Operating systems

ELTE IK.

Dr. Illés Zoltán

zoltan.illes@elte.hu

Financed from the financial support ELTE won from the Higher Education Restructuring Fund of the Hungarian Government

We have talked earlier…

- Progress of Operating systems
  - Computer – Op.system generations
- Notions of Operating system, structures
  - Client-server model, system call
- Files,directories, filesystems
  - Hardware architecture
  - Logical architecture
  - FAT, UNIX, NTFS,…
- Processes, process communications
  - Race conditions, critical sectionmodel

What comes today…

- Process commucation
  - Monitors
  - Message sending
- Classical IPC problems
  - Case of eating philosopers
- Scheduling of processes
  - Theories, implementations
  - Thread scheduling

Is there any problem with semaphores?

- No…but small mistyping might cause big problems.
  - E.g. Let's change 2 rows (red), when bakery is full, the baker is stopped by semaphor empty, at the same time the customer is stopped by free, therefore both processes are stopped! (deadlock)

```
void baker() /*original recipe
*/
{
  int bread;
  while (1)
   {
     bread=baker_bake();
     down(&empty);

     down(&free);
     to_shelf(bread);
     up(&free);
     up(&full);
   }
}
```

```
void baker() /*modified recipe*/
{
  int bread;
  while (1)
   {
     bread=baker_bake();
     down(&free);  /*changed part*/
     down(&empty);
     to_shelf(bread);
     up(&free);
     up(&full);
   }
}
```

Do we have any problem with semaphors?

- Generally no! But as we saw, a little mistake causes program error! (Big error!).
- Also the changing two up and down instructions can cause same problem!
- Other reason: if we forget it…
- Do we have any better thing to do?
  - At Kernel level no.

Monitors

- Brinch Hansen (1973), Charles Anthony Richard Hoare (1974) suggested a higher level construction supported by languages.
- This construction is called: monitor
  - It is similar to a class definitions

```
Monitor critical_zone
          Integer shelf[];
          Condition c;
          Procedure baker(x);
          …
          End;
          Procedure customer(x);
          …
          End;
End monitor;
```

Monitor properties

- We can define inside a monitor procedures, data structures.
- At the same time only 1 process can enter into a monitor!
- This is supported by programming language compiler.
  - If a process calls a method inside a monitor, first it cheks whether is there any other active process inside the monitor?
    - If yes, the process will be hold, suspended.
    - If not, the process enters into the monitor.

Monitor implementation

- Using Mutex
- A customer has no concrete idea about implementations, but it is not needed.
- Result: a more safety mutual exclusion
- Only we have one problem: what about if a process can’t continue his work inside a monitor?
  - E.g: a baker can’t bake bread, because the bakery shelves are full!
- Solution: lets create a condition variable
  - We can do 2 operation on condition variable: wait, signal

A Solution of the producer-consumer problem with monitors I.

N element

```
monitor Baker-Customer
          condition full, empty;
          int number;
          bread_to_shelf(bread  element)
          {
                    if (number==N) wait(full);
                    shelf(element);
                    number++;
                    if (number==1) signal(empty);
          }
          bread bread_from_shelf()
          {
                    if (number==0) wait(empty);
                    bread element=bread_from_shelf();
                    number--;
                    if (number==N-1) signal(full);
                    return element;
          }
 end monitor
```

The process of Pék&Vásárló

```
baker()
{
        while(1)
        {
                bread new;
                new=bread_bake();
                Baker-Customer.bread_to_shelf (new);
        }
}
customer()
{
        while(1)
        {
                bread new_bread;
                new_bread=Baker-Customer.bread_from_shelf();
                eat(new_bread);
        }
}
```

Other solutions

- The previous example based on a theoretical language: Pidgin Pascal
- In C there is no monitor
  - In C++ there is monitor, also wait, notify
- Java:
  - Synchronized methods
  - No condition variable, but there are wait & notify
- C#
  - Monitor class
    - Enter,TryEnter,Exit,Wait,Pulse (this same as notify)
  - Lock keyword
  - Sample: VS2008 solution, monitor project.

Do we have any problem with monitors?

- Hm,…no, we have not!
- It is more safety than a simple semaphor.
- It works with one or more CPU, but only with a common memory space!
- If a CPU has a standalone memory this solution does not fit, does not work!

Message sending

- Processes typically uses two general method:
  - Send(to, message)
  - Receive(from, message)
    - Source may be optional!
- These are system calls and not language constructions!
- Receipt message: If a sender wants to know about accepting message, the receiver process sends back a receipt!
  - If a sender does not get the receipt, it will send again the message!
  - If the receipt will be lost, the sender also will send again!
  - A message contain a number which holds the message number too to distinguish(mark) messages .

A Solution of the producer-consumer problem with message sending. I.

- The producer (baker) process:

```
#define N 100                    // size of shelves
void baker()                     // baker process
{
        int bread;               // „bread” variable, storage
        message m;               // message storage place
        while(1) // baker works all time
        {
                bread= bake_bread();
                receive(customer,m);      // we wait a message from
                                           //the customer
                                          // empty message in m
                m=create_message(bread);
                send(customer,m);// send the bread to the customer
        }
}
```

A Solution of the producer-consumer problem with message sending. II.

- Consumer process :

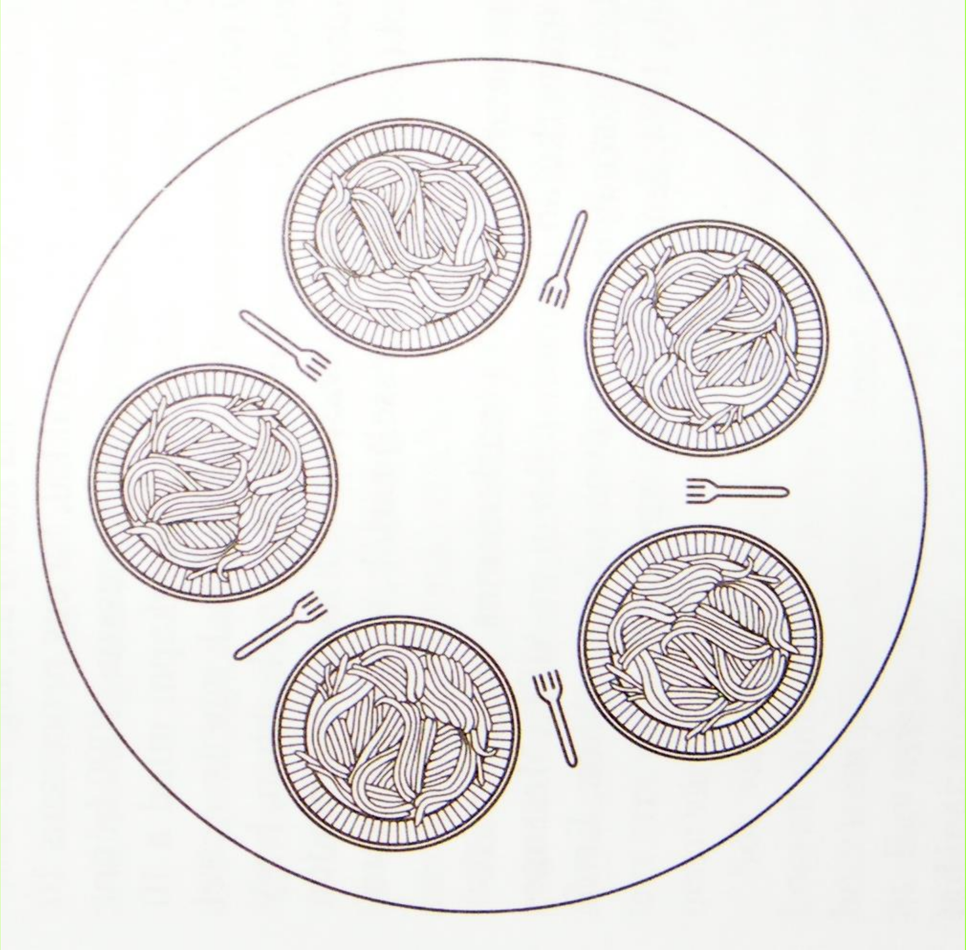
```
void customer()                // consumer (customer) method
{
        int bread;             // „bread” variable
        message m;             // the place for the message
        int I;
        for(i=0;i<N;i++) send(baker,m);
                               //setup N empty space for bred
                               //sending to baker
        while(1) // the shopping is going on all the time too
        {
                receive(baker,m);          // buying a bread
                bread=message_outpack(m);
                send(baker,m);   // sending back empty store
                eat_bread (bread);
        }
}
```

Summary of message sending

- We have to create temp. place for messages on both places. (like mail store)
- We can omit this, in this case the receive statement will be blocked before send command!
  - Trysting strategy.
  - Minix 3 also uses this one!
  - Messages are the same as pipes, but inside pipes there is no any delimiters!
- The message sending is a general technique in parallel systems.

Classic IPC problems I.

- Eating philosophers case:
  - They need 2 forks to eat spagetti!
  - All philosophers can access only forks near his own plate.
  - They are thinking-eating…
  - Lets create a program to solve this problem!



Solution I.

- It has a problem, all processes can be in deadlock if all gets the lefts fork and will wait for the rigth one!.
- If somebody puts back left fork and again try to get both also not good, because all processes do this! (Starvation)

```
Void philosopher (int i)
{
  while(1)
    {
          thinking();
          need_fork(i);   // left fork
          need_fork ((i+1)%N); //right
          eating();
          dont_need_fork (i);
          dont_need_fork ((i+1)%N);
    }
}
```

Solution II.

```
int s[5];                  // values: eat, hungry, thinking
semaphore safe_s=1; //a sign for using s array
semaphore philo[5]={0,0,0,0,0}; //1 semaphore to each philosopher, 0=not
permitted
Void philosopher(int i)
{
 while(1)  {
    thinking();
    down(safe_s);    //only i modify s[]
    s[i]=hungry;       //
    if (s[left]!=eat && s[right]!=eat) //whether the two neighbour’s fork are free?
      { s[i]=eat; up(philo[i]); };  //i eat, philo[i] shows free
    up(safe_s);         // other may access s[]
    down(philo[i]);
// it is blocked, if there is no 2 forks, if the i. Philosopher is not  eating
    spaghetti_eating();
    down(safe_s); // finished eating, again protect s because i modify it
    s[i]=thinking;
    if (s[left]==hungry && s[left--]!=eat) { s[left]=eat;up(philo[left]);}
    if (s[right]==hungry && s[right++]!=eat) { s[right]=eat;up(philo[right]);}
    up(safe_s);
  }
}
```

Solution III.

- Lets 5 forks semaphore[] for each fork.
- Max semaphore
- Sample to access restricted resources.

```
Int N=5;
semaphore forks[]={1,1,1,1,1}; //all free
semaphore max=4; //max 4 fork used in the same time
void philozophus(int i)
{
  while(1)
    {
            thinking();
            down(max);
            down(forks[i]);    // left fork
            down(forks[(i+1)%N];    //right
            eating();
            up(forks[i]);
            up(forks[(i+1)%N]);
            up(max);
    }
}
```

Readers–Writer problem

- The database can read by more than 1 process, but only 1 can write:

```
// writing process
semaphore database=1;
semaphore mutex=1;
int rc=0;
Void writer()
{
  while(1)
  {
     do_something();
     down(database);    // critical
     write_to_database();
     up(database);
  }
}
```

```
Void reader()
{
  while(1)
  {
     down(mutex);
     rc++;
     if (rc==1) down(database);
     up(mutex);
     Read_from_database();
     down(mutex);    // critical
     rc--;
     if (rc==0) up(database);
     up(mutex);
     working_on_data();
  }
}
```

Scheduling

- We have seen before that several processes may run „parallel” to each other.
- Only 1 at a time can run.
- Which one should run?
- The scheduler makes the decision
- The decision is based on the scheduling algorithm

The I/O need of processes

- Typically a process may have two different types of work:
  - It is counting…
  - I/O need, wants to read or write data from/to the periphery
- CPU-bound process
  - It is working (count) a lot, it does not wait to much for I/O
- I/O-bound process
  - Works only for a small amount of time it is waiting for I/O long

When do we have to switch to a new process?

- There is a process (context) switch:
  - If a process ended
  - If a process' state becomes blocked (because I/O or semaphore)
- Usually there is a context switch:
  - A new process is created
  - I/O interrupt occurs
    - After an I/O interrupt, a blocked process, the process which waited for it typically may continue its running.
  - Timer interrupt
    - Preemptive scheduling
    - Non-preemptive scheduling

Groups of scheduling

- Representatives for each system:
  - Fairness – everybody can use CPU
  - The same rules are valid for everybody
  - Balance –Everybody should get the same loading
- Batched systems
  - throughput, turnaround time, CPU utilization
- Interactive systems
  - Response time, proportionality to the user's expectations
- Real-time systems
  - To keep deadlines, to avoid data loss, quality corruption

Scheduling in batch systems I.

- Ranking scheduling, it is non-preemptive
  - First Come First Served – (FCFS)
  - The process runs till it stops or becomes blocked.
  - If it blocks, it goes to the end of the queue.
  - The processes are in a fair, simple, chained list.
  - Disadvantage: I/O-bound processes are very slow.
- The shortest job runs first, non-preemptive (SJB)
  - We have to know the running time ahead
  - It is optimal, if everybody is known at the beginning

Scheduling in batch systems II.

- The process with the shortest remaining time runs
  - Preemptive, monitoring at each entering.
- Three level scheduling
  - Inlet scheduler
    - It lets the task into the memory in rotation.
  - Disk scheduler
    - If it lets in to much processes and memory becomes full then they all have to be written to the disk and back.
    - It runs rarely.
  - CPU scheduler
    - We may choose from the previously mentioned algorithms.

Scheduling in interactive systems I.

- Round Robin
  - Everybody gets a time quantum, at the end of it or in the case of blocking comes the following process
  - At the end of the quantum the next process in the round list will be the actual
  - Fair, simple
  - We may store the processes (features) in a list and we go round and round all the time.
  - There is only one question: How big should be the quantum?
    - The process switch needs some amount of time.
    - Small quantum-> a lot of CPU time for switching
    - Too big quantum -> the interactive user feels it too slow (keyboard handling)

Scheduling in interactive systems II.

- Priority scheduling
  - Importance, priority is shown
    - Unix: 0-49 -> not preemptive (kernel) priority
    - 50-127 -> user priority
  - The process with the highest priority may run
    - Modifying: dynamical priority to avoid starvation
  - Usage of priority classes
    - Usage of Round Robin among the same class processes
    - Must modify the priority of the processes, because low priority processes get CPU rarely.
    - Typically at each 100. quantum the priorities are recounted
      - Typically the high priority processes become lto ower level, then comes RR. At last the original classes will be built up again.

Scheduling in interactive systems III.

- Multiple queues
  - Also have priorities and uses RR
  - At the highest level each quantum gets 1 time quantum.
  - The next 2, then 4, 8, 16,32,64.
  - If the time of the highest process is used up it go down with one level.
- The shortest process next
  - Though we do not know the remaining execution time we have to estimate it!
  - Aging, weighted average for the time quantum.
    - T0, T0/2+T1/2, T0/4+T1/4+T2/2, T0/8+T1/8+T2/4+T3/2

Scheduling in interactive systems IV.

- Guaranteed Scheduling
  - Each active process gets proportional CPU time.
  - Must be registered how many time was used up for a process and if somebody has got less time it „goes" forward.
- Lottery scheduling
  - Similar to the previous one, except the processes gets some lottery ticket. That process can run which has got the pulled out one.
  - It is easy to support proportional CPU time, useful e.g. at video servers
- Fair-share scheduling
  - Let's pay attention to the user as well! Similar to guaranteed one – only it refers to the users

Scheduling in real-time systems I.

- What is a real-time system?
  - The time is an important actor. We must garantee to give a response within the deadline.
  - Hard Real Time, absolute, not modifyable deadlines.
  - Soft Real Time (tolerant), there are deadlines, but a small difference is tolerable.
  - The programs are devided into several smaller process.
  - In the case of an outer event, have to give a response within the deadline.
  - Schedulable: if the response CPU time sum of n event is <=1.
- Unix, Windows are real-time systems?

Scheduling ideas, implementation

- Frequently there are child processes in the system.
- It is not sure that the priority of the parent and the child process must be the same.
- Typically the kernel uses priority scheduling (+RR)
  - It warrants a system call with which the parent may give the priority of the child
  - Kernel schedules – the user process modify the priority (nice)

Threadscheduling

- User level threads
  - The Kernel does not know about it, the process gets a quantum, within the thread scheduler makes the decision who should run
  - Quick switch between the threads
  - It is possible to use application dependent thread scheduling
- Kernel level threads
  - Kernel knows the threads, Kernel decides which process which thread should run.
  - Slow switch, between the switch of two threads needs a full context switch
  - This is also noticed.

Thanks for your attention!

zoltan.illes@elte.hu