

Producer-Consumer problem

```
circular_pool: monitor  
begin  
    pool: array 0..N-1 of buffer;  
    count, head, tail: int;  
    nonempty, nonfull: condition;
```

```
Procedure extract(x)  
begin  
    if count = 0 then nonempty.wait;  
    x := pool[tail] ;  
    tail := tail + 1 mod N;  
    count := count - 1;  
    nonfull.signal  
end
```

```
Procedure insert(x)  
begin  
    if count = N then nonfull.wait;  
    pool[head] := x;  
    head := head + 1 mod N;  
    count := count + 1;  
    nonempty.signal  
end
```

```
    count := 0  
    head := 0; tail := 0;  
end circular_pool
```

Readers-Writers with concurrent reader access

```
procedure startRead
begin
    readers = readers+1;
end
<READ FILE>
```

```
procedure endRead
begin
    readers = readers -1;
    if (readers == 0) then
        writer.signal;
    end
end
```

```
procedure writer
begin
    if (readers >0) then
        writer.wait;
    <WRITE FILE>
end
```

- This solution works, but does not guarantee readers priority
 - hint: who is allowed into the monitor when a writer exits?

Readers-Writers solution with readers' priority

```
procedure startRead;  
begin  
  if busy then OKtoread.wait;  
  readcount := readcount + 1;  
  OKtoread.signal;  
end startRead;
```

```
procedure endRead;  
begin  
  readcount := readcount - 1;  
  if readcount = 0 then OKtowrite.signal;  
end endRead;
```

```
procedure startWrite;  
begin  
  if busy OR readcount ≠ 0  
  then OKtowrite.wait;  
  busy := true;  
end startWrite;
```

```
procedure endWrite;  
begin  
  busy := false;  
  if OKtoread.queue  
  then OKtoread.signal  
  else OKtowrite.signal;  
end endWrite;
```

wait with priority

- An enhanced version of the **wait** operation accepts an optional priority argument:
 - syntax: *<variable>.wait <parameter>*
 - the smaller the value of the parameter, the highest the priority
- When the variable is signaled, the process with highest priority in the queue is activated
 - the base wait implementation used a First-In-First-Out (FIFO) discipline

Example: Smallest job first

```
procedure startPrint;  
begin  
  if NOT printerIsBusy  
    then jobAvailable.wait;  
  printer-file := buffer;  
end startPrint;
```

<print printer-file>

```
procedure endPrint;  
begin  
  printerIsBusy := false;  
  OKtoprint.signal;  
end endPrint;
```

```
procedure enqueueJob(file);  
begin  
  if printerIsBusy  
    then OKtoprint.wait sizeof(file);  
  printerIsBusy := true;  
  buffer := file;  
  jobAvailable.signal  
end;
```

Monitors: pros and cons

- Pros:
 - encapsulation provides automatic serialization
 - flexibility in blocking and unblocking process execution within monitor procedures
- Cons
 - lack of concurrency if monitor encapsulates shared resources
 - possibility of deadlock with nested monitor calls

Lessons learned

- Encapsulation of critical section of code is desirable
 - provides automatic mutual exclusion
 - single copy of code, single point of synchronization
 - however would be nice to have some form of controlled concurrency
- Blocking/unblocking of processes is powerful tool
 - basic ingredient are named queues, enqueue and dequeue operations
 - enqueue and dequeue operations usually subject to condition

Other existing mechanisms to handle concurrency

- Path Expressions
 - abstraction designed to describe the list of all possible legal executions of operations on shared resource
- Communicating Sequential Processes (CSP)
 - the exchange of messages as synchronization points between sequential processes
- ADA tasks
 - language constructs for the message passing
- One thing in common
 - None in practical use currently (though ADA was a popular language in 80s and 90s)

Multi-threaded programming in Java

- Java allows program to specify multiple threads of execution
- Provides instructions to ensure mutual exclusion, and selective blocking/unblocking of threads

What is a thread in Java ?

- A thread is a program-counter and a stack
- All threads share the same memory space
- A running thread can
 - Yield
 - Sleep
 - Wait for I/O or notification
 - Be pre-empted
- A key feature: Synchronized methods
 - Allow an exclusive lock, e.g., in an update method

Basic Syntax

- Build a thread by extending the class `java.lang.Thread`
- Must have a `public void run()` method – it is executed at the start of the thread, and when it finishes, the thread finishes
- Synchronized statement
 - `Synchronized (obj) { block }`
 - Obtains a lock on obj before executing block, releases lock after executing block
- `Wait()` gives up lock and suspends the thread
- `Notifyall()` resumes all threads waiting on object, resumed tasks must reacquire lock before continuing

Producer Consumer Example

```
Public class ProdCons {  
    private boolean ready ;  
    private Object obj ;  
    public ProducerConsumer() {  
        ready = false ;  
    }  
    public ProducerConsumer (Object o) {  
        obj = o ;  
        ready = true ;  
    }  
}
```

```
Synchronized Object consume() {  
    while (!ready) wait() ;  
    ready = false ;  
    notifyAll() ;  
    return obj ;  
}  
Synchronized void produce (object o)  
{  
    while (ready) wait() ;  
    obj = o ;  
    ready = true ;  
    notifyAll() ;  
}  
}
```