```
// 3.1.a

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main() {
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 semid=semget(IPC_PRIVATE, 1, 00600);
 // semaphore 0 : to block the Parent
 arg.val=0; // arg.val=1;
 semctl(semid, 0, SETVAL, arg);
 sop[0].sem_flg=0;

 if (fork() == 0) {
   printf("CHILD PID: %d\n", getpid());
   sop[0].sem_num=0; sop[0].sem_op=+1; // sop[0].sem_op=-1;
   semop(semid, sop, 1);
   exit(0);
 }
 sop[0].sem_num=0; sop[0].sem_op=-1; // sop[0].sem_op=0;
 semop(semid, sop, 1);
 printf("PARENT PID: %d\n", getpid());
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.1.b

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main() {
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 semid=semget(IPC_PRIVATE, 1, 00600);
 // semaphore 0 : to block the Child
 arg.val=0; // arg.val=1;
 semctl(semid, 0, SETVAL, arg);
 sop[0].sem_flg=0;

 if (fork() == 0) {
   sop[0].sem_num=0; sop[0].sem_op=-1; // sop[0].sem_op=0;
   semop(semid, sop, 1);
   printf("CHILD PID: %d\n", getpid());
   semctl(semid, 0, IPC_RMID, NULL);
   exit(0);
 }
 printf("PARENT PID: %d\n", getpid());
 sop[0].sem_num=0; sop[0].sem_op=+1; // sop[0].sem_op=-1;
 semop(semid, sop, 1);
 return(0);
}
```

```
// 3.2.a: prone to malfunction (why?)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

void count(int min, int max)
{
 int i;
 for (i=min; i<=max; i++)
     printf("%d : %d\n", getpid(), i);
}

int main()
{
 int semid;
 struct sembuf sop[1];
 union semun { int val; } arg;

 semid=semget(IPC_PRIVATE, 1, 00600);
 arg.val=0;
 semctl(semid, 0, SETVAL, arg);
 sop[0].sem_flg=0;

 if (fork()==0) { // Child
     sleep(1); // execute with(out) sleep
     sop[0].sem_num=0;sop[0].sem_op=-1; semop(semid,sop,1);
     count(4,6);
     sop[0].sem_num=0;sop[0].sem_op=+1; semop(semid,sop,1);

     sop[0].sem_num=0;sop[0].sem_op=-1; semop(semid,sop,1);
     count(10,12);
     sop[0].sem_num=0;sop[0].sem_op=+1; semop(semid,sop,1);

     exit(0);
 }

 count(1,3);
 sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);

 sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
 count(7,9);
 sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);

 sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
 count(13,15);

 semctl(semid, 0, IPC_RMID, NULL); // execute with(out) semctl
 return(0);
}
```

```
// 3.2.b ; correct implementation (uses two semaphores: one
           to block the Parent, another to block the Child)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

void count(int min, int max)
{
 int i;
 for (i=min; i<=max; i++)
     printf("%d : %d\n", getpid(), i);
}

int main()
{
 int semid;
 struct sembuf sop[1];
 union semun { int val; } arg;

 semid=semget(IPC_PRIVATE, 2, 00600);
 // semaphore 0 : to block the Parent
 // semaphore 1 : to block the Child
 arg.val=0;
 semctl(semid, 0, SETVAL, arg);
 semctl(semid, 1, SETVAL, arg);
 sop[0].sem_flg=0;

if (fork()==0) { // Child
     sleep(1);
     sop[0].sem_num=1;sop[0].sem_op=-1; semop(semid,sop,1);
     count(4,6);
     sop[0].sem_num=0;sop[0].sem_op=+1; semop(semid,sop,1);

     sop[0].sem_num=1;sop[0].sem_op=-1; semop(semid,sop,1);
     count(10,12);
     sop[0].sem_num=0;sop[0].sem_op=+1; semop(semid,sop,1);

     exit(0);
 }

 count(1,3);
 sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);

 sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
 count(7,9);
 sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);

 sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
 count(13,15);

 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.3

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/types.h>

int main() {
 int shmid, *shmaddr;
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 shmid=shmget(IPC_PRIVATE, sizeof(int), 00600);
 shmaddr=(int*)shmat(shmid, NULL, 0);

 semid=semget(IPC_PRIVATE, 2, 00600);
 // semaphore 0 : to block the Parent
 // semaphore 1 : to block the Child
 arg.val=0; // arg.val=1;
 semctl(semid, 0, SETVAL, arg);
 semctl(semid, 1, SETVAL, arg);
 sop[0].sem_flg=0;

 if (fork()==0) {
     do {
         scanf("%d", shmaddr);
         // unblock parent
         sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);
         // block child
         sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
         printf("%d\n", *shmaddr);
     } while (*shmaddr!=1);
     shmctl(shmid, IPC_RMID, 0);
     semctl(semid, 0, IPC_RMID, NULL);
     exit(0);
 }

 do {
     // block parent
     sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
     (*shmaddr)++;
     // unblock child
     sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
 } while (*shmaddr!=1);

 return(0);
}
```

```
// 3.4

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/types.h>

int main() {
 int shmid, *shmaddr;
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 shmid=shmget(IPC_PRIVATE, sizeof(int), 00600);
 shmaddr=(int*)shmat(shmid, NULL, 0);

 semid=semget(IPC_PRIVATE, 3, 00600);
 // semaphore 0 : to block the 1st Child
 // semaphore 1 : to block the 2nd Child
 // semaphore 2 : to block the Parent
 arg.val=0; // arg.val=1;
 semctl(semid, 0, SETVAL, arg);
 semctl(semid, 1, SETVAL, arg);
 semctl(semid, 2, SETVAL, arg);
 sop[0].sem_flg=0;

 if (fork()==0) { // 1st child
     do {
         scanf("%d", shmaddr);
         // unblock 2nd child
         sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
         // block 1st child
         sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
         printf("%d\n", *shmaddr);
     } while (*shmaddr!=2);
     shmctl(shmid, IPC_RMID, 0);
     semctl(semid, 0, IPC_RMID, NULL);
     exit(0);
 }

 if (fork()==0) { // 2nd child
     do {
         // block 2nd child
         sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
         (*shmaddr)++;
         // unblock parent
         sop[0].sem_num=2; sop[0].sem_op=+1; semop(semid, sop, 1);
     } while (*shmaddr!=1);
     exit(0);
 }

 do { // parent
     // block parent
     sop[0].sem_num=2; sop[0].sem_op=-1; semop(semid, sop, 1);
     (*shmaddr)++;
     // unblock 1st child
     sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);
 } while (*shmaddr!=2);

 return(0);
}
```

```
// 3.5

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#include <sys/sem.h>

int sum(int min, int max)
{
 int result=0, i;

 for (i=min; i<=max; i++) result += i;

 return(result);
}

int main()
{
 int shmid, *shmaddr, sum_parent;

 int semid;
 struct sembuf sop[1];
 union semun { int val; } arg;

 shmid=shmget(IPC_PRIVATE, sizeof(int), 00600);
 shmaddr=(int*)shmat(shmid, NULL, 0);
 shmaddr[0]=0;

 semid=semget(IPC_PRIVATE, 1, 00600);
 // semaphore 0 : to block the Parent
 arg.val=0; // arg.val=1;
 semctl(semid, 0, SETVAL, arg);
 sop[0].sem_flg=0;

 if ( fork()==0 ) { // Child
    shmaddr[0]=sum(51,100);
    sop[0].sem_num=0; sop[0].sem_op=+1; // sop[0].sem_op=-1;
    semop(semid, sop, 1);
    exit(0);
 }

 sum_parent=sum(1,50);
 sop[0].sem_num=0; sop[0].sem_op=-1; // sop[0].sem_op=0;
 semop(semid, sop, 1);
 printf("%d\n", sum_parent + shmaddr[0]);
 shmctl(shmid, IPC_RMID, 0);
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.6.i ; 10 integers in shared memory and 10 semaphores
//         (one per integer) to block the Parent;
// the semaphores are initialized to 0; before reading (writing)
// a specific integer, Parent (Children) decrements (increments)
// the protecting semaphore by 1 unit (alternative: semaphores
// set to initial value 1, Parent tests, Children decrements)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#include <sys/sem.h>

int sum(int min, int max)
{
 int result=0, i;

 for (i=min; i<=max; i++) result += i;

 return(result);
}

int main() {
 int shmid_sums, *ptr_sums;
 int i, sum_final=0;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 shmid_sums=shmget(IPC_PRIVATE, 10*sizeof(int), 00600);
 ptr_sums=(int*)shmat(shmid_sums, NULL, 0);

 semid=semget(IPC_PRIVATE, 10, 00600);
 arg.val=0; // arg.val=1;
 for (i=0; i<10; i++) semctl(semid, i, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++)
     if (fork()==0) { // Child
         ptr_sums[i]=sum(i*10+1, i*10+10);
         sop[0].sem_num=i; sop[0].sem_op=+1; //sop[0].sem_op=-1;
         semop(semid, sop, 1);
         exit(0);
     }

 for (i=0; i<10; i++) { // blocking waiting
     sop[0].sem_num=i; sop[0].sem_op=-1; //sop[0].sem_op=0;
     semop(semid, sop, 1);
     sum_final = sum_final + ptr_sums[i];
 }

 printf("%i\n", sum_final);
 shmctl(shmid_sums, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.6.ii ; 10 integers in shared memory and 1 semaphore to
//          block the Parent;
// the semaphore is initialized to 10; Parent tests the
// semaphore before reading the integers; each Child
// decrements the semaphore after writing in its specific
// integer (alternative: semaphore initially set to 0,
// Parent decrements 10 units, each Child increments 1 unit)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#include <sys/sem.h>

int sum(int min, int max)
{
 int result=0, i;

 for (i=min; i<=max; i++) result += i;

 return(result);
}

int main()
{
 int shmid_sums, *ptr_sums;
 int i, sum_final=0;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 shmid_sums=shmget(IPC_PRIVATE, 10*sizeof(int), 00600);
 ptr_sums=(int*)shmat(shmid_sums, NULL, 0);

 semid=semget(IPC_PRIVATE, 1, 00600);
 arg.val=10; // arg.val=0;
 semctl(semid, 0, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++)
     if (fork()==0) { // Child
         ptr_sums[i]=sum(i*10+1, i*10+10);
         sop[0].sem_num=0; sop[0].sem_op=-1; // sop[0].sem_op=+1;
         semop(semid, sop, 1);
         exit(0);
     }

 sop[0].sem_num=0; sop[0].sem_op=0; semop(semid, sop, 1);
 // sop[0].sem_num=0; sop[0].sem_op=-1;
 // for (i=0; i<10; i++) semop(semid, sop, 1);
 for (i=0; i<10; i++) {
      sum_final = sum_final + ptr_sums[i];
 }

 printf("%i\n", sum_final);
 shmctl(shmid_sums, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.6.iii ; uses 1 integer in shared memory and 1 semaphore
// to block the Parent; the semaphore is initialized to 10;
// Parent tests the semaphore before reading the integer
// and each Child decrements the semaphore after updating the
// integer; this approach is INCORRECT (see // RACE-CONDITION)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#include <sys/sem.h>

int sum(int min, int max)
{
 int result=0, i;

 for (i=min; i<=max; i++) result += i;

 return(result);
}

int main()
{
 int i, shmid_sum, *ptr_sum;
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 shmid_sum=shmget(IPC_PRIVATE, sizeof(int), 00600);
 ptr_sum=(int*)shmat(shmid_sum, NULL, 0);
 *ptr_sum=0;

 semid=semget(IPC_PRIVATE, 1, 00600);
 arg.val=10; semctl(semid, 0, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++)
     if (fork()==0) { // Child
         (*ptr_sum)+=sum( i*10+1 , i*10+10 ); // RACE-CONDITION
         sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
         exit(0);
     }

 sop[0].sem_num=0; sop[0].sem_op=0; semop(semid, sop, 1);

 printf("%i\n", *ptr_sum);
 shmctl(shmid_sum, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.6.iv ; uses 1 integer in shared memory and 2 semaphores;
// semaphore 0 is initialized to 10 and blocks the Parent;
// semaphore 1 is initialized to 1 and ensures mutual exclusion
// of Children when accessing the shared integer; this approach
// avoids the RACE-CONDITION of the previous variant

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#include <sys/sem.h>

int sum(int min, int max)
{
 int result=0, i;

 for (i=min; i<=max; i++) result += i;

 return(result);
}

int main()
{
 int i, shmid_sum, *ptr_sum;
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 shmid_sum=shmget(IPC_PRIVATE, sizeof(int), 00600);
 ptr_sum=(int*)shmat(shmid_sum, NULL, 0);
 *ptr_sum=0;

 semid=semget(IPC_PRIVATE, 2, 00600);
 arg.val=10; semctl(semid, 0, SETVAL, arg);
 arg.val=1; semctl(semid, 1, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++)
     if (fork()==0) { // Child
         sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
         (*ptr_sum) += sum( i*10+1 , i*10+10 );
         sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
         sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
         exit(0);
     }

 sop[0].sem_num=0; sop[0].sem_op=0; semop(semid, sop, 1);

 printf("%i\n", *ptr_sum);
 shmctl(shmid_sum, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.6.v ; uses 2 integers in shared memory and 1 semaphore
// to ensure mutual exclusion when updating both integers;
// the 1st integer accumulates the partial sums of the Children;
// the 2nd integer counts the Children that finished; this
// counter is watched by the Parent and when it reaches 10, the
// Parent collects the overall sum from the 1st integer; the
// Parent doesn't need a semaphore when reading the counter (it
// doesn't change the counter and this will eventually reach 10)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#include <sys/sem.h>

int sum(int min, int max)
{
 int result=0, i;

 for (i=min; i<=max; i++) result += i;

 return(result);
}

int main()
{
 int i, shmid_nums, *ptr_nums;
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 shmid_nums=shmget(IPC_PRIVATE, 2*sizeof(int), 00600);
 ptr_nums=(int*)shmat(shmid_nums, NULL, 0);
 ptr_nums[0]=ptr_nums[1]=0;

 semid=semget(IPC_PRIVATE, 1, 00600);
 arg.val=1; semctl(semid, 0, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++)
     if (fork()==0) { // Child
         sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
         ptr_nums[0] += sum( i*10+1 , i*10+10 );
         ptr_nums[1] ++;
         sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);
         exit(0);
     }

 while (ptr_nums[1] != 10); // MUTUAL EXCLUSION UNNECESSARY !...

 printf("%i\n", ptr_nums[0]);
 shmctl(shmid_nums, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.7
// - 11 semaphores, DECREMENTS for blocking
// - semaphore 0 initialized to 0: blocks the Parent
// - semaphore 1 to 10 initialized to 0: blocks Children
// (alternative: 11 semaphores, TESTS for blocking initially;
//  semaphore 0 initialized to 1 blocks the Parent;
//  semaphores 1 to 10 initialized to 1 blocks the Children)

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/types.h>

int main()
{
 int i, shmid, *shmaddr;
 int semid; struct sembuf sop[1];
 union semun { int val; } arg;

 shmid=shmget(IPC_PRIVATE, sizeof(int), 00600);
 shmaddr=(int*)shmat(shmid, NULL, 0);
 *shmaddr=0;

 semid=semget(IPC_PRIVATE, 11, 00600);
 arg.val=0; // arg.val=1;
 for (i=0; i<11; i++)
     semctl(semid, i, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=1; i<11; i++)
     if (fork()==0) { // Child
         sop[0].sem_num=i; sop[0].sem_op=-1; //sop[0].sem_op=0;
         semop(semid, sop, 1);
         // printf("Child %d\n", i-1);
        (*shmaddr) = (*shmaddr) + (i*i);
        sop[0].sem_num=(i+1)%11; sop[0].sem_op=+1; //sop[0].sem_op=-1;
         semop(semid, sop, 1);
        exit(0);
     }

 sop[0].sem_num=1; sop[0].sem_op=+1; //sop[0].sem_op=-1;
 semop(semid, sop, 1);
 sop[0].sem_num=0; sop[0].sem_op=-1; //sop[0].sem_op=0;
 semop(semid, sop, 1);
 printf("%d\n", *shmaddr);
 shmctl(shmid, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.8.a

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

int main() {
 int N, *A, i, ret, shmid_A, shmid_odds, *odds;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 scanf("%d", &N);

 shmid_A=shmget(IPC_PRIVATE, N*sizeof(int), 00600);
 A=(int*)shmat(shmid_A, NULL, 0);

 shmid_odds=shmget(IPC_PRIVATE, sizeof(int), 00600);
 odds=(int*)shmat(shmid_odds, NULL, 0);
 *odds=0;

 semid=semget(IPC_PRIVATE, 2, 00600);
 // semaphore 0 : to block the Parent
 // semaphore 1 : to protect the counter
 arg.val=N; semctl(semid, 0, SETVAL, arg);
 arg.val=1; semctl(semid, 1, SETVAL, arg);
 sop[0].sem_flg=0;

 srandom(getpid());
 for (i=0; i<N; i++)
   A[i] = random();

 for (i=0; i<N; i++) {
   ret=fork();
     if (ret == 0) {
       if (A[i] % 2 != 0) {
         printf("A[%d]: %d : odd\n", i, A[i]);
         sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
         (*odds)++;
         sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
       }
       sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
       exit(0);
    }
 }

 sop[0].sem_num=0; sop[0].sem_op=0; semop(semid, sop, 1);
 printf("odds: %d\n", *odds);
 shmctl(shmid_A, IPC_RMID, NULL);
 shmctl(shmid_odds, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);

 return(0);
}
```

```
// 3.8.b.i

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

int main() {
 int N, *A, i, ret, shmid_A, shmid_odds, *odds;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 scanf("%d", &N);

 shmid_A=shmget(IPC_PRIVATE, N*sizeof(int), 00600);
 A=(int*)shmat(shmid_A, NULL, 0);

 shmid_odds=shmget(IPC_PRIVATE, sizeof(int), 00600);
 odds=(int*)shmat(shmid_odds, NULL, 0);
 *odds=0;

 semid=semget(IPC_PRIVATE, 3, 00600);
 // semaphore 0 : to block the Parent
 // semaphore 1 : to protect the counter
 // semaphore 2 : to block the Children
 arg.val=N; semctl(semid, 0, SETVAL, arg);
 arg.val=1; semctl(semid, 1, SETVAL, arg);
 arg.val=0; semctl(semid, 2, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<N; i++) {
   ret=fork();
   if (ret == 0) {
     sop[0].sem_num=2; sop[0].sem_op=-1; semop(semid, sop, 1);
       if (A[i] % 2 != 0) {
         printf("A[%d]: %d : odd\n", i, A[i]);
         sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
         (*odds)++;
         sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
       }
     sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
     exit(0);
   }
 }

 srandom(getpid());
 for (i=0; i<N; i++)
   A[i] = random();

 for (i=0; i<N; i++) {
   sop[0].sem_num=2; sop[0].sem_op=+1; semop(semid, sop, 1);
 }

 sop[0].sem_num=0; sop[0].sem_op=0; semop(semid, sop, 1);
 printf("odds: %d\n", *odds);
 shmctl(shmid_A, IPC_RMID, NULL);
 shmctl(shmid_odds, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);

 return(0);
}
```

```
// 3.8.b.ii

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

int main() {
 int N, *A, i, ret, shmid_A, shmid_odds, *odds;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 scanf("%d", &N);

 shmid_A=shmget(IPC_PRIVATE, N*sizeof(int), 00600);
 A=(int*)shmat(shmid_A, NULL, 0);

 shmid_odds=shmget(IPC_PRIVATE, sizeof(int), 00600);
 odds=(int*)shmat(shmid_odds, NULL, 0);
 *odds=0;

 semid=semget(IPC_PRIVATE, 2, 00600);
 // semaphore 0 : to block the Children and the Parent
 // semaphore 1 : to protect the counter
 arg.val=N; semctl(semid, 0, SETVAL, arg);
 arg.val=1; semctl(semid, 1, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<N; i++) {
      ret=fork();
      if (ret == 0) {
            sop[0].sem_num=0; sop[0].sem_op=0; semop(semid, sop, 1);
            if (A[i] % 2 != 0) {
                  printf("A[%d]: %d : odd\n", i, A[i]);
                  sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
                  (*odds)++;
                  sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
            }
            sop[0].sem_num=0; sop[0].sem_op=1; semop(semid, sop, 1);
            exit(0);
      }
 }

 srandom(getpid());
 for (i=0; i<N; i++)
      A[i] = random();

 for (i=0; i<N; i++) {
      sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
 }

 for (i=0; i<N; i++) {
      sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
 }
 printf("odds: %d\n", *odds);
 shmctl(shmid_A, IPC_RMID, NULL);
 shmctl(shmid_odds, IPC_RMID, NULL);
 semctl(semid, 0, IPC_RMID, NULL);

 return(0);
}
```

```
// 3.9.i
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main()
{
 int pids[10], i;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 semid=semget(IPC_PRIVATE, 11, 00600);
 // semaphore 0..9 : to block the Children
 // semaphore 10 : to block the Parent
 arg.val=0; for (i=0; i<11; i++) semctl(semid, i, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++) {
      pids[i]=fork();
      if (pids[i] == 0) {
           // await for parent
           sop[0].sem_num=i; sop[0].sem_op=-1; semop(semid, sop, 1);
           printf("%d\n", getpid());
           // unblock parent
           sop[0].sem_num=10; sop[0].sem_op=+1; semop(semid, sop, 1);
           exit(0);
      }
 }

 for (i=0; i<10; i++) { // unblock children with even PID
      if (pids[i] % 2 == 0) {
           sop[0].sem_num=i; sop[0].sem_op=+1; semop(semid, sop, 1);
      }
 }

 for (i=0; i<10; i++) { // await for children with even PID
      if (pids[i] % 2 == 0) {
           sop[0].sem_num=10; sop[0].sem_op=-1; semop(semid, sop, 1);
      }
 }

 for (i=0; i<10; i++) { // unblock children with odd PID
      if (pids[i] % 2 != 0) {
           sop[0].sem_num=i; sop[0].sem_op=+1; semop(semid, sop, 1);
      }
 }

 for (i=0; i<10; i++) { // await for children with odd PID
      if (pids[i] % 2 != 0) {
           sop[0].sem_num=10; sop[0].sem_op=-1; semop(semid, sop, 1);
      }
 }

 printf("parent: done\n");
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.9.ii
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main()
{
 int pid, num_pids_even=0, num_pids_odd=0, i;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 semid=semget(IPC_PRIVATE, 3, 00600);
 // semaphore 0 : to block even Children
 // semaphore 1 : to block odd Children
 // semaphore 2: to block the Parent
 arg.val=0; for (i=0; i<3; i++) semctl(semid, i, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++) {
      pid=fork();
      if (pid == 0) {
            // await for parent
            if (getpid() % 2 == 0) {
                  sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
            }
            else {
                  sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
            }
            printf("%d\n", getpid());
            // unblock parent
            sop[0].sem_num=2; sop[0].sem_op=+1; semop(semid, sop, 1);
            exit(0);
      }
      if (pid % 2 == 0) num_pids_even ++; else num_pids_odd ++;
 }

 for (i=0; i<num_pids_even; i++) { // unblock children with even PID
            sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);
 }

 for (i=0; i<num_pids_even; i++) { // await for children with even PID
            sop[0].sem_num=2; sop[0].sem_op=-1; semop(semid, sop, 1);
 }

 for (i=0; i<num_pids_odd; i++) { // unblock children with odd PID
            sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
 }

 for (i=0; i<num_pids_odd; i++) { // await for children with odd PID
            sop[0].sem_num=2; sop[0].sem_op=-1; semop(semid, sop, 1);
 }

 printf("parent: done\n");
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```

```
// 3.9.iii

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main()
{
 int pid, num_pids_even=0, num_pids_odd=0, i;
 int semid; struct sembuf sop[1]; union semun { int val; } arg;

 semid=semget(IPC_PRIVATE, 3, 00600);
 // semaphore 0 : to block even Children
 // semaphore 1 : to block odd Children
 // semaphore 2: to block the Parent
 arg.val=0; for (i=0; i<2; i++) semctl(semid, i, SETVAL, arg);
 sop[0].sem_flg=0;

 for (i=0; i<10; i++) {
      pid=fork();
      if (pid == 0) {
            // await for parent
            if (getpid() % 2 == 0) {
                  sop[0].sem_num=0; sop[0].sem_op=-1; semop(semid, sop, 1);
            }
            else {
                  sop[0].sem_num=1; sop[0].sem_op=-1; semop(semid, sop, 1);
            }
            printf("%d\n", getpid());
            // unblock parent
            sop[0].sem_num=2; sop[0].sem_op=-1; semop(semid, sop, 1);
            exit(0);
      }
      if (pid % 2 == 0) num_pids_even ++; else num_pids_odd ++;
 }

 // set parent semaphore to the number of children with even PID
 arg.val=num_pids_even; semctl(semid, 2, SETVAL, arg);
 for (i=0; i<num_pids_even; i++) { // unblock children with even PID
            sop[0].sem_num=0; sop[0].sem_op=+1; semop(semid, sop, 1);
 }
 // await for children with even PID
 sop[0].sem_num=2; sop[0].sem_op=0; semop(semid, sop, 1);

 // set parent semaphore to the number of children with odd PID
 arg.val=num_pids_odd; semctl(semid, 2, SETVAL, arg);
 for (i=0; i<num_pids_odd; i++) { // unblock children with odd PID
            sop[0].sem_num=1; sop[0].sem_op=+1; semop(semid, sop, 1);
 }
 // await for children with even PID
 sop[0].sem_num=2; sop[0].sem_op=0; semop(semid, sop, 1);

 printf("parent: done\n");
 semctl(semid, 0, IPC_RMID, NULL);
 return(0);
}
```