VIBGYOR ePress

International Journal of Signal Processing and Analysis

ISSN: 2631-5114

Preliminary Analysis of Synchronization and Mutual Exclusion of Process in Operating System

Xie Hao, Chen Ming, Yang Chao Shen and Xu Yun Long*

Applied Technology College of Soochow University, China

Abstract

The synchronization and mutual exclusion between operating system processes is the key point and difficulty of this course. Synchronization is the process of sending messages with each other between groups of concurrent processes in an asynchronous environment due to direct constraints, making all processes carried out at a certain speed. Mutual exclusion means that two or over two concurrent processes that share this resource cannot go into the critical zone simultaneously. There are four types on the realization mechanisms of process synchronization and mutual exclusion: Semaphore, monitor, rendezvous and distribution system. The basic principle of semaphore mechanism is that two or more processes cooperate through simple signal application and release. The process applies for signal through execution primitive P(s) and releases signal through execution primitive V(s).

Keywords

Operating system, Process, Semaphore, P, V operation

Introduction

Operating system is the most important control and management center of computer system, whose important feature is process concurrency. After introducing process concurrency, each process can execute simultaneously and move forward at independent speed, which improve resource utilization and system throughput; At the same time, promote system performance. However, they share system resources and work together, which produce complicated and mutually restricted relationship between processes and cause "chaos" to program execution. To make the concurrent execution processes share the resources and work together effectively and make program execution with reproducibility, it needs reasonable control and coordination for correct operation. Operating system provides process synchronization mechanism to solve these problems, ensuring the normal activities of all processes within the system.

In Synchronization, for example, process A needs to read the information generated by the process B from the buffer, when the buffer is empty, the process B is blocked because it cannot read information. When the process A generates information into the buffer, the process B will be awakened.

In Mutual Exclusion like process B needs to access the printer, but at this point process A occupies the printer, process B will be blocked, until the process A released the printer resources, process B can resume.

Summary of Related Theories

Process synchronization and mutual exclusion

Process synchronization realizes ordered access of visitor for resource through other mechanisms on the basis of mutual exclusion. Under most situations, synchronization has realized mutual exclusion; in particular, all written resource situations must be mutually exclusive. Minority cases refer to that several visitors can access resource at the same time.

***Corresponding author:** *Xu Yun Long, Applied Technology College of Soochow University, Suzhou Jiangsu 215325, China, E-mail: 11432123@qq.com*

Received: October 17, 2017: **Accepted:** December 13, 2017: **Published:** December 15, 2017

Citation: Hao X, Ming C, Shen YC, Long XY (2017) Preliminary Analysis of Synchronization and Mutual Exclusion of Process in Operating System. Int J Signal Process Anal 2:003

Process mutual exclusion refers to that only one visitor is allowed to access one resource. It is with uniqueness and exclusion. However, mutual exclusion can't restrict the access order of visitor for resource, which is that the access is unordered [1].

Critical zone and critical resource

Critical zone refers to the program that accesses critical resource in each process. The access of process for critical zone has to be mutually exclusive. Only one process is allowed to access critical zone each time and other processes need to wait. Critical resource refers to the resource that only allows one process access each time.

The basic principle of critical zone management is that if several processes want to access free critical zone, it only allows one process each time. At any time, the process in critical zone can't exceed one. If the critical zone already has one process, other processes that try to access critical zone have to wait. The process accessing critical zone has to exit within limited time, so that other processes can enter their critical zones timely. If the process can't access own critical zone, it should give up CPU and avoid "busy" phenomenon in process.

Realize Process Mutual Exclusion with Semaphore

How to make sure that only one process accesses resource at one moment? This is the management method of critical resource. The semaphore and P, V operation mechanism proposed by Dijkstra has been introduced after introducing some immature management plans.

Definition of semaphore

Semaphore is that process is forced to stop the execution at one special point until receive a corresponding special variable value [2]. The process uses P, V two primitive operations to send and receive signal. If the signal is not sent out, the process will be hanged until the signal is sent out.

Semaphore can be categorized into integer semaphore and recorded semaphore based on its value.

It needs to be noticed that in integer signal, wait(s) and signal(s) are two atom manipulations; therefore, they can't be interrupted during execution. In addition, in wait operation, test for s value and s-operation can't be interrupted, so this mechanism does not follow the principle of "let right to wait", it will produce "busy wait" problem, which affect the operation efficiency of the system seriously.

In recorded semaphore, it adopts "let right to wait" strategy, but there will be the situation that several processes wait the same critical resource. Therefore, it needs to use a value representing resource number, but also needs to add a linked list pointer list for the process of link jam wait.

P, V operation definition description

In semaphore structure, it needs an integer count and a waiting object. P operation means existing process applies for resources to the system, decrease semaphore value by 1, such as s.value < 0, and then this process enters blocked queue. V operation means existing process releases this resource, increase semaphore value by 1 and number of system available resource can add one. If s.value ≤ 0, it means there is waiting process in blocked queue, and then it will wake up one of the first processes [3].

Synchronization Problem of Classical Process

Problem of producer-consumer

The problem description is: One group of producer process is producing product and the product will be provided to consumer for consumption. To ensure the concurrent execution between producer and consumer, set n buffer pools between them. Producer process can put their produced products into one buffer pool. Consumer process can get a product for consumption from one buffer zone.

Problem analysis: Set two synchronous semaphores: One is to explain the number of empty buffer zones, expressed with empty. The initial value is the number of buffer zones n in public buffer pool; the other is to explain the number of the full butter zones, expressed with full and the initial value is 0. It needs to operate bounded buffer zone in executing production activity and consumption activity. Bounded butter zone is a critical resource and has to be used mutually exclusive. Therefore, it needs to set one mutex and the initial value is 1.

Specific implementation code is in the Appendix Code 1.

Problem of reader and writer

Problem description: There are two groups of concurrent processes. Reader and writer share one data zone or one shared file. Requirement: It allows several readers to operate at the same time; it does not allow reader and writer to operate at the same time; it does not allow several writers to operate at the same time. When one reader process is reading, writer process is not allowed to write. The essence is that reader takes the priority.

Problem analysis:

If reader comes:

1) If there is no reader and writer, new reader can read.

2) If there is writer waiting, but other readers are reading, and then the new reader can read.

3) If there is writer writing, new reader shall wait.

If writer comes:

1) If there is no reader, new writer can write.

2) If there is reader, new writer should wait.

3) If there are other writers, new writer should wait.

The following codes adopt recorded semaphore set to solve reader-writer problem (reader takes priority):

Set two semaphores wmutex = 1, rmutex = 1

Set another global variable readcount = 0, it indicates the number of readers that are reading.

(Why use readcount for counting? If writer comes, it has to wait all readers to sign out. If there is no counting, how can we know that all the readers are signing out?)

Wmutex is used for the mutual exclusion between reader and writer, between writer and writer.

Rmutex is used for the mutex access of this critical resource.

Specific implementation code is in the Appendix Code 2.

From another perspective, what will be the situation if "writer takes priority"? That is when shared data zone is occupied by reader, the subsequent arrival readers can continue to access. If a writer comes and blocks the waiting, and then several readers come behind the writer will block waiting.

In other words, new reader will not be allowed to read data as long as there is one writer applying for writing data. This solution solves the problem of writer hungry, but it greatly decreases concurrent program and the system performance is poor.

Complicated process synchronization and mutual exclusion problem

Problem description: Three processes P1, P2 and P3 mutual exclusions use one buffer zone with N (N > 0) units. P1 generates one positive integer with product () and transfer to one empty unit in buffer zone with put (); P2 selects one odd from this buffer zone with get odd () every time and uses count odd () to make statistics for the number of odds; P3 selects one even from this buffer zone with get even () every time and uses count even () to make statistics for the number of evens. Please use semaphore mechanism to realize the synchronization and mutual exclusion activity of these three processes, explain the meaning of the defined semaphore and describe them with pseudo code.

Problem analysis: Buffer zone is a mutually exclusive resource; therefore, mutex semaphore is set. P1 and P2 are synchronous due to odd placement and retrieval; set synchronous semaphore odd; P1 and P3 are synchronous due to even placement and retrieval; set synchronous semaphore even; P1, P2 and P3 set synchronous semaphore empty due to shared buffer zone.

Specific implementation code is in the Appendix Code 3.

Conclusion

The synchronization mechanism of process well solves the numerous problems brought by process concurrency in operating system, but this is not the only method. In practical application, we should study and think more, ensure numerous concurrent processes share system resource and coordinate with each other more effectively and ensure the normal operation of all processes. In addition, the concurrency among numerous processes may make system trapped into deadlock. It should take corresponding measures to promote process get out of deadlock state quickly, so that it can improve the system performance. This is the target pursued by operating system.

References

1. Sun Zhongxiu, Fei Xiangli, Luobin (2008) Operating System Tutorial [M]. Higher Education Press, Beijing, China.
2. Tang Xiaodan, Liang Hongbing, Zhe Fengping (2003) Computer Operating System (revised) [M]. Xidian University Press, Xi'an, China.
3. Chen Xiangqun, Yang Fuqing (2006) Operating System Tutorial [M], 2 version. Peking University Press, Beijing, China.

Appendix

Code 1:

```
int in=0,out=0;
item buffer[n];
Semaphore mutex=1,empty=n,full=0;
void producer(){
do {
    ……
produce an item in nextp;
    ……
wait(empty); //waiting for the number of empty buffers is not 0.
wait(mutex); //waiting for no process to operate the buffer. These 2sentences reversed may be deadlocked
buffer(in)=nextp; //put the product on buffer[in]
in=(in+1)%n;
signal(mutex); //allow other processes to operate buffers
signal(full); //increase the number of buffers already used
    }while(1);
}
void consumer(){
do {
wait(full); //waiting for the number of empty buffers is not 0.
wait(mutex); //waiting for no process to operate the buffer. These 2sentences reversed may be deadlocked
nextc=buffer(out); //take out the product from buffer[out]
out=(out+1)%n;
signal(mutex); //allow other processes to operate buffers
signal(empty); //increase the number of buffers already used
consume the item in nextc; //consume the product of nextc
    ...
  } while(1);
}
void main() {   //main program
cobegin
producer(); consumer();
coend
}
```

Code 2:

```
semaphorermutex=1,wmutex=1;
intreadcount=0;
void reader(){
do{
wait(rmutex); //wait for no process to access the critical section of readcount if(readcount= =0) wait(wmutex); //wait
        for no writer to write
readcount++; //number of readers plus 1
signal(rmutex); //allow other process to visit readcount and read data
read data:
wait(rmutex); //wait for no process to access the critical section of readcountreadcount--;  //number of readers minus
        1
if(readcount==0) signal(wmutex); //allow writers to write
signal(rmutex); //allow other process to visit readcount
}while(1);
}

void writer() {
do {
wait(wmutex); //Wait for no one to write and read
write data;
signal(wmutex); //allow them to write and read
}while(1);
}
void main(){  //main program
cobegin
reader(); writer();
coend
}
```

Code 3:

```
Semaphore mutex=1;
Semaphore odd=0; Semaphore even=0;
Semaphore empty=N;
main()
```

```
cobegin
{
process P1
while(1){
number=produce();//generate a number
wait(empty);//determine whether the buffer has empty units
        wait(mutex);//determine whether the buffer is occupied
    put();
signal(mutex);//after using the buffer, buffer need to be released
if(number % 2==0)
signal(even);//If it is even, wake up the process in even
else
signal(odd); //If it is odd, wake up the process in odd
}
process P2
while (1){
wait(odd);//receive information from the odd semaphore and produce an
        odd number
wait(mutex); //determine whether the buffer is occupied
getodd();
signal(mutex); ///after using the buffer, buffer need to be released
signal(empty);//signals to the P1 can produce data
countodd();//statistic odd numbers
}
process P3
while (1){
wait(even); ///receive information from the even semaphore and produce an even number
wait(mutex); //determine whether the buffer is occupied
geteven();
signal(mutex); ///after using the buffer, buffer need to be released
signal(empty); //signals to the P1 can produce data
counteven();//statistic even numbers
}
}
coend
```