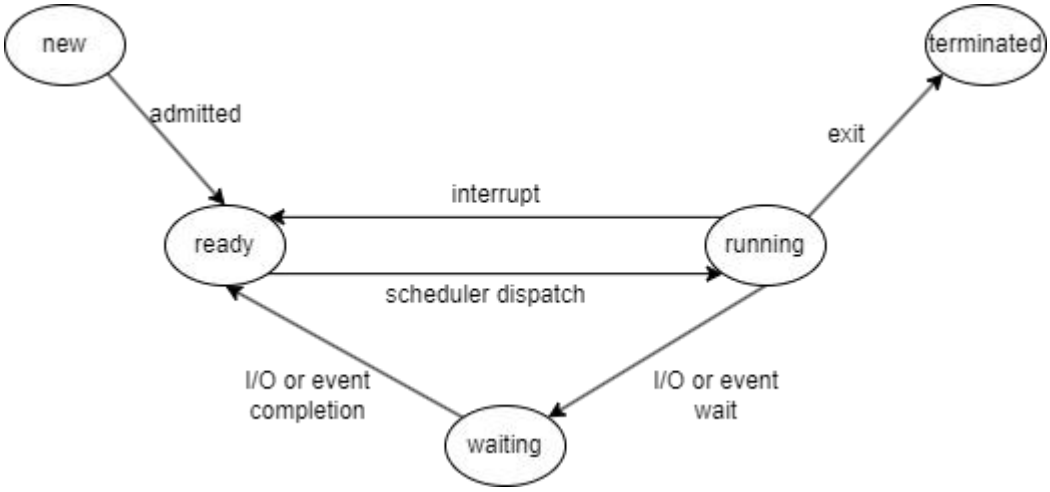
第二次作业

1、画出进程的 5 状态转换图，并说明转换原因。



进程的 5 状态：创建状态（new），就绪状态（ready），等待状态（waiting），运行状态（running），终止状态（terminated）。

进程转换原因：

创建状态到就绪状态：某一新进程被创建时，它会从创建状态进入就绪状态，等待系统调度与分配资源；

就绪状态到运行状态：CPU 被分配给某一进程时，该进程从就绪状态转换成运行状态；

运行状态到就绪状态：CPU 完成某一进程的执行后，该进程回到就绪状态等待下一次执行；

运行状态到等待状态：若一进程需要等待某些事件发生，则该进程进入等待状态；

等待状态到就绪状态：事件发生且资源可用时，进程从等待状态进入就绪状态；

运行状态到终止状态：进程执行完成或被强制结束时，进入终止状态。

2、Describe the differences among short-term, medium-term, and long-term scheduling.

从时间尺度上看，短程调度工作周期非常短，通常在毫秒或微秒级别，中程调度通常在秒到分钟的量级，长程调度的时间尺度最长，通常在分钟到小时甚至更长时间。

从任务上看，短程调度负责从就绪队列中选择一个进程来执行，并在进程时间片到期、进程等待资源或被更高优先级进程抢占时进行进程的切换；中程调度负责将进程在内存和外存之间移动；长程调度责从作业队列中选择作业并将其转换为进程，分配必要的资源，并决定何时将作业调入内存。

从目标上看，短程调度的主要目标是提高 CPU 的利用率，减少进程切换的开销，并保证系统的响应时间；中程调度的目标是优化内存使用，将不活跃的进程换出，使得内存中保持的是最活跃的进程集合；长程调度的目标是控制系统中进程的总数，确保系统不会因为过多的进程而超载，同时保证作业的吞吐量。

3、 Describe the actions taken by a kernel to context-switch between processes.

当内核进行进程上下文切换时，首先会保存当前进程的上下文，更新进程的控制块（PCB），再根据算法决定下一个运行的进程并加载下一个进程的上下文，接着更新内存管理单元（MMU），后将 CPU 从内核模式切换到用户模式，允许下一个进程开始执行其用户空间代码，从上次被中断的地方继续，最后恢复执行。

4、采用下述程序，确定 A、B、C、D 四行中 pid 和 pid1 的值。（假设父进程和子进程的 pid 分别为 2600 和 2603）

```
#include
#include
#include

int main()
{
   pid_t pid,pid1;

   pid=fork();

   if (pid<0)
   {
      fprintf(stderr,"fork fail");
      return 1;
   }
   else if (pid==0)
   {
         pid1=getpid();
         printf("child:pid=%d",pid);          //A
         printf("child:pid1=%d",pid1);        //B
    }
    else
    {
         pid1=getpid();
         printf("parent:pid=%d",pid);         //C
         printf("parent:pid1=%d",pid1);       //D
```

```
        wait(NULL);
    }
  return 0;
}
```

解：

当 fork（）被调用时，程序有两个执行路径：

子进程中，pid 是 fork()的返回值，值为 0，pid1 的值是 getpid（）的返回值，getpid()返回当前进程的 pid，即子进程的 pid，则 pid1 的值为 2603；

父进程中，pid 的值为子进程的 pid 的值，所以 pid 是 2603，pid1 的值是 getpid()的返回值，getpid()返回当前进程的 pid，即父进程的 pid，所以 pid1 是 2600；

综上：child:pid=0　　child:pid1=2603　　parent:pid=2603　　parent:pid1=2600

5、使用以下程序，请解释一下行 X 和 Y 的输出是什么。

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

解：

若程序运行到行 X 处，、pid=fork()的返回值为 0，程序在子进程中，每个

元素的值为初始值乘以其索引的负数，故 X 行输出为：

CHILD:0 CHILD:-1 CHILD:-4 CHILD:-9 CHILD:-16

若程序运行到行 Y 处，、pid=fork()的返回值大于 0，程序在父进程中。父进程将等待子进程运行结束，但子进程修改全局变量并不影响父进程中变量的值，nums 数组保持初始状态不变，故 Y 行输出为：

PARENT:0 PARENT:1 PARENT:2 PARENT:3 PARENT:4