Chapter 4: Process Management

Operationg System

Dr. Do Tien Dung
dungdt@ptit.edu.vn

Posts and Telecommunications
Institute of Technology
Faculty of Information Technology 1

Faculty of Information Technology 1
Posts and Telecommnuication Institute of Techonology

August 15, 2022

Contents

Contents

Synchronize concurrrent processes

Concurrrent process or *Competing process* are processes running at the same time.
Manage concurrent processes includes following issues:

- Communicate between processes
- Compete and share resources
- Cooperate and synchronize between processes
- Deadlock
- Starvation

Process compete resources

- Compete directly:
  - Processes have to wait since only one is allocated resources
  - Affect running time of process
  - Indirect effect:
    - Because of the need of using resources such as memory, disk, I/O

For competed processes, it need to solve below issues:

- Ensure **loại trừ tương hỗ** (Mutual exclusion):
  - **Mutual exclusion** is a mechanísm to make sure the resources allocation of one process only and other process can not use that resources
  - That resources is critical resources. The program section using that resources is **critical section**
- No **deadlock**:
  - Deadlock: state that two or many processes can not run due to the waiting time without allocation.
  - For instance: Processes P1 và P2 need continoustly 2 resources T1, T2; OS allocates T1 to P1 and T2 to P2. P1 wait P2 to free up T2, while P2 wait for P1 to free up T1 => P1 and P2 go to deadlock state and can not continue running.
- Starvation

- Starvation is waiting too long without using resources.
- For instance: 3 processes P1, P2, P3 have request of using the same resources. P1 and P2 are allocted resources many times, while P3 does not have => can not run though do not have deadlock.

Process cooperate through share resources: Information exchange between cooperated process is sharing common memory (global variable), or files.

- Processes access data simultaneously may lead to some problems:
  - Ensure mutual elimination
  - Deadlock and starvation
  - Data consistency
- **Race condition**: threads/processes use shared data and results depend on the order of read/write
  - Put shared data access and update in dangerous section
  - Use the mutual elimination to not be interrupted by other process.

Communication process by passing messsage:

- Processes can exchange information directly by message passing.
- Message passing mechanism is supported by library of OS.
- Do not have requirement of mutual elimination
- May exist deadlock and starvation

Important requirement of process synchronization is to solve dangerous section and mutual elimination.
Solution of dangerous section has to meet following requirements:

- *Mutual exclusion*: at one moment, only one process is at dangerous section.
- *Progress*: one process is at dangerous section *is not allowed* preventing other processes from its dangerous section
- *Limited waiting time*: if a process has the need to get in the dangerous section, that process has to get in after a short amount of time.

Solutions for dangerous sections based on below assumptions:

- Do not depend on the speed of process
- A process does not allow to stay too long in the dangerous section
- Read/write memory is atomic action and can not be interrupted in the middele

Solutions are divided into 3 main groups:

- Software solutions: users do by themselves, OS provides some tools
- Hardware solutions: Interrupt disable, test-and-set
- Support from OS or libraries

Peterson algorithm proposed by Gary Peterson at 1981 for critical section is a software solution
Peterson algorithm proposes for synchronizing 2 processes P0 and P1:

- P0 and P1 share resources and have critical section
- Each process use loop and put critical section at the middle of whole process
- Require 2 processes exchange information via 2 variables:
  - *Int Turn*: identify which process goes to critical section
  - Flag for each process: flag[i]=true if process i request for going in critical section

Synchronize the simultaneous processes (cont.)
Peterson algorithm

```
…
bool flag[2];
int turn;

void P0(){     //tiến trình P0
    for(;;){     //lặp vô hạn
        flag[0]=true;
        turn=1;
        while(flag[1] && turn==1);//lặp đến khi điều kiện không thỏa
        <Đoạn nguy hiểm>
        flag[0]=false;
        <Phần còn lại của tiến trình>
    }
}

void P1(){     //tiến trình P1
    for(;;){     //lặp vô hạn
        flag[1]=true;
        turn=0;
        while(flag[0] && turn==0);//lặp đến khi điều kiện không thỏa
        <Đoạn nguy hiểm>
        flag[1]=false;
        <Phần còn lại của tiến trình>
    }
}

void main(){
      flag[0]=flag[1]=0;
      turn=0;
//tắt tiến trình chính, chạy đồng thời hai tiến trình P0 và P1
      StartProcess(P0);
      StartProcess(P1);
}
```

Hình 2.9: Giải thuật Peterson cho hai tiến trình

Requirements:

- Mutual exclusion
- Progress:
  - P0 can block P1 go in critical section if *flag[1]* = *true* and *turn* =*1* are true
  - There are 2 possibilities that P1 is out of critical section:
    - P1 is not ready to go to critical section => *flag[1]* = *false*, P0 can go to critical section
    - P1 sets *flag[1]*=*true* and in while loop => *turn* = *1* or *turn* = *0*:
    - Turn = 0: P0 goes to critical section
    - Turn = 1: P1 goes to critical section, then sets *flag[1]* = *false* => turns to Possibility 1
- Limited waiting time

Peterson algorithm:

- Quite complicated in reality
- The requesting process has to be in **active waiting** state
- Active waiting state: process still uses CPU to check whether it can go to critical section? wasting CPU

- Hardware can be designed to solve mutual exclusion and critical section.
- Hardware solution is easy to use and high speed.
- **Ban interrupt**: forbid interrupts during the time that process is in critical section.
- **Use special machine code**: Hardware is designed to have special codes
  - Check and change values of memory cell *Test_and_Set*, or compare/exchange values of 2 variables, are worked in the same code
  - Ensure it is worked together without interuption – atomic action

Mutual exclusion using Test_and_Set:

```
…
const int n; //n là số lượng tiến trình
bool lock;

void P(int i){     //tiến trình P(i)
    for(;;){     //lặp vô hạn
        while(Test_and_Set(lock));//lặp đến khi điều kiện không thỏa
        <Đoạn nguy hiểm>
        lock = false;
        <Phần còn lại của tiến trình>
    }
}

void main(){
      lock = false;
//tắt tiến trình chính, chạy đồng thời n tiến trình
      StartProcess(P(1));
      ...
      StartProcess(P(n));
}
```

Use special machine code:

- Advantages:
  - Simple and intuitive
  - Can be used to synchronize many processes
  - Can be used for multitasking with many CPUs
- Disadvantages:
  - Active waiting
  - Starvation

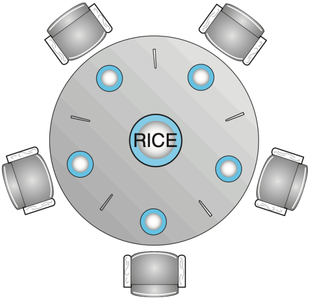
Semaphore S là a variable that is initiated to serve simultanously many processes
S value can change thanks to 2 actions *Wait* and *Signal*:

- Wait(S):
  - Decrease S one unit
  - If S<0, the process call: wait(S) will be blocked
  - If S>0, the process will be proceeded.
- Signal(S):
  - Increase S one unit
  - If S<=0: one of blocked processes will be released and continues to run

```
const int n; //n là số lượng tiến trình
semaphore S = 1;
void P(int i){    //tiến trình P(i)
    for(;;){    //lặp vô hạn
        Wait(S);
        <Đoạn nguy hiểm>
        Signal(S);
        <Phần còn lại của tiến trình>
    }
}
void main(){
//tắt tiến trình chính, chạy đồng thời n tiến trình
            StartProcess(P(1));
            ...
            StartProcess(P(n));
}
```

- To avoid active waiting, use 2 actions: block and wakeup:
  - If the process does "wait"action and S<0, it will be blocked and put in stack of flag
  - When a process does "signal"process, one of blocked processes will change to "ready"state through wakeup action
- When a process need to access resources, does "wait"action
- After finishing resources, the process does "signal"process: increase S and allow a blocked process to continue

Dining-Philosophers Problem:



- 5 philosophers sit on round table
  - There are 5 chópsticks on the table: on the left and right of each person, there is one chopstick

  - Phílosoper can pick up 2 chopstick in the order: pickup one that is on the table
  - During the chopstick holding, the philosopher eats and does not put down the chopstick
  - After eating, the person put down 2 chopsticks on the table
- 5 philosophers like 5 processes with critical resources being chopsticks and critical section is eating
- Flag allow solving following problem:
  - One chopstick is one flag
  - Pickup: wait()
  - Put down: signal()

Dining-Philosophers Problem:

```
…
semaphore chopstick[5] = {1,1,1,1,1,1};

void Philosopher(int i){          //tiến trình P(i)
   for(;;){    //lặp vô hạn
      Wait(chopstick[i]);         //lấy đũa bên trái
      Wait(chopstick[(i+1)%5]);  //lấy đũa bên phải
      <Ăn cơm>
      Signal(chopstick[(i+1)%5]);
      Signal(chopstick[i]);
      <Suy nghĩ>
   }
}

void main(){
// chạy đồng thời 5 tiến trình
     StartProcess(Philosopher(1));
     ...
     StartProcess(Philosopher (5));
}
```

Hình 2.15. Bài toán triết gia ăn cơm sử dụng cờ hiệu

Producter, customer with limited storage
Producter: make product, put it in a storge, one product per time
Customer: take the products from storage, each time one product
Capacity of storage is limited, N products limits

- Three requirements for synchronization:
  - Producers and consumers cannot use the buffer at the same time
  - When the buffer is empty, the consumer should not attempt to retrieve the product
  - When the buffer is full, producers cannot add products
- Solving by flag:
  - Requirement 1: use lock flag initialized to 1
  - Requirement 2: flag is empty and initialized to 0
  - Requirement 3: full flag, initialized with N

Synchronize the simultaneous processes (cont.)

Some synchronization problems

Producter, customer with limited storage

Const int N; // kích thước bộ đệm
Semaphore lock = 1;
Semaphore empty = 0;
Semaphore full = N

```
Void producer () {
    for (; ;) {
        <sản xuất>
        wait (full);
        wait (lock);
        <thêm 1 sản phẩm vào bộ đệm>
        signal (lock);
        signal (empty);
    }
}
```

```
Void consumer() {
    for (; ;) {
        wait (empty);
        wait (lock);
        <lấy 1 sản phẩm từ bộ đệm>
        signal (lock);
        signal (full);
        <tiêu dùng>
    }
}
```

```
Void main() {
    startProcess(producer);
    startProcess(consumer);
}
```

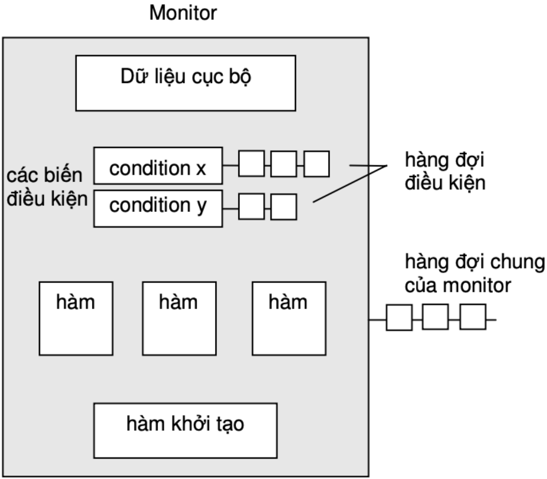
Monitor ís defined as an abstract data type in a high-level programming language, such as a class in C++ or Java. Each monitor consists of its own data, constructor, and a number of functions or methods to access the data with the following characteristics.

- Process/thread can only access monitor data through monitor functions or methods
- At each moment:
  - Only one process is executed in the monitor
  - Other processes calling the monitor's function will be blocked and placed in the monitor's queue to wait until the monitor is released

- Ensuring mutual exclusion for dangerous segments, placing dangerous resources in the monitor
- The process is executing in the monitor and is stopped to wait for an event or a certain condition to be satisfied => returning the monitor for another process to use.
- The waiting process will be restored from the breakpoint after the waiting condition is satisfied => Using condition variables

- Conditional variables are declared and used in monitor with 2 operations:*cwait()* và *csignal()*:
  - *x.cwait():*
    - The process that is in monitor and calling cwait is blocked until condition X occurs
    - The process is queued for the condition variable X
    - Monitor is released and another process is entered
  - *x.csignal():*
    - The process calls csignal to notify the condition X has been satisfied
    - If there is a process that is blocked and in x's queue due to a previous X.cwait() call, it will be released.
    - If there is no blocked process, the csignal operation will have no effect at all

Monitor structure with condition variables:



Conclusion

Chapter 4 Process management

- Synchronize the simultaneous processes
- Deadlock and starvation

Conclusions

- Projects
- Examinations
- Questions