**Process Synchronization**

1. a： B
2. b： C

If there is no special method, let the wife, husband free call deposit, withdraw function, in the execution of the two functions, the program is divided into three steps to assign value change deposit, if the wife and husband also called a function before the end of another function, the result will only retain the result who last call the person, which will cause inconsistent data.

Solution: Set the signal, the wife and husband first wait () before executing the function, if no one is performing and then go below, after the end of the withdrawal or deposit, signal() indicates that the use is finished, the next person can use.

3.

Count system signal mode, define the remaining connection position as N, for server, call wait () before access, wait if the remaining connection position is less than or equal to 0, and whenever the customer rolls out, the customer calls signal() releases a signal quantity.

4.

The first readers–writers problem is that the reader after reaches the author must wait, even if the author does. The second reader-writer question favors the writer to write as soon as he is ready to write.

5.

Implementation: semaphore S1=1, S2=0, S3=0;P(S2);

**process P1 {**

**while (TRUE) {**

**// read record；**

**P(S1);**

**// put record in to buffer;**

**V(S2); }**

**process P2 {**

**while (TRUE) {**

**P(S2);**

**// process record;**

**V(S3); } }**

**process P3 {**

**while (TRUE) {**

**P(S3);**

**// print processed record;**

**V(S1); } }**

6.

int readcount = 0, writecount = 0; semaphore x = 1, y = 1, wrt = 1, red = 1;

**process writer {**

**while (true) {**

**wait(y);**

**writecount++;**

**if (writecount == 1)**

**wait(red);**

**signal(y);**

**wait(wrt);**

**// writing is performed**

**signal(wrt);**

**wait(y);**

**writecount--;**

**if (writecount == 0)**

**signal(red);**

**signal(y);**

**}}**

**process reader {**

**while (true) {**

**wait(red);**

**wait(x);**

**readcount++;**

**if (readcount == 1) wait(wrt);**

**signal (x);**

**signal(red);**

**// reading is performed**

**wait (x);**

**readcount--;**

**if (readcount == 0) signal(wrt);**

**signal(x);**

**}}**

}

7.

int numsLeft=0; int numsRight=0; right=1;left=0;

**process left {**

**while (true) {**

**numsLeft++;**

**wait(right);**

**while (numsLeft>0)**

**{**

**//go through the bridge**

**numsLeft--；**

**}**

**signal (left)**

**}}**

**process right {**

**while (true) {**

**numsRight++;**

**wait(left);**

**while (numsRight>0)**

**{**

**//go through the bridge**

**numsRight--；**

**}**

**signal (right)**

**}}**

**Deadlocks**

1. The worst condition is like that:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Allocation | Max | Available |
| P0 | 1 | 2 | 1 |
| P1 | 1 | 2 |  |
| P2 | 1 | 2 |  |

This implies that each process is holding one resource and is waiting for one more.But if any of them request resource for system,system could provide enough resource for them.After that ,they will release the resource they allocate,and system will never be deadlock.

this system is deadlock-free.

2.

a circumstance: any time

c circumstance:when its need still less than resource that system have c could be made

d circumstance:any time

e circumstance:any time as long as the new process need less than the former ones

f circumstance:any time

3.

a.:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | AllocationA | | | |
|  | A | B | V | D |
| P0 | 0 | 0 | 0 | 0 |
| P1 | 0 | 7 | 5 | 0 |
| P2 | 1 | 0 | 0 | 2 |
| P3 | 0 | 0 | 2 | 0 |
| P4 | 0 | 6 | 4 | 2 |

b.:safe

c:yes it can

4.

a: safe : P2 P4 P0 P1 P3

b: safe : P1 P2 P3 P4 P0

5.

1. P0 P2 P3 P4 P1
2. P0 P2 P3 P4 P1
3. no it can not(unsafe)

**Main Memory**

1.

Internal Fragmentation is the storage space that remains unused between the allocated memory blocks, whereas External Fragmentation is the holes between the contiguous blocks that are very small to serve a request.

2.

first-fit：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | memory partitions | | | | |
| processes | 100KB | 500KB | 200KB | 300KB | 600KB |
| 212KB |  | √ |  |  |  |
| 417KB |  |  |  |  | √ |
| 112KB |  | √ |  |  |  |
| 426KB |  |  |  |  |  |

process which has 426KB need to wait until there are enough memory.

best-fit：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | memory partitions | | | | |
| processes | 100KB | 500KB | 200KB | 300KB | 600KB |
| 212KB |  |  |  | √ |  |
| 417KB |  | √ |  |  |  |
| 112KB |  |  | √ |  |  |
| 426KB |  |  |  |  | √ |

worst-fit：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | memory partitions | | | | |
| processes | 100KB | 500KB | 200KB | 300KB | 600KB |
| 212KB |  |  |  |  | √ |
| 417KB |  | √ |  |  |  |
| 112KB |  |  |  | √ |  |
| 426KB |  |  |  |  |  |

process which has 426KB need to wait until there are enough memory.

best-fit algorithm makes the most efficient use of memory

3.

a:3,13

b:41,111

c:210,161

d:634,784

e:1953,129

4.

a: 2\*200=400

b: 0.75\*(20+100)+0.25\*(20+2\*200)=145

(EAT = (1 + e)a + (2 + e)(1–a ) = 2 + e –a )

5.

1. 219+430=649
2. 2300+10=2310
3. illegal reference(500>100)
4. 1327+400=1727
5. illegal reference (112>96)