Interprocess Synchronization and Communication
Problem with Semaphores

• Like fork/join/quit, semaphores are too low level
• Combine blocking with counting
  • Really two separate operations, and should be treated as such
• Hard to debug
  • Easy to make mistakes
  • Think of typing wait when you meant to type signal
  • Original name for wait (P), signal (V) even easier to mistype
    • P from the Dutch passing ("passing")
    • V from the Dutch verhogen ("increase")
    • Taken from railroad signals
Alternate Approach

• Key idea: data abstraction
• Think about classes in object-oriented programming
• Classes define abstract data types and the functions that can access them
  • *Must* access the data structures by calling functions in the class
Monitors

• Implement classes, but *guarantee* mutual exclusion so at most 1 process can be active in the monitor (class)

• Access to the encapsulated resource (abstract data type) should be possible *only* through the monitor

• Procedures in the monitor are mutually exclusive
  
  • When 1 process is executing within the monitor, other processes calling procedures within monitor are delayed until the process currently in monitor leaves the monitor
Synchronization

• Define a *condition variable* with 2 operations:
  
  • `x.wait`: block process; it goes onto a queue associated with the condition variable `x`
  
  • `x.signal`: if any process is blocked on condition variable `x`, unblock one of them; if not, this is ignored

• Difference between these and semaphores is these do *not* maintain signal (ie, are memoryless)
  
  • If `signal(sem)` given and no process blocked on `sem`, the next process to encounter a `wait(sem)` does not block
  
  • If `x.signal` given and no process blocked on `x`, the next process to encounter an `x.wait` will block
Problem with *signal*

- Process 1 blocked on `x.wait`
- Process 2 executes `x.signal`
- Which process proceeds?
  - Only 1 process can be active in the monitor at a time
- Does process 1 wait for process 2 to leave the monitor, or *vice versa*?
Process 1 Continues

• C. A. R. Hoare’s approach
• Process 2 waits until process 1 blocks on a `wait` or leaves the monitor
• Process 2 has priority over processes waiting to enter the monitor
• Leads to simpler, more elegant proofs of solutions to problems
Process 2 Continues

• Lampson and Redell’s approach; used in programming language Mesa
• Idea is that Hoare’s approach may lead to the “logical” condition that process 1 blocked on being false by the time process 2 leaves the monitor
• Under this scheme, the monitor must say

  while not B do x.wait;

  rather than

  if not B do x.wait;
Example: Binary Semaphores

• A binary semaphore is 0 or 1 (false or true)

• \textit{signal(bsem)} sets binary semaphore \textit{bsem} to 1 (true)

• To implement this with monitors, define the condition variable \textit{notbusy} on which blocked processes will wait

• Boolean variable \textit{busy} says whether binary semaphore is set (true, 1) or not (false, 0)

• Initially the caller of \textit{wait} passes it; then subsequent ones block, until a signal releases one
Example: Binary Semaphores

binary_semaphore: monitor;
  var  busy: boolean;
  notbusy: condition

(* wait *)
procedure entry wait;
begin
  if busy then
    notbusy.wait;
  busy := true;
end;
Example: Binary Semaphores

procedure entry signal;
    begin
        busy := false;
        notbusy.signal;
    end;
begin
    begin
        busy := false;
    end.

Example Use

Process 1:

```c
... bsem.wait;
(* critical section *)
bsem.signal;
...```

Process 2:

```c
... bsem.wait;
(* critical section *)
bsem.signal;
...```
Producer-Consumer Solution with Monitors

buffer: monitor

var array slots[0..n-1] of item;
  count, in, out: integer;
  notempty, notfull: condition;
Producer-Consumer Solution with Monitors

**procedure** deposit(data: item)
**begin**

  if count = n then
    notfull.wait;
  slots[in] := data;
  in := in + 1 mod n;
  count := count + 1;
  notempty.signal;

**end;**
Producer-Consumer Solution with Monitors

**procedure** extract**(var data: item)**

*begin*

*if* count = 0 *then*

  notempty.*wait*;

  data := slots[out];

  out := out + 1 *mod* n;

  count := count - 1;

  notfull.*signal*;

*end*;
Producer-Consumer Solution with Monitors

begin
    count := 0;
    in := 0;
    out := 0;
end.
Analysis

Producer:
• If buffer full, block on notfull
• Otherwise (or after), deposit data, add 1 to number in buffer, increment index so next deposit goes into next slot
• If any process is blocked on notempty, unblock it

Consumer:
• If buffer empty, block on notempty
• Otherwise (or after), extract data, subtract 1 from number in buffer, decrement index so next extraction is from next slot
• If any process is blocked on notfull, unblock it
First Readers-Writers Problem Solution

readerwriter: monitor;
  var readcount: integer;
  writing: boolean;
  oktored, oktowrite: condition;
procedure beginread
begin
    readcount := readcount + 1;
    if writing then
        oktoread.wait;
end;

procedure endread
begin
    readcount := readcount - 1;
    if readcount = 0 then
        oktowrite.signal;
end;
procedure beginwrite
begin
  if readcount > 0 or writing then oktowrited.wait;
  writing := true;
end;

procedure endwrite
begin
  var i: integer;
  writing := false;
  if readcount > 0 then
    for i := 1 to readcount do oktoread.signal;
  else
    oktowrite.signal;
end;
First Readers-Writers Problem Solution

begin
  readcount := 0;
  writing := false;
end.
Analysis

Readers on entry:
• Add in another reader
• Block on condition oktoread if there is a writer
• Otherwise, or when unblocked, go in

Readers on exit:
• Subtract a reader as it is exiting critical section
• If no more readers, signal any waiting writer that it can go in
Analysis

Writers on entry:
• If any process (reader or writer) in critical section, block on condition oktowrite
• Otherwise, or when unblocked, set writing to true to indicate a writer is entering

Writers on exit:
• Set writing to false to indicate writer is leaving critical section
• Unblock any readers that are waiting on condition oktoread
• If none waiting, unblock a writer if any are waiting
Implementing Monitors with Semaphores

- Operating system has semaphores
- Programming language/environment implements monitors
- Compiler must translate monitors into semaphores
- In this version, processes that signal and as a result block are to be restarted before any process waiting to enter the monitor
  - Processes signaling block on semaphore urgent
  - Processes entering block on semaphore mutex
- Monitor condition variable $x$ represented by semaphore $xcond$
Variables

mutex, urgent, xcond: semaphore;
urgentcount, xcondcount: integer;
Monitor Procedure

• Each procedure in the monitor set up like this:

```c
mutex.wait;
(* procedure body *)
if urgentcount > 0 then
    urgent.signal;
else
    mutex.signal;
```
Monitor Waits

• Replace each x.wait with:

```c
xcondcount := xcondcount + 1;
if urgentcount > 0 then
    urgent.signal;
else
    mutex.signal;
Xcond.wait;
xcondcount := xcondcount - 1;
```
Monitor Signals

• Replace each \texttt{x.signal} with:

\begin{verbatim}
urgentcount := urgentcount + 1;
if xcondcount > 0 then begin
   xcond.signal;
   urgent.wait;
end
urgentcount := urgentcount - 1;
\end{verbatim}