So far...

- Seen one method for dealing with mutual exclusion when sharing common resources: busy waiting

This lecture

- How can the mutual exclusion and synchronisation problems be solved with other constructs?
  - Semaphore and monitor
- What do we gain by using the support in OS or programming language?

Summary of early solutions

- Busy waiting solutions to mutual exclusion problems are:
  - Difficult to implement – consider n processes instead of 2 in solution from last lecture!
  - Wasteful for the CPU as a resource
- But...
  - They do not require any support
- Would be useful to have support from an operating system or a programming language

Semaphores

- A semaphore S is a non-negative integer variable on which only two operations `wait` and `signal` can be performed

  ```
  wait(S)
  if S > 0 then S = S-1 else
  suspend the process calling wait(S)

  signal(S)
  if some process P has been suspended by an earlier wait(S), then wake up P
  else S = S+1
  ```

Recall: two general problems

- Mutual exclusion

  ```
  Process Pi
  {
  entry-protocol
  critical-section
  exit-protocol
  non-critical-section
  }
  ```

- Condition synchronisation
  - Ex: Compute the interest when all transactions have been processed
Using a semaphore ...

- Let’s use a semaphore with the following implementation for solving mutual exclusion

```plaintext
wait(S)
  if S > 0 then S = S-1 else
    suspend the process calling wait(S)

signal(S)
  if some process P has been suspended by an earlier wait(S), then wake up P
  else S = S+1
```

Mutual exclusion

```plaintext
var mex: semaphore;
  (* initially 1 *)
process Pi;
  loop
    wait(mex);
    critical_section;
    signal(mex);
    non_critical_section;
  end
end Pi;
```

Food for thought

Questions:
- How does the program behave with initial value 0 for the semaphore?
- If there are two processes, what is the maximum value of mex?
- If there are n processes?

Counting semaphores

- When more than one instance of a resource is available, e.g. print servers
- Processes can use up to max available but no more
- The semaphore can be initialised to provide access for n processes

Recall: two general problems

- Mutual exclusion
  Process Pi
  {
    entry-protocol
    critical-section
    exit-protocol
    non-critical-section
  }
- Condition synchronisation
  - Ex: Compute the interest when all transactions have been processed

Condition synchronisation

```plaintext
var condition: semaphore;
  (* initially 0 *)
process P1; (* waiting process *)
  ...
  wait(condition)
  ...
end P1;
process P2; (* signalling process *)
  ...
  signal(condition)
  ...
end P2;
```
Semaphore implementations

- Wait and Signal are implemented as atomic operations
- Semaphore variable is always initialised as non-negative

But
- Which process to wake up among all suspended ones is not specified

Atomicity: alternatives

- For programmer wait and signal are atomic (indivisible) operations
- In the supporting environment, atomicity has to be assured by:
  - allowing only one wait/signal to operate on a semaphore at any one time e.g. by
    - disabling interrupts during execution
    - busy waiting on entry to semaphore
  - hardware support (e.g. test-and-set operations) for locking before wait/signal

Spin locks

- The original definitions of wait & signal just used a version of busy waiting to implement the semaphore:
  
  wait(s): while s ≤ 0 do nothing;
  s = s - 1

  signal(s): s = s + 1

Semaphore implementations

- Wait and Signal are implemented as atomic operations
- Semaphore is always initialised as non-negative

But
- Which process to wake up among all suspended ones is not specified

Different implementations

<table>
<thead>
<tr>
<th>process P1;</th>
<th>process P2;</th>
<th>process P3;</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>wait(s)</td>
<td>wait(s)</td>
<td>wait(s)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>signal(s)</td>
<td>signal(s)</td>
<td>signal(s)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>end;</td>
<td>end;</td>
<td>end;</td>
</tr>
</tbody>
</table>

One queue/semaphore at run-time:

Semaphores vs. Busy waiting

- For long critical sections, semaphores more efficient in using CPU
- Better code organisation, less errors?

- What about problems with deadlock and starvation?
- We will come back to these ...
Example

process A
while (true) {
  print(A)
  print(K)
}

process B
while (true) {
  print(T)
  print(C)
}

Insert semaphores so that a sequence of “TACK”s is printed out.

Student solution

var S: semaphore (* initially 1 *)
var Z: semaphore (* initially 1 *)

process A
while (true) {
  wait(S);
  print(A);
  signal(S);
  wait(Z);
  print(K);
  signal(Z);
}

process B
while (true) {
  wait(S);
  print(T);
  signal(S);
  wait(Z);
  print(C);
  signal(Z);
}

What is a monitor?

• A programming abstraction consisting of:
  – A data structure on which programmer can
    define operations – which can only be run
    one at a time
  – Condition variables for synchronisation
• Encapsulates shared data that several
  processes can operate upon
• All access is with mutual exclusion
• Pre object-orientation!

[Hoare 74]

Overview

Shared data
Condition variables X, Y
Operations on
Shared data
Initialisation code

Condition variables

• Declared as special synchronisation variables:
  Condition X;
• With two designated operations:
  wait(X): suspend the calling process
  signal(X): if there are suspended
  processes on this variable, wake one up

Example: Bounded buffer

(* in some language that supports monitors *)

monitor BoundedBuffer;
Buf: array [0..SizeOfBuffer] of integer;
Base, Top, Count: integer;
NotFull, NotEmpty: condition;
operation Append(E: integer);
...
end Append;
operation Take(var E: integer);
...
end Take;
begin
<initialize> (* set Base,Top,Count to 0 *)
end BoundedBuffer;
Operation Append

operation Append (E: integer);
begin
  if Count == SizeOfBuffer + 1 then
    wait(NotFull);
  Buff[Top] = E;
  Top = (Top + 1) mod SizeOfBuffer;
  Count = Count + 1;
  signal(NotEmpty)
end Append;

Operation Take

operation Take (var E: integer);
begin
  if Count == 0 then
    wait(NotEmpty);
  E = Buff[Base];
  Base = (Base + 1) mod SizeOfBuffer;
  Count = Count - 1;
  signal(NotFull)
end Take;

Producer-Consumer problem

process Producer;
var Element: integer;
begin
  loop
    Produce(Element);
    Append(Element)
  end Producer;

process Consumer;
var Element: integer;
begin
  loop
    Take(Element);
    Consume(Element)
  end Consumer;

Observations

• Programmer uses `wait` and `signal` inside the code that defines the operations on the shared data structure

Note:
• The condition variable has no values assigned to it
• The queue associated with each variable is the main synchronisation mechanism
• Different semantics from semaphore operations for `wait` and `signal`

Process queues

Queue of processes wanting to execute some monitor operation

These operations may use `wait/signal` on X, Y

Operations on Shared data

Condition variables X, Y

Initialisation code

How does it work?

Process P1
  ... wait(X)

Process P4
  ... signal(X)

Process P2

Which process to run at time t?
Options

- Original Hoare monitor: let the woken up process (P1) continue

  What if there are several processes waiting on X?

- Pragmatic solution (Java): let the signalling process continue, and wake up P1 once P4 is suspended/exits

  P1 has to check for condition X when woken up!

Summary

- Monitors have the same power as semaphores but are at a higher level of abstraction

  Exercise: Try implementing producer-consumer solution with semaphores!

- Monitor has two different mechanisms for handling synchronisation and for data communication

- Mutually exclusive access to data automatic, but matching waits and signals still a problem!

Questions?