Politecnico di Milano

Inter Process Interaction

Synchronization and Mutual Exclusion

Lecturer:
William Fornaciari
Politecnico di Milano
fornacia@elet.polimi.it
www.elet.polimi.it/~fornacia

Summary

Motivations

- Several processes exist simultaneously in a system
  - co-existing processes may interfere with each other's execution
    - incorrect calculations,decrease in efficiency,deadlock
- Processes may cooperate within the same task
  - the activities of cooperating processes need to be coordinated
- The use of shared resources must be coordinated
- The separation of independent activities in processes may lead to higher performance and more convenient use

Concurrency: different contexts

- Multiple applications
  - process multiplexing: the CPU switches between different processes
  - to the human user, these processes seem to be executed simultaneously
- Structured application
  - Application can be a set of concurrent processes
- Operating-system structure
  - Operating system is a set of processes or threads

Difficulties with concurrency

- Sharing global resources
  - several processes can utilize them simultaneously
- Management of allocation of resources
  - the operating system must manage the use of resources
- Programming errors difficult to locate
  - error discovering can become very hard

A simple example of error

- The processes P1 and P2 are both in execution on a different processor
- Two events on the same line happen at the same time

```
Process P1                  Process P2
.                           .
in = getchar();             .
.                           in = getchar();
chout = chin;               chout = chin;
putchar(chout);             .
.                           putchar(chout);
.                           .
```

- Error! : the char read by P1 is lost

Concurrency problems in O.S.

- Keep track of active processes
- Allocate and deallocate resources
  - processor time
  - memory
  - files
  - I/O devices
  - ...
- Protect data and resources
- Result of process must be independent of the speed of execution since other processes share the processor time

Process interaction

- Processes unaware of each other
  - the processes are independent, they cannot affect or be affected by the execution of another process
  - possible competition for resources
- Processes indirectly aware of each other
  - the processes share the access to some objects
  - it is possible cooperation among processes
- Process directly aware of each other
  - the processes work together communicating one another
  - cooperative *nature* of processes interaction

Processes unaware of each other (competition)

- If two processes wish access to a single resource, one process take the resource and the other will have to wait
- Competition Among Processes for Resources (Control Problems)
  - Mutual Exclusion
    - Critical sections
      - Only one program at a time is allowed in a critical section
      - Example: only one process at a time is allowed to send command to the printer
  - Deadlock
  - Starvation

Process indirectly aware of others (cooperation)

- Processes use and update shared data such as shared variables, files, and data bases
- Writing must be mutually exclusive
- Critical sections are used to provide data integrity
- Cooperation Among Processes by Sharing (Control Problems)
  - Mutual Exclusion
  - Deadlock
  - Starvation
  - Data Consistency

Process directly aware of other (cooperation)

- Communication provides a way to synchronize, or coordinate, the various activities
- Cooperation Among Processes by Communication (Control Problems)
  - Possible deadlock
    - each process waiting for a message from the other process
  - Possible starvation
    - two processes sending message to each other while another process waits for a message

Mutual exclusion: requirements

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its non-critical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Critical section

- Several processes cooperate on a task by sharing resources
- A critical section is a segment of code in which a process utilizes some shared resources, and no other process may interfere
  - execution of critical sections must be mutually exclusive in time
  - there may be different critical sections for different sets of shared resources
- A protocol us required to guarantee that only one process is in the critical section

Solution to critical-section problem (1)

- Solutions for the critical section problem must satisfy
  - mutual exclusion
    - if one process is executing in its critical section, then no other processes can execute in their critical sections
  - progress
    - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

Solution to critical-section problem (2)

- bounded waiting
  - after a process has made a request to enter its critical section, only a finite number of other processes may go first
- assumptions
  - processes execute at non-zero speed
  - no assumptions on their relative speed

Algorithm 1 : busy waiting

- Processes P0 and P1 share a common variable `turn` initialized to `0` or `1`.
- If `turn = 0` then process P0 enters critical section
- If `turn = 1` then process P1 enters critical section

- Process P_i

```
repeat
        while turn ≠ i do no-op;
                critical section
        turn := j;
                reminder section
until false;
```

Evaluation of algorithm 1

- Requirements satisfied?
  - mutual exclusion
    - yes, only one process is allowed in its critical section
  - progress
    - no, requires strict alternation
  - bounded waiting
    - yes, exactly one waiting period is required

Algorithm 2 : busy waiting

- An array `flag` of two elements `flag[0]` and `flag[1]` indicates if a process wants to enter its critical section
  - if `flag[i]` is true, then Pi is ready to enter its critical section
- Process P_i

```
repeat
        while flag[j] do no-op;
                flag[i]:=true;
                critical section
                flag[i]:=false
                reminder section
until false;
```


Evaluation of algorithm 2

- Requirements satisfied?
  - mutual exclusion
    - This method does not guarantee mutual exclusion
    - Each process can check the flags and then proceed to enter the critical section at the same time
  - progress
    - yes, if no other process is in the critical section, it sets its status for the critical section
  - bounded waiting
    - yes, a process sets `flag` in the exit section such that the other can continue

Algorithm 3 : busy waiting

- if `flag[i]` is true, then Pi is ready to enter its critical section
- elements of the array `flag` are initialized to `false`
- Process P_i

```
repeat
        flag[i] := true;
        while flag[j] do no-op;
                critical section
        flag [i] := false;
                remainder section
until false;
```

Problem with algorithm 3

- Problem
  - What happens if P0 sets `flag[0]` to `true` and P1 sets `flag[1]` to `true` before they get into their while statement?
- Answer
  - Infinite loop within **`while`** statements
    - depends too much on the exact timing of the two processes
    - not very likely, but may happen

Evaluation of algorithm 3

- Requirements satisfied?
  - mutual exclusion
    - yes, only one process is allowed in its critical section
  - progress
    - no, endless loop in the entry sections possible if both processes set their flags simultaneously
    - switching the order of the two instructions in the entry section is not a solution, it will lead to mutual exclusion problems
  - bounded waiting
    - yes, a process sets `flag` in the exit section such that the other can continue

Algorithm 4: yet another busy waiting

- Process P_i:

```
repeat
        flag[i] := true;
        while flag[j] do
                begin
                        flag [i] := false;
                        <small pause>;
                        flag [i] := true;
                end;
        critical section
        flag [i] := false;
        remainder section
until false;
```


Evaluation of algorithm 4

- Requirements satisfied?
  - mutual exclusion
    - yes, only one process is allowed in its critical section
  - progress
    - no, endless loop in the entry sections possible if both processes set their flags simultaneously
  - bounded waiting
    - yes, a process sets `flag` in the exit section such that the other can continue

Correct solution : Peterson's algorithm

- Processes share two Boolean variables
  - `turn`
  - the array `flag`
- Process P_i

```
repeat
        flag [i] := true;
        turn := j;
        while (flag [j] and turn = j) do no-op;
                critical section
        flag [i] := false;
                remainder section
until false;
```


Evaluation of Peterson's algorithm

- Requirements satisfied?
  - mutual exclusion
    - yes
  - progress
    - yes
  - bounded waiting
    - yes
- This algorithm is a satisfactory solution for the two-process critical section problem

Mutual Exclusion: Hardware support

- Disabling Interrupts
- Special Machine Instructions

Disabling Interrupts (1)

- A process runs until it invokes an operating-system service or until it is interrupted
- Interrupts are disabled during the time period when mutual exclusion is required
- Processor is limited in its ability to interleave programs (without interrupts no process switching can occur)
- Efficiency of execution could be noticeably degraded

Disabling Interrupts (2)

- Somewhat dangerous: one process can hold up the whole system
  - endless loop
  - waiting for resources
- Multiprocessing
  - disabling interrupts on one processor will not guarantee mutual exclusion for multiprocessing system
- Used in special-purpose systems with limited hardware (only one processor)

Special Machine Instructions (1)

- Instructions that do two steps indivisibly
  - Performed in a single instruction cycle
  - Not subject to interference from other instructions
  - **test_and_set**
    - test a value; if it is **false** set it to **true**, else leave it as **false (**reading and testing)
  - **Exchange**
    - swap the values of two variables (reading and writing)
- One machine instruction is used to update a memory location so other instructions cannot interfere
- It can be used for multiple critical sections

Special Machine Instructions (2)

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify

Test and Set Instruction

```
boolean testset (int i) {
        if (i == 0) {
                i = 1;
                return true;
        }
        else {
                return false;
        }
}
```


Special Machine Instructions (3)

Exchange Instruction

```
  void exchange(int register,int memory)
 {
     int temp;
     temp = memory;
     memory = register;
     register = temp;
   }
```

Special Machine Instructions(4)

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible
    - when a process leaves a critical section and more than one process is waiting. Who is next?
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Sleep and Wakeup

- Both Peterson's and TSL solution are correct but require busy waiting
  - wasting of CPU time
  - priority inversion
    - Two processes (H, L) with High and Low priority. Scheduling rule: H is run whenever it is in ready state
    - If L is in critical region and H becomes ready (e.g. I/O completes), H begins busy waiting, but sice L is never scheduled while H is running, L never exit its critical region and H loops forever
- Possible solution: SLEEP & WAKEUP primitives
  - System calls that cause the caller to block (suspend) without wasting CPU time, until another process wakes it up through

Semaphore (Dijkstra 1965)

- Used to solve synchronization problems among *n* processes
  - fundamental synchronization tool used in many operating systems
- It is an integer variable that can be accessed only via two atomic operations
  - `wait`
  - `signal`
- If a process is waiting for a signal, it is suspended until that signal is sent
- When a process blocks itself on a semaphore, it is added to a queue of waiting processes

Semaphore implementation (1)

- Semaphore is a variable that has an integer value
  - Define a semaphore as a record

```
type semaphore = record
                     counter: integer
                     queue: list of process;
                 end;
```

  - Wait operation decrements the semaphore value

```
wait(S):
S.counter := S.counter – 1;
if S.counter < 0
        then begin
                add this process to S.queue;
                block this process;
        end;
```


Semaphore implementation (2)

- Signal operation increments semaphore value (this operation removes a process from the queue and wakes the process up)

```
signal(S):
S.counter := S.counter + 1;
        if S.counter ≤ 0
                then begin
                remove a process P from S.queue;
                wakeup(P);
        end;
```

- Semaphore variable may be initialized to a nonnegative number

Critical sections and semaphores

- Operations on semaphores must be atomic
  - we must ensure that no two processes can execute *wait* and *signal* on the same semaphore at the same time
- Uni-processor system
  - hardware support such that transactions are atomic
  - utilize interrupts during the time *wait* and *signal* are executing
- Multiprocessor system
  - hardware support is expensive for multiple CPUs
  - employ software algorithms if hardware instructions are not available

Deadlock and starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

Binary semaphores

- semaphores in which their integer values can only be either 0 or 1
- used by 2 or more processes to ensure only one can enter critical section

Binary semaphore implementation (1)

- Semaphore is a variable that has an integer value (can only be either 0 or 1)
  - Define a binary_semaphore as a record

```
type binary_semaphore = record
                            value:  (0,1);
                            queue: list of process;
                        end;
wait(S):
if S.value ==1 then
                S.value := 0;
            else begin
                add this process to S.queue;
                block this process;
end;
```


Binary semaphore implementation (2)

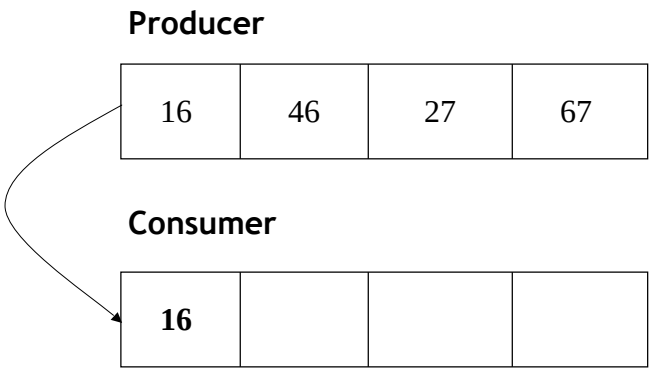
- Signal operation increments binary_semaphore value (this operation removes a process from the queue and wakes the process up)

```
signal(S):
if S.queue is empty then
            S.value := 1;
else begin
            remove a process P from S.queue;
            wakeup(P);
end;
```

Producer-consumer problem

- One producer is generating data and placing these in a buffer
- A consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time
- Three semaphores are used
  - two to represent the amount of items in the buffer (full, empty)
  - one to signal that it is all right to use the buffer (mutex)

Diagram of producer-consumer problem



Semaphore solution (1.1)

- Shared data

```
type item = …
var buffer = …
        full, empty, mutex: semaphore;
        nextp, nextc: item;
        full :=0; empty := n; mutex :=1;
```

- The buffer has n elements
- mutex is used for granting mutual exclusion

Semaphore solution (1.2)

- Producer Process

```
repeat
  ...
   -- produce an item in  nextp
  ...
   wait(empty);
   wait(mutex);
  ...
   -- add nextp to  buffer
  ...
   signal(mutex);
   signal(full);
until false;
```

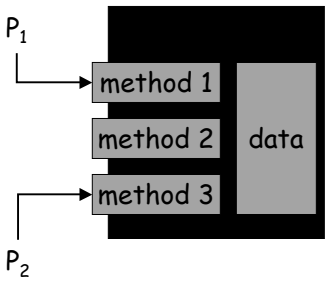
Semaphore solution (1.3)

- Consumer Process

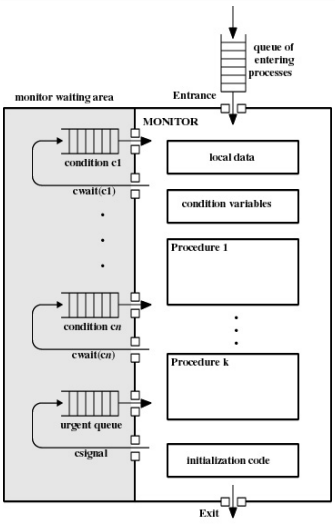
```
repeat
    wait(full);
    wait(mutex);
   ...
    -- remove  an item from buffer to nextc
   ...
    signal(mutex);
    signal(empty);
   ...
    -- consume the item in nextc
   ...
until false
```


Monitor (Hoare '74, Hansen '75)

- Collection of procedure, variables anda data structures that are grouped together in a package, similarly to an Abstact Data Type
- Processes may call procedures in a monitor but cannot directly access the internal data structures
- Mutual exclusion: only one process can be active in a monitor at any instant



Monitor (Hoare '74, Hansen '75)



- There exists *condition variables*, manipulated trough WAIT and SIGNAL primitives, acting as queues, where the processes can be suspended and awaked
- All procedures are critical sections
- Automatic implementation of critical regions simplify parallel programming

Monitors: declarations

```
type monitor-name = monitor
        variable declarations
        procedure entry P1 :(…);
        begin … end;
            procedure entry P2(…);
        begin … end;
        .
        .
        .
        procedure entry Pn (…);
        begin…end;
        begin
        initialization code
        end
```

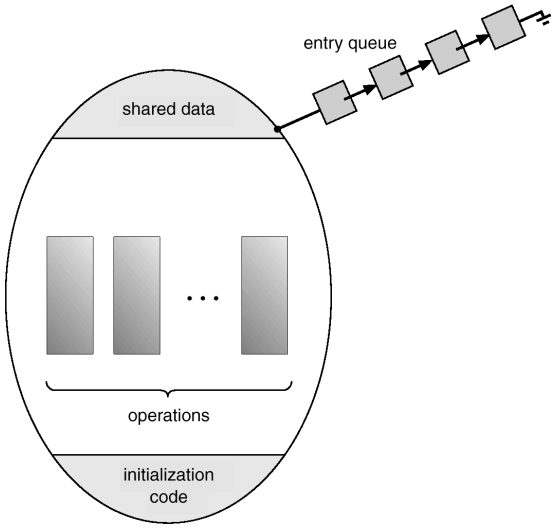

Monitors with condition variables

- To allow a process to wait within the monitor, a condition variable must be declared, as

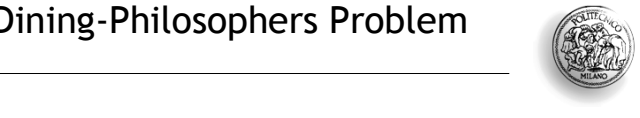
  ```
  var x, y: condition
  ```

- Condition variables can only be used with the operation wait and signal
- `x.wait` means that the invoking process is suspended until another one invokes `x.signal`
- `x.signal` resumes exactly one suspended process, if no one exists it has no effect
- Typically a process doing a `signal` must exist the monitor immediately to prevent the possibility to have two active processes within the monitor

Monitor with condition variables

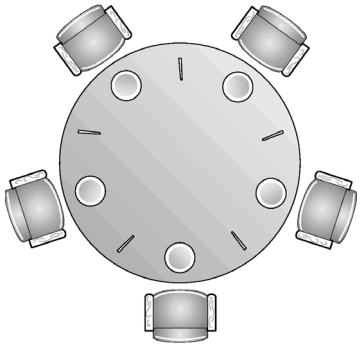


Example: Dining-Philosophers Problem



- Shared data

  Semaphore chopStick[] = new Semaphore[5];



Dining Philosophers

```
Philosopher i:
  repeat
  wait(chopstick[i])
  wait(chopstick[i+1 mod 5])
  …
  eat
  …
  signal(chopstick[i]);
  signal(chopstick[i+1 mod 5]);
  …
  think
  …
  until false;
```


Dining-Philosophers

- Philosopher *i*:

```
while (true) {
  // get left chopstick
  chopStick[i].P();
  // get right chopstick
  chopStick[(i + 1) % 5].P();

  // eat for awhile

  //return left chopstick
   chopStick[i].V();
  // return right chopstick
   chopStick[(i + 1) % 5].V();

  // think for awhile}
```


Dining Philosophers

type *dining-philosophers* = **monitor**

var *state* : **array** [0..4] **of** :(*thinking, hungry, eatin*g);

var *self* : **array** [0..4] **of** *condition*;

procedure entry *pickup (*i: 0..4);

begin

state[i] := *hungry*,

test (i);

if *state[i]* ≠*eating* **then** *self[i], wai*t,

end;

Dining Philosophers

```
procedure entry putdown (i: 0..4);
    begin
        state[i] := thinking;
        test (i+4 mod 5);
        test (i+1 mod 5);
    end;
```


Dining Philosophers

```
procedure test(k: 0..4);
begin
    if state[k+4 mod 5] <>eating
    and state[k] = hungry
    and state[k+1 mod 5] ] <>eating
    then begin
            state[k] := eating;
            self[k].signal;
    end;
end;
begin
for i := 0 to 4
do state[i] := thinking;
end.
```


Solution to Dining Philosophers

```
monitor diningPhilosophers {
  int[] state = new int[5];
  static final int THINKING = 0;
  static final int HUNGRY = 1;
  static final int EATING = 2;
  condition[] self = new condition[5];
  public diningPhilosophers {
        for (int i = 0; i < 5; i++)
              state[i] = THINKING;
  }
  public entry pickUp(int i) { /* see next slides */ }
  public entry putDown(int i) { /* see next slides */ }
  private test(int i) {/* see next slides */ }
}
```


Monitors implementation via semaphores

Variables

var *mutex: semaphore* (init = 1)

next: semaphore (init = 0)

next-count: integer (init = 0)

• Each external procedure *F* will be replaced by

wait(*mutex*);

…

body of F;

…

if next-count > 0

then signal(next)

else signal(mutex);

Note: mutual exclusion within a monitor is ensured

Monitor implementation via semaphores

- For each condition variable x, we have

  var *x-sem*: *semaphore* (init = 0)

  x-count: *integer* (init = 0)

- The operation x.wait can be implemented as

  x-count := *x-count* + 1;

  if *next-count* >0

  then *signal(next)*

  else *signal(mutex);*

  wait(x-sem);

  x-count := *x-count* – 1;

Monitor implementation

- The operation x.signal can be implemented as

  if *x-count* > 0

  then begin

  next-count := *next-count* + 1;

  signal(x-sem);

  wait(next);

  next-count := *next-count* – 1;

  end;

Monitor implementation

- *Conditional-wait* construct: ***x.wait(c);***
  - *c* – integer expression evaluated when the wait operation is executed
  - value of *c (priority number*) stored with the name of the process that is suspended
  - when ***x.signal*** is executed, process with smallest associated priority number is resumed next.
- Check tow conditions to establish correctness of system
  - User processes must always make their calls on the monitor in a correct sequence
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols

REFERENCES

- Stallings W. **Sistemi Operativi** Jackson Libri
- Silberschatz A. **Sistemi Operativi** Addison-Wesley
- Tanenbaum A. **I Moderni Sistemi Operativi** Jackson Libri
- Cay S. Horstmann. **Concetti di Informatica e fondamenti di JAVA 2** Apogeo
- Web Material (see links)

