂起态，任何一种情况都不能直接进入就绪/挂起态。

阻塞态 → 运行态

阻塞态的程序等待事件发生时会进入就绪态，需要等待操作系统分配资源后才能进入运行态。

习题3.4



习题3.6

a.

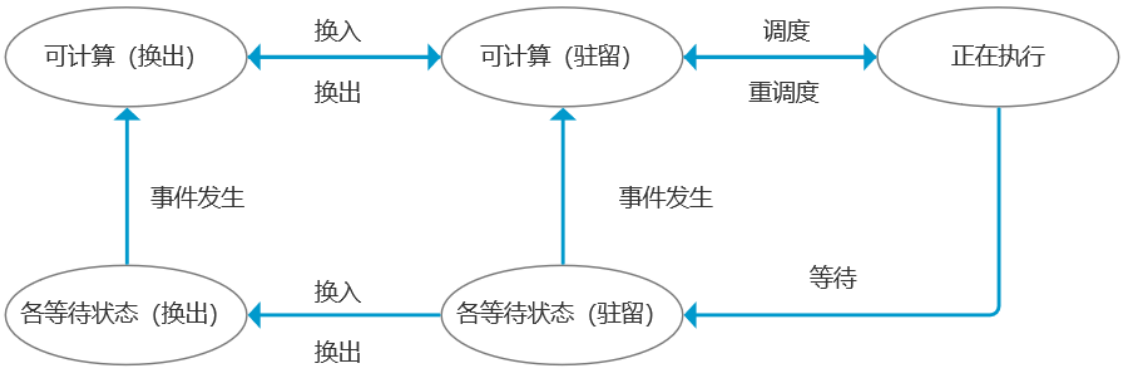
每一种等待都有一个单独的队列与其相关联（如习题3.4图）。当影响某一等待进程的事件发生时，把等待进程分成不同的队列就减少了定位这一等待所花费的时间。例如，当一个页错误完成时，调度程序就可以在页错误等待队列找的该等待的进程。根据等待的事件划分等待状态有利于提高系统的效率。

b.

在这些状态下，允许进程换出只会使效率更低。例如，发生页冲突等待时，进程引用了另一个正处于页面失效等待的进程所等待的共享页或进程正在读入/写出的私有页，这时把进程换出是没有意义的。

c.

状态转移图如下：



引发状态转换的动作或条件：

正在执行 → 各等待状态（驻留）

正在执行的进程请求其必须等待某些事件时将会转换为对应的等待状态。

正在执行 → 可计算（驻留）

当正在运行的进程已到达“允许不中断执行”的最大时间段时，或操作系统给不同的进程分配不同的优先级，正在执行的进程将被迁出到可计算队列。

可计算（驻留） → 正在执行

需要选择一个新进程运行时，操作系统选择调度一个可计算（驻留）的进程运行。

可计算（驻留） → 可计算（换出）

系统内存不足，释放内存来得到足够空间的唯一方法是换出一个可计算进程时。

各等待状态（驻留） → 可计算（驻留）

当该进程所等待的事件发生时，它将会转换为可计算（驻留）。

各等待状态（驻留） → 各等待状态（换出）

若没有可计算（驻留）进程，需要换出一个驻留的处于等待状态的进程，以便为其他不处于等待状态的进程腾出空间。

各等待状态（换出） → 可计算（换出）

当该进程所等待的事件发生时。

可计算（换出） → 可计算（驻留）

若内存中没有可计算进程或可计算进程的队列较空时，操作系统会换入一个进程。

编程任务

1.

The program is as belows:

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

```
#include <sys/types.h>
int main(){
    int p1,x=100;
    p1=fork();
    sleep(3);
    if(p1>0){
        printf("From parent: %d\n", x);
    }
    else{
        printf("From child: %d\n", x);
    }
    return 0;
}
```

The output is:

```
From child: 100
From parent: 100
```

The value of x is the same as the parent process.

Rewrite the program as belows:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
    int p1,x=100;
    p1=fork();
    sleep(3);
    if(p1>0){
        x=x+7;
        printf("From parent: %d\n", x);
    }
    else{
        x=x-7;
        printf("From child: %d\n", x);
    }
    return 0;
}
```

The output is:

```
From parent: 107
From child: 93
```

According to the output above, we can see that fork() creates a copy of parent process, and then the parent process and the child process runs in their own private address space respectively, both of them can not reach the memory of each other. So, each maintain their own copy of variables.

2.

The program is as belows:

```
#include <stdio.h>
#include <unistd.h>
#include <syspes.h>
#include <fcntl.h>
int main(){
    char filename[]="test.txt";
    int f=open(filename,O_RDWR);
    int p1=fork();
    if(p1>0){
        printf("From parent: %d\n", f);
    }
    else{
        printf("From child: %d\n", f);
    }
    return 0;
}
```

The output is:

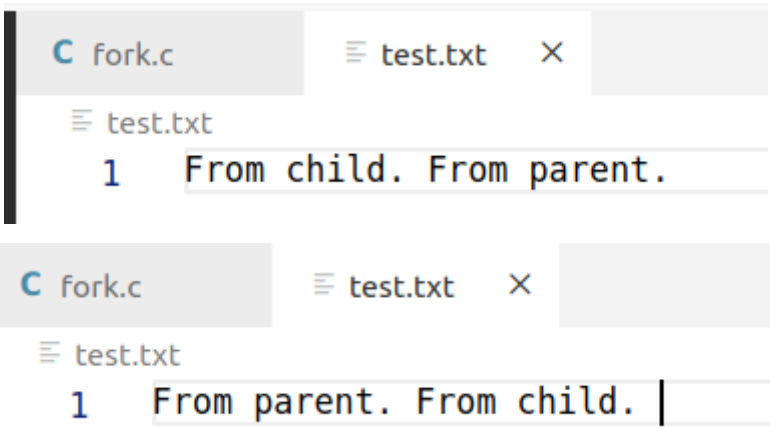
```
From child: 3
From parent: 3
```

The value of the file descriptor is the same as the parent process. Both child and parent can access the file descriptor opened using **open()** .

Rewrite the program as belows:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>
int main(){
    char filename[]="test.txt";
    int f=open(filename, O_RDWR, O_APPEND);
    int p1=fork();
    if(p1>0){
        char str[]="From parent. ";
        int w=write(3,(void *)str, strlen(str));
    }
    else{
        char str[]="From child. ";
        int w=write(3,(void *)str, strlen(str));
    }
    return 0;
}
```

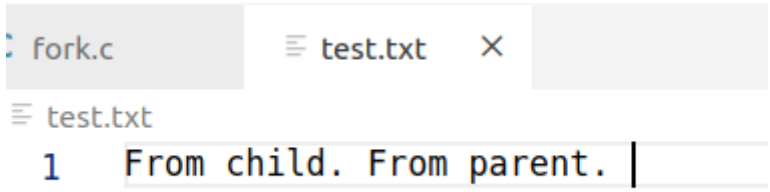
Both output as below happen:



Add **wait()** to the program above:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>
#include <wait.h>
int main(){
    char filename[]="test.txt";
    int f=open(filename, O_RDWR, O_APPEND);
    int p1=fork();
    if(p1>0){
        char str[]="From parent. ";
        wait(NULL);
        int w=write(3,(void *)str, strlen(str));
    }
    else{
        char str[]="From child. ";
        int w=write(3,(void *)str, strlen(str));
    }
    return 0;
}
```

The output always is:



According to the output above, we can see that both child and parent can access the file descriptor opened using
open(), both are able to write to the file but the order in which they do is un-deterministic, if we use wait(), the process would execute by the order we want.

3.

Try **execl()**:

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

int main(){
    int p1=fork();

    execl("/bin/ls", "ls", "-l", NULL);
    printf("From main.\n");
    return 0;
}
```

The output is:

```
总用量 28
-rwxrwxr-x 1 wenny wenny 19472 3月  28 15:36 fork
-rw-rw-r-- 1 wenny wenny   155 3月  28 15:36 fork.c
-rw-rw-r-- 1 wenny wenny    25 3月  28 14:22 test.txt
总用量 28
-rwxrwxr-x 1 wenny wenny 19472 3月  28 15:36 fork
-rw-rw-r-- 1 wenny wenny   155 3月  28 15:36 fork.c
-rw-rw-r-- 1 wenny wenny    25 3月  28 14:22 test.txt
```

We can see that the output is NOT containing "From main." which is supposed to print in **main**, because exec() would create a new process by covering the old process. Thus the codes behind the line can not be run.

Try **execv()**:

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

int main(){
    int p1=fork();
    char* args[]={"ls","-l",NULL};
    execv("/bin/ls", args);
    printf("From main.\n");
    return 0;
}
```

The output is:

```
总用量 28
-rwxrwxr-x 1 wenny wenny 19584 3月  28 15:44 fork
-rw-rw-r-- 1 wenny wenny   220 3月  28 15:44 fork.c
-rw-rw-r-- 1 wenny wenny    25 3月  28 14:22 test.txt
总用量 28
-rwxrwxr-x 1 wenny wenny 19584 3月  28 15:44 fork
-rw-rw-r-- 1 wenny wenny   220 3月  28 15:44 fork.c
-rw-rw-r-- 1 wenny wenny    25 3月  28 14:22 test.txt
```

The same to execl().

Because execl() and execv() are nearly identical, the difference is that passing arguments to execl() is by list and to execv() is by vector.

Try **execlp()**:

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

int main(){
    int p1=fork();
    execlp("ls", "ls", "-l", NULL);
    printf("From main.\n");
    return 0;
}
```

The output is :

```
总用量 28
-rwxrwxr-x 1 wenny wenny 19584 3月  28 15:44 fork
-rw-rw-r-- 1 wenny wenny   220 3月  28 15:44 fork.c
-rw-rw-r-- 1 wenny wenny    25 3月  28 14:22 test.txt
总用量 28
-rwxrwxr-x 1 wenny wenny 19584 3月  28 15:44 fork
-rw-rw-r-- 1 wenny wenny   220 3月  28 15:44 fork.c
-rw-rw-r-- 1 wenny wenny    25 3月  28 14:22 test.txt
```

The execvp() are similiar. We just provide execlp() with filename, NOT including the path. It can search the directory of environment variables to find the file.

To show it is different from excl(), we run this code:

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

int main(){
    int p1=fork();
    execl("ls", "ls", "-l", NULL);
    printf("From main.\n");
    return 0;
}
```

The output is:

```
From main.
From main.
```

Due to the process can not find the target(It was NOT trying to search the directory of environment variables) .

The execle() and execvpe() could be passed an environment variable table, so it could search the table to find the target file.

The different variants gives the users more choices to execute the program in the way they like.

4.

The program is:

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int main(){
    int p1=fork();
    if(p1>0){
        printf("From parent.\n");
        int w=wait(NULL);
        printf("wait: %d\n", w);
    }
    else{
        sleep(3);
        printf("From child.\n");
    }
    return 0;
}
```

The output is:

```
From parent.
From child.
wait: 29670
```

wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.

Rewrite the program:

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int main(){
    int p1=fork();
    if(p1>0){
        printf("From parent.\n");
        int w=wait(NULL);
        printf("wait: %d from parent.\n", w);
    }
    else{
        sleep(3);
        printf("From child.\n");
        int w=wait(NULL);
        printf("wait: %d from child.\n", w);
    }
    return 0;
}
```

The output is:

```
From parent.
From child.
wait: -1 from child.
wait: 29720 from parent.
```

If we use **wait()** in child process then **wait()** returns -1. Because, there is no child of child. So, there is no wait for any process (child process) to exit .

Reference:

https://juyou.blog.csdn.net/article/details/70741655?utm_medium=distribute.pc_relevant.none-task-blog-searchFromBaidu-8.control&dist_request_id=1328741.26959.16169137868803603&depth_1-utm_source=distribute.pc_relevant.none-task-blog-searchFromBaidu-8.control