操作系统作业 3

1. What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

解：进程间通信的模式可以分为**共享内存**和**消息传递**两大类。

· **共享内存**：采用共享内存的进程间通信，需要通信进程建立共享内存区域。通常一片共享内存区域驻留在创建共享内存段的进程地址空间内，其他需要使用这段共享内存通信的进程将其附加到自己的地址空间。

因为所有进程共享同一块内存，共享内存在各种进程间通信方式中具有最高的效率。访问共享内存区域和访问进程独有的内存区域一样快，并不需要通过系统调用或者其它需要切入内核的过程来完成。同时它也避免了对数据的各种不必要的复制。

但是由于系统内核没有对访问共享内存进行同步，必须提供自己的同步措施。所有程序之间必须达成并遵守一定的协议，以防止诸如在读取信息之前覆写内存空间等竞争状态的出现。

· **消息传递**：消息传递提供一种机制(可以是操作系统提供)，以便允许进程不必通过共享地址空间来实现通信和同步，在分布式环境中尤为有用。线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信。在一般情况下，消息传递通常需要需要大量的系统开销。正因为这些系统开销，通常信息的传递无法频繁的进行，还是需要通过共享内存的方式来实现处理器之间的批量数据传送。

2. What are the benefits of multi-threading? Which of the following components of program state are shared across threads in a multithreaded process?

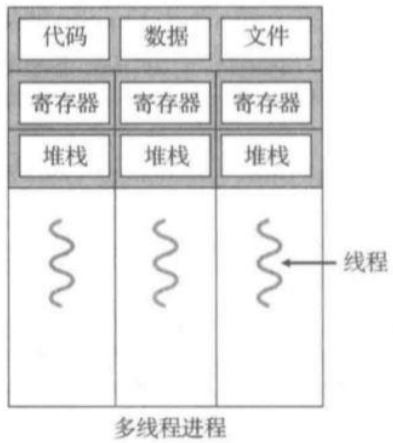
a. Register values

b. Heap memory { 全局堆 / 局部堆

c. Global variables

d. Stack memory

· 全局堆和全局变量允许在多线程之间共享。



多线程能让多个执行部分可以同时执行，具有以下优点：

即使部分阻塞或执行冗长操作，使用多线程的进程仍可以继续执行，从而增加对用户的响应程度。使用线程可以把占据时间长的程序中的任务放到后台去处理（**响应性**）；用户界面更加吸引人,这样比如用户点击了一个按钮去触发某件事件的处理,可以弹出一个进度条来显示处理的进度；程序的运行效率可能会提高；在一些等待的任务实现上如用户输入,文件读取和网络收发数据等,线程就比较有用了.

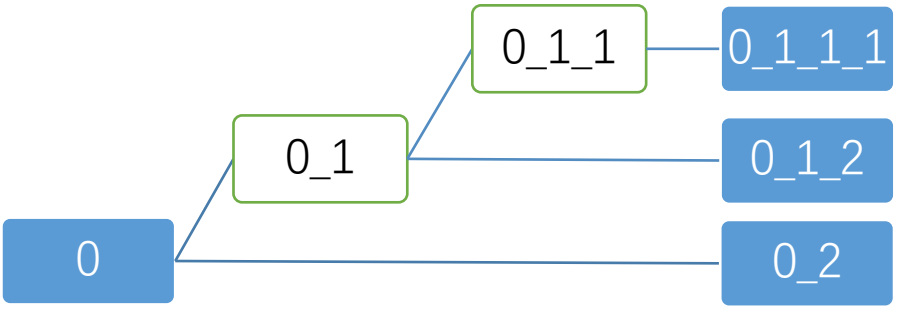
多线程技术用于编程中也有一些好处：多线程编程能实现资源共享，同一进程的多个线程共享它们所属进程的内存和资源（**资源共享**）。多线程编程更经济，对于单个程序需要执行多个类似任务的情形，线程的创建和切换所需要的开销都远小于进程（**经济**）。多线程的应用程序可以在多处理核上并行执行多个任务，单线程则只能在一个CPU上运行（**可伸缩性**）。

3. Consider the following code segment:

```
pid t pid;
pid = fork();
if (pid == 0) { /* child process */
    fork();
    thread create( . . .);
}
fork();
```

a. How many unique processes are created?

记初始的进程为0号进程。该进程运行完全部代码，不进入if段，共产生两个子进程 0_1 和 0_2，其中0_2已经没有代码可执行，创建后就运行结束；0_1进程进入if段，创建子进程0_1_1和一个线程。之后各进程运行至结尾，调用$fork()$;创建新进程结束。全过程如下图所示，加绿框的进程创建了新的线程。全过程共创建六个不同的进程。



b. How many unique threads are created?

共创建八个不同的线程（算上每个进程本身的一个线程）。

4. The program shown in the following figure uses Pthreads. What would be the output from the program at `LINE C` and `LINE P`?

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

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Figure: C program for Question 4.

输出结果:

CHILD: value = 5

PARENT: value = 0

分析:

运行至pid = fork();时父进程创建一个子进程。子进程执行if语句后创建一个新线程，这个线程令value的值为5。而value是全局变量，子进程产生的线程之间是共享的，因此在子进程中value=5.

父进程执行else if语句，并且会等待子进程运行结束后执行LINE P。因为子进程和父进程独立运行，全局变量都有各自的一份，父进程的value值不受子进程的影响。

5. What are the differences between ordinary pipe and named pipe?

·**普通管道**：普通管道允许两个进程按标准的生产者-消费者方式进行通信：生产者向管道的一端（写入端）写，消费者从管道另一端（读出端）读。因此，普通管道是单向的，只允许**单向通信**。如果需要双向通信，就要采用两个管道。普通管道**只能由创建进程所访问。**通常情况下，父进程创建一个管道，并使用它来与其子进程通信。管道是一种特殊类型的文件，因此子进程继承了父进程的管道。对于UNIX和Windows系统而言，采用普通管道通信需要有父子关系，这意味着这些管道**只可用于同一机器的进程间通信**。一旦进程已完成通信且终止了，普通管道也就不再存在了。

·**命名管道**：命名管道提供了一个更强大的通信工具。通信可以是双向的，并且父子关系不是必需的。当建立了一个命名管道之后，多个进程都可用它通信。在典型场景中，一个命名管道有几个写者；当进程通信完成后，命名管道继续存在。在常用的操作系统中，Windows系统的命名管道通信机制更加丰富，允许全双工通信且通信进程可以位于同一机器或不同机器，且同时允许字节流和消息流的数据。

6. What is race condition? Which property can guarantee that race condition will not happen?

· **竞争条件：**指多个线程或者进程在读写一个**共享数据**时结果依赖于它们执行的相对时间的情形。竞争条件发生在当多个进程或者线程在读写数据时，其最终的的结果依赖于多个进程的指令执行顺序。

为避免这种情况，应该为公共变量设立临界区，每个进程添加临界区代码，只有当某个进程处于临界区内才有权更改公共变量、更新表等。此外，还需要保证任意时刻最多只有一个进程在临界区内。

7. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process Pi (i == 0 or 1) is shown in the following Figure; the other process is Pj (j == 1 or 0). **Prove** that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```

Figure: The structure of process Pi for Question 7.

为证明一个算法是临界区问题的解决方案，需要满足以下三点：

- **互斥**：同一时刻临界区内至多只有一个进程在执行。
- **进步**：若没有进程在临界区内，并且有进程需要进入临界区，那么只有那些不在剩余区内的进程可以参加选择，以确定下次谁进入临界区。而这种选择不可以无限的推迟。
- **有限等待**：从一个进程做出进入临界区的请求直到这个请求允许为止，其他进程允许进入其临界区的次数具有上限。

首先说明该算法满足互斥条件：

flag和turn相互配合使用，能满足互斥条件。flag的含义是：若flag[x]=true，则表示进程Px希望进入临界区；turn的含义是，若turn=x，则表示此时仅允许进程Px进入临界区。

程序倒数第三行保证了进程不在临界区内时flag值为0.如果进程Pi和Pj不同时发起flag请求，那么先发起请求的进程会进入临界区执行（while不执行），另一个进程若在此过程中发起flag请求，则不可以进入直到先进入临界区的进程执行完剩余区代码，把本进程的flag置0且将turn转交给后进进程，完成临界区的交接； 而若flag[i],flag[j]同时置1（指先发起请求的进程运行至while语句及之前的这段时间内，另一进程发起进入请求），则根据此时turn的值决定谁先进入临界区。因为turn只会有一个确定的值，总是只有一个进程进入临界区。综上的各种情况，这个算法满足互斥条件。

再说明该算法满足进步条件：

当临界区内没有进程且两个进程有若干个进入临界区的需要时，如果只有一个进程发起请求，则直接进入临界区；而若flag[i],flag[j]同时置1（指先发起请求的进程运行至while语句及之前的这段时间内，另一进程发起进入请求），它们会根据turn的值选择一个进程进入临界区，而不会两个进程都等待另一个进程进入（死锁），满足进步条件；

最后说明该算法满足有限等待：

当某个进程发出希望进入临界区的请求以后，只有两种可能：直接进入临界区执行（此时另一个进程不在临界区内）；或者等待此时临界区内的进程运行完成，而剩余区的代码保证了临界区内的进程执行完成以后，临界区使用权立即交付给等待进程。因此某进程发出进入临界区请求以后，最多等待一个进程在临界区内执行完毕，满足有限等待条件。

综上，这个算法是一个临界区问题的解决方案。

8. Can strict alternation and Peterson's solution sastify all the requirements as a solution of the critical-section problem? Please explain why.

严格轮转没有全部满足要求，对**进步**的要求不满足：比如当一个进程0进入临界区后，会将turn置为1，当进程0还需要进入临界区时，必须等待进程1进入一次临界区之后才可以进入。如果进程1一直不进入临界区，那么进程0会一直阻塞在进入区。

Peterson方法满足三个要求，但依然存在问题。Peterson的办法对进程的优先级欠缺考虑。当一个低优先权的进程正在临界区运行时，往往不允许其他进程访问临界区。而如果这时有高优先权的进程需要访问临界区，这个进程就会被阻塞直到低优先权的进程结束临界区。

9. What is semaphore? How to use semaphore to implement section entry and section exit (no busy waiting)? Please give the code.

·**信号量**：一个信号量S是一个整型变量，它除了初始化外只能通过两个标准原子操作：*wait*()和*signal*()来访问（V操作，P操作）。当一个进程修改信号量的值时，没有其他进程能够同时修改同一信号量的值；且对S的操作也不能被中断。

为了克服忙等待需要，可以修改信号量操作的定义：当一个进程执行操作*wait*()且发现信号量不为正时，该进程不是忙等待而是阻塞自己。阻塞操作将一个进程放到与信号量相关的等待队列中，并将该进程的状态切换成等待状态；当其他进程执行*signal*()后，通过*wakeup*()操作使该进程重新执行。

代码如下所示：

```
//S定义
 typedef struct{
    int value;
    struct process *list;
 }semaphore;

 //wait
 wait(semaphore *S){
     S->value--;
      if(S->value<=0){
          add this process to S->list;
          block();//挂起调用它的进程
      }
 }

 //signal
 signal(semaphore *S){
     S->value++;
     if(S->value<=0){
          remove a process P from S->list;
          wakeup(P);
      }
  }
```

10. What is deadlock? List the four requirements of deadlock.

死锁：在多道程序环境中，多个进程可以竞争数量有限的资源。当一个进程申请资源时如果此时没有可用资源，那么这个进程进入等待状态；如果所申请的资源被其他**等待进程**占有，那么这个等待进程可能再也无法改变状态。

引起死锁的四个必要条件：

· **互斥**：至少有一个资源必须处于非共享模式，即一次只有一个进程可以使用。如果另一进程申请该资源，必须等到该资源被释放。

· **占有并等待**：一个进程占有至少一个资源，并等待另一个资源，而该资源被其他的进程所占有。

· **非抢占**：资源不能被抢占，只能在被进程完成任务后自愿释放。

· **循环等待**：有一组等待进程$\{P_0, P_1, \cdots\cdots P_n\}$,后一个进程所需要的资源被前者占有，均处于等待状态。