Communicating Sequential Processes

Syntax

Syntax

- Destination!port(e1, ..., en);

Syntax

- Destination!port(e1, ..., en);

- Source?port(x1, ..., xn);

Example: Copy

Example: Copy

Example: Copy


```
process Copy {
```

Example: Copy


```
process Copy {
  char c;
```

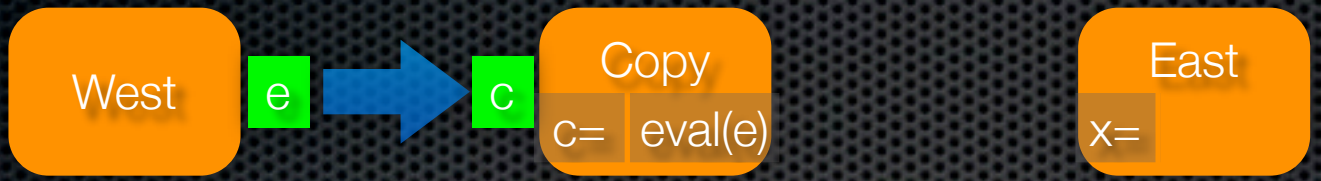
Example: Copy


```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
```

Example: Copy

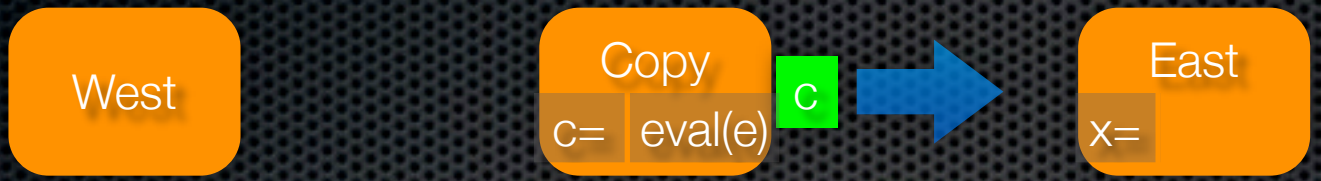

```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
```

Example: Copy



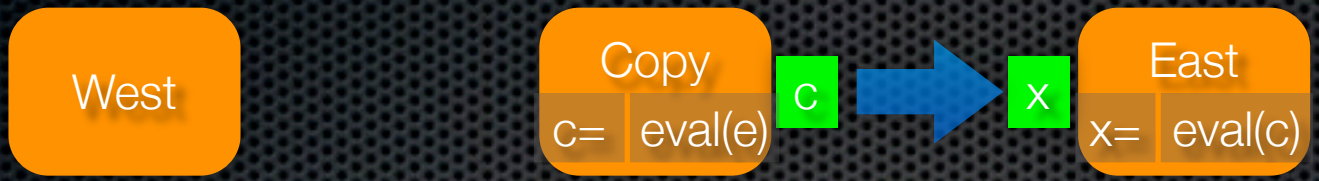

```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
```

Example: Copy



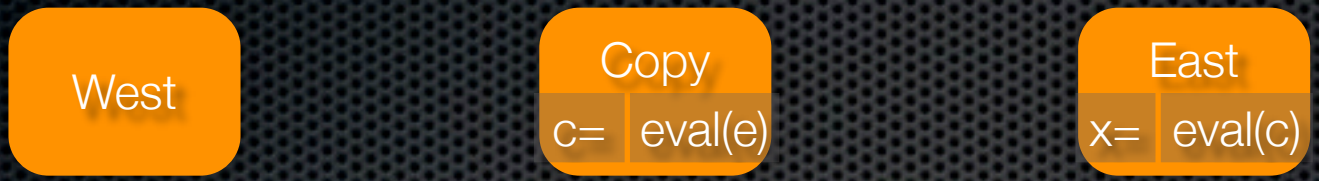

```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
      East!c;   # output char to East
```

Example: Copy



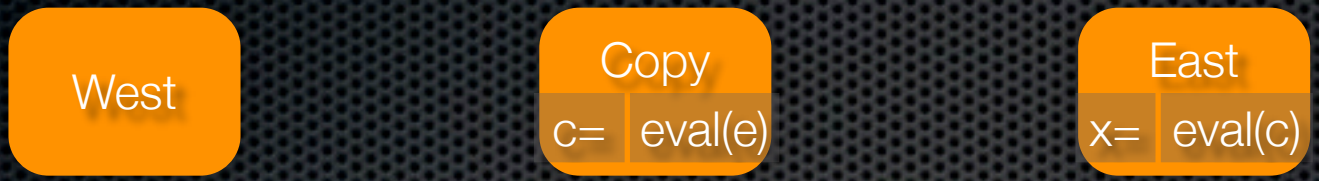

```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
      East!c;   # output char to East
```

Example: Copy




```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
      East!c;   # output char to East
   od
```

Example: Copy




```
process Copy {
   char c;
   do true ->
      West?c; # input char from West
      East!c;   # output char to East
   od
}
```

Example: GCD

```
process GCD {
    int id, x, y;
    do true ->
        Client[*] ? args(id, x, y);    # input a "call"
        # repeat the following until x == y
        do x > y -> x = x - y;
        [ ] x < y -> y = y - x;
        od
        Client[id] ! result(x); # return the result
    od
}
```

Example: GCD

```
process GCD {
    int id, x, y;
    do true ->
        Client[*] ? args(id, x, y);    # input a "call"
        # repeat the following until x == y
        do x > y -> x = x - y;
        [ ] x < y -> y = y - x;
        od
        Client[id] ! result(x); # return the result
    od
}
```

Example: GCD

```
process GCD {
    int id, x, y;
    do true ->
        Client[*] ? args(id, x, y);    # input a "call"
        # repeat the following until x == y
        do x > y -> x = x - y;
        [ ] x < y -> y = y - x;
        od
        Client[id] ! result(x); # return the result
    od
}
```

Example: GCD

```
process GCD {
    int id, x, y;
    do true ->
        Client[*] ? args(id, x, y);    # input a "call"
        # repeat the following until x == y
        do x > y -> x = x - y;
        [ ] x < y -> y = y - x;
        od
        Client[id] ! result(x); # return the result
    od
}
```

Destination

Guarded Communication

Guarded Communication

- Syntax: B; C -> S;

Guarded Communication

- Syntax: B; C -> S;
- Example:

```
process Copy {
    char c;
    do West?c -> East!c; od
}
```

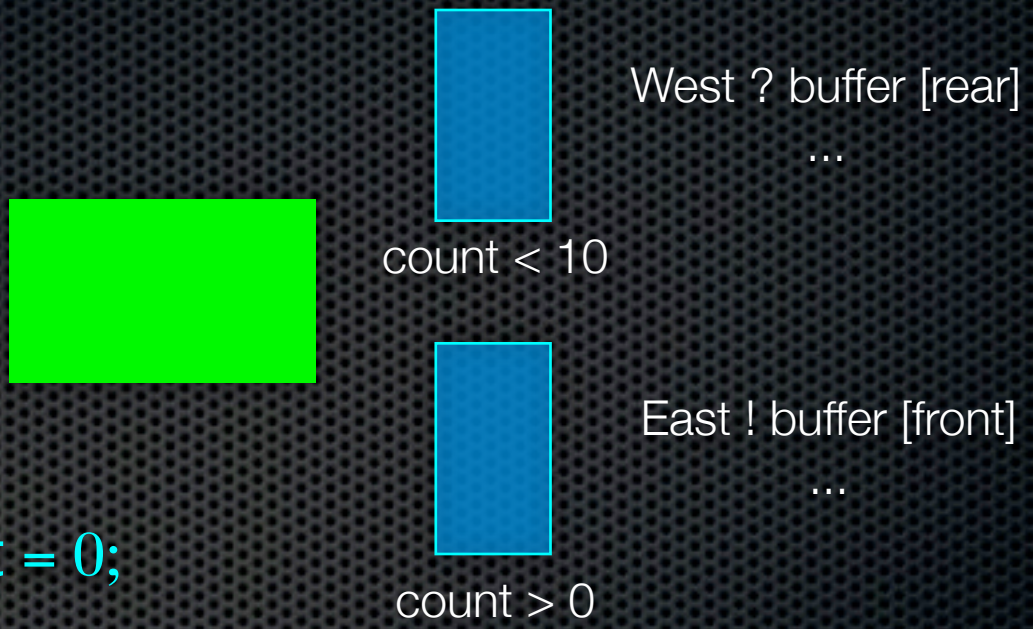
Buffering I

```
process Copy {
   char c1, c2;
   West ? c1;
   do West ? c2 -> East ! c1; c1 = c2;
   [ ] East ! c1 -> West ? c1;
   od
}
```

Buffering II

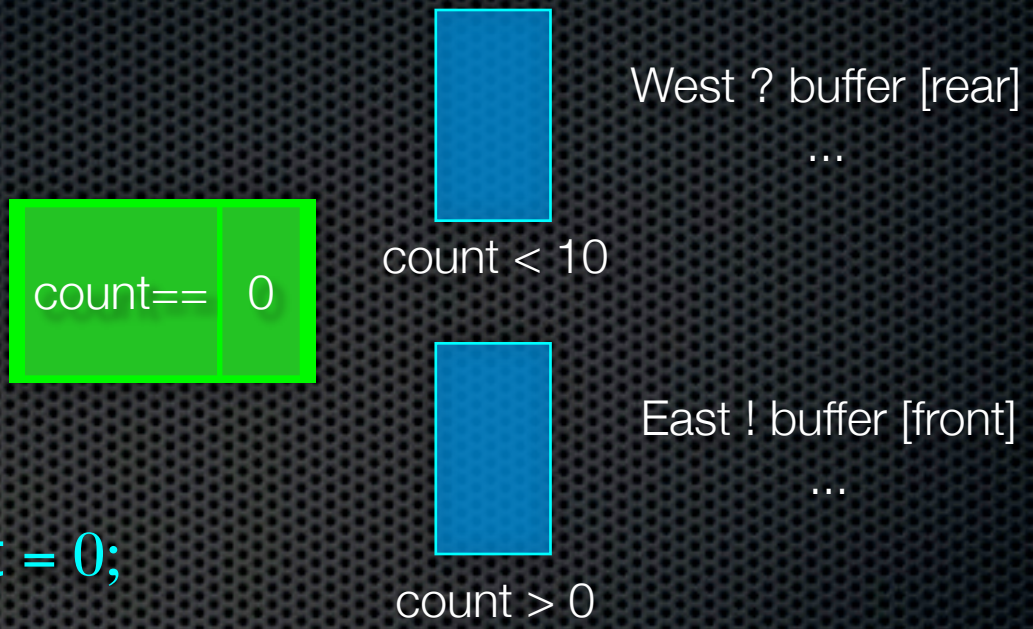
```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



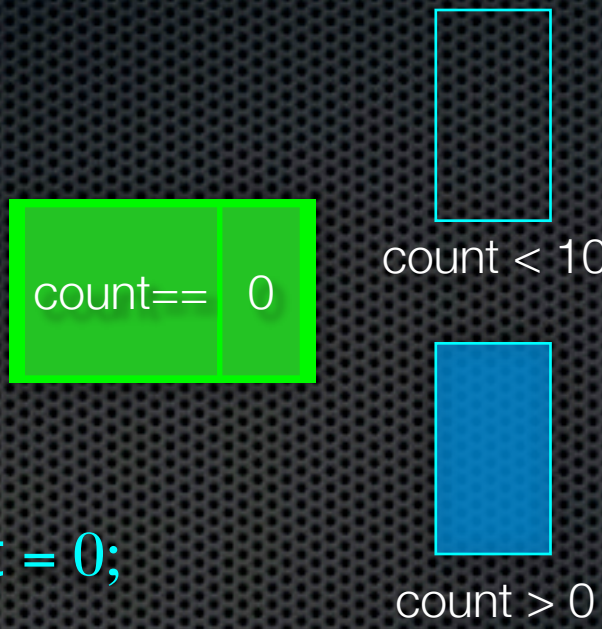

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



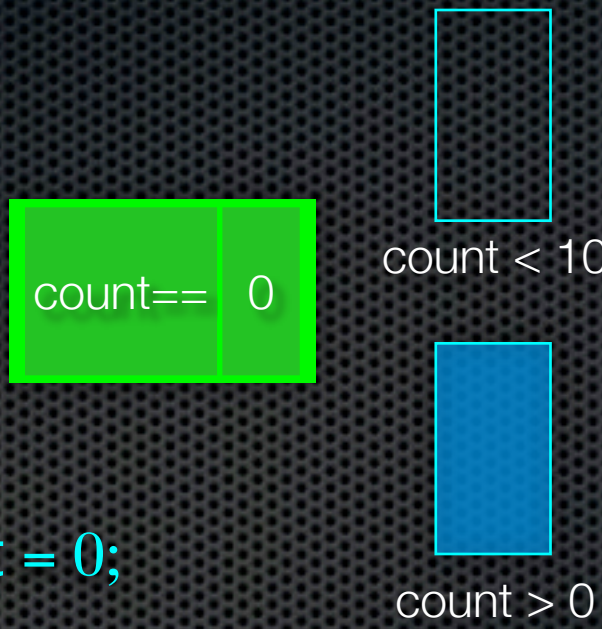

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



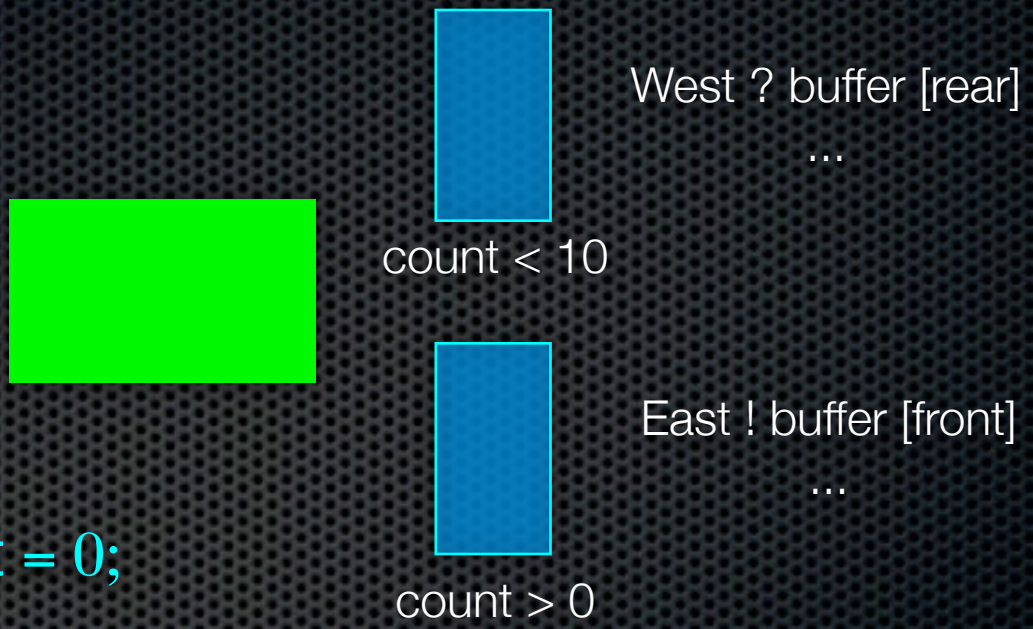

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



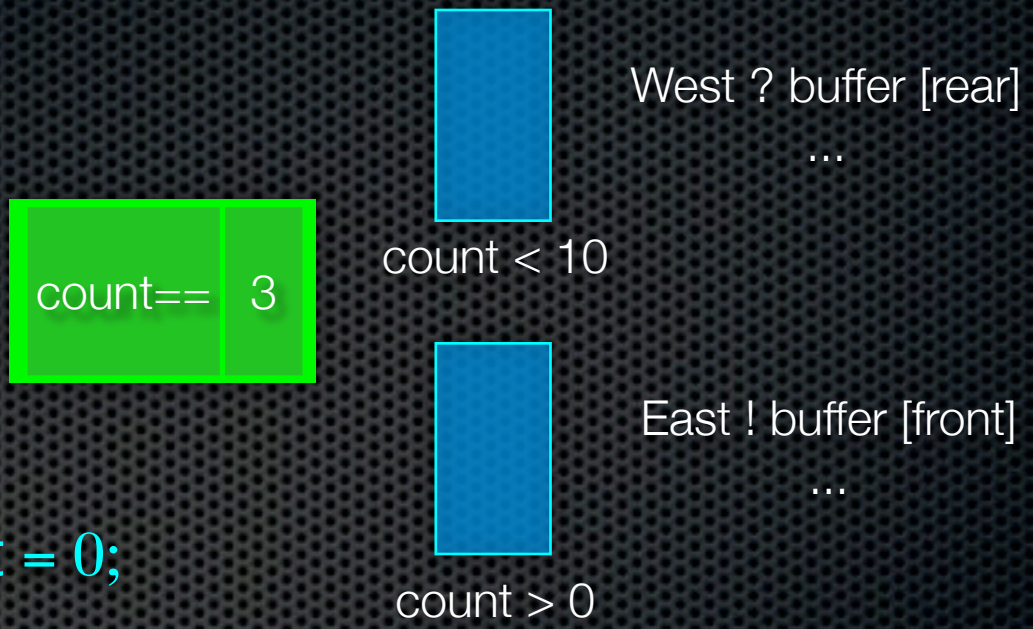

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



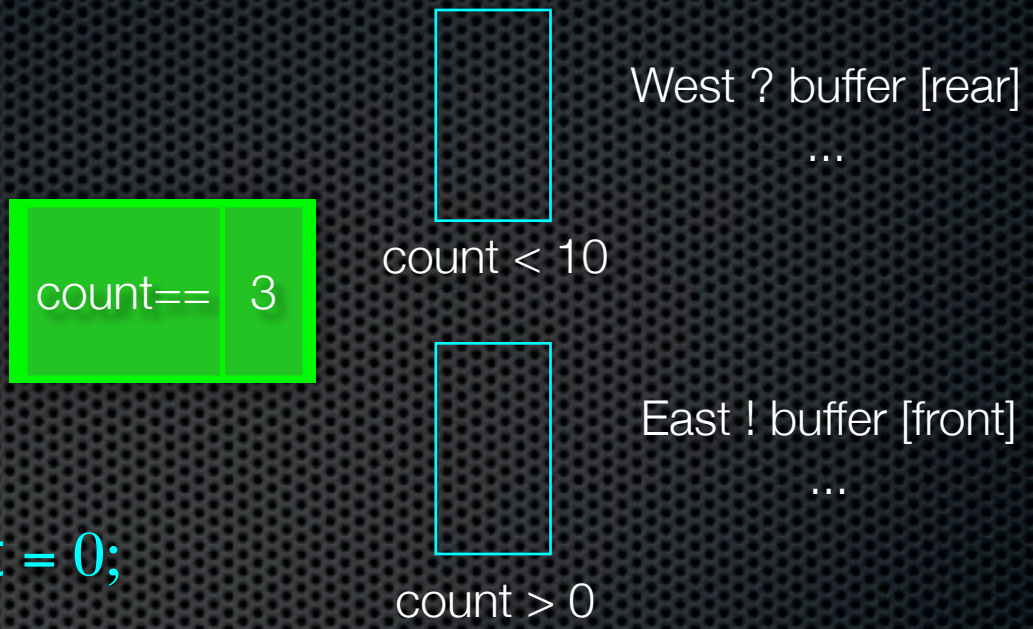

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



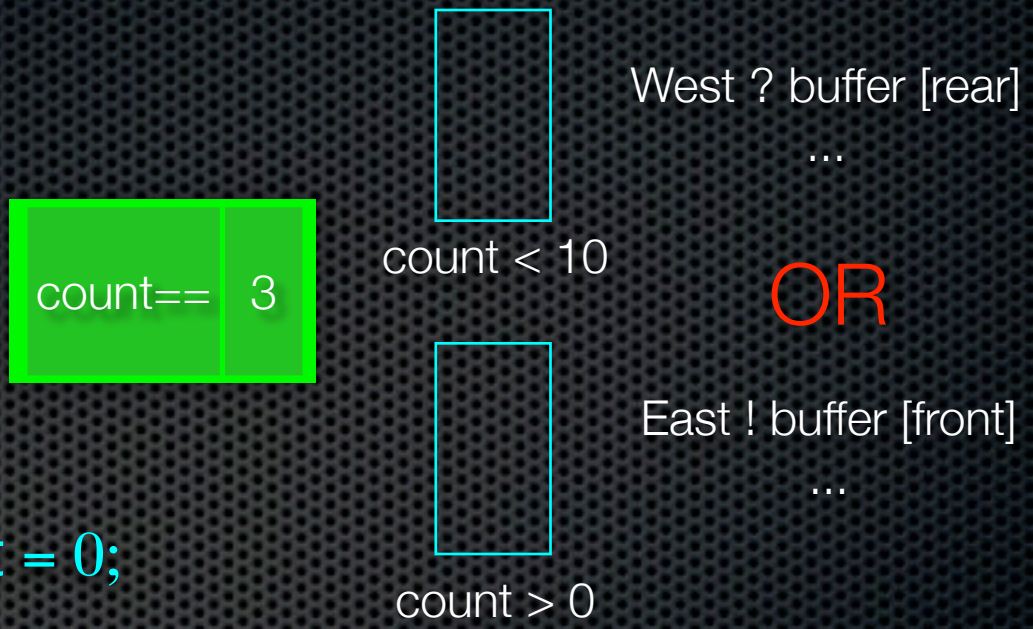

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



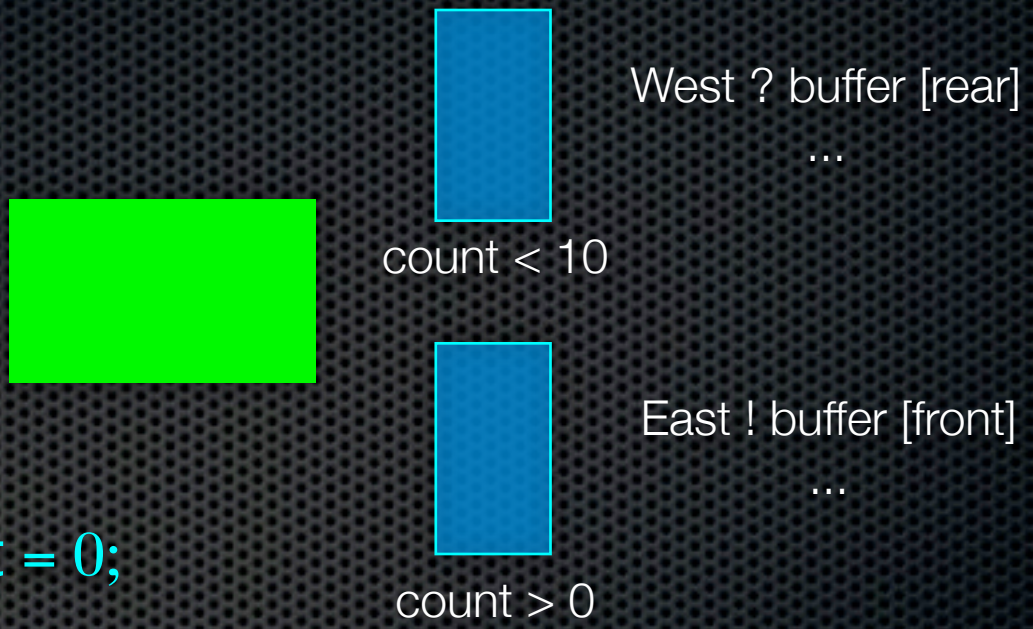

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



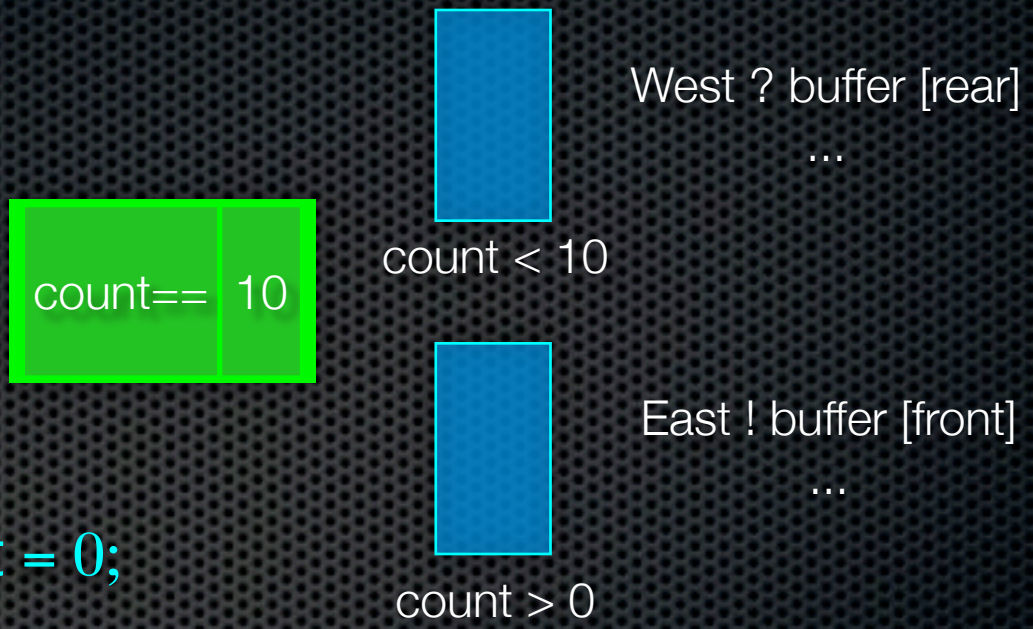

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



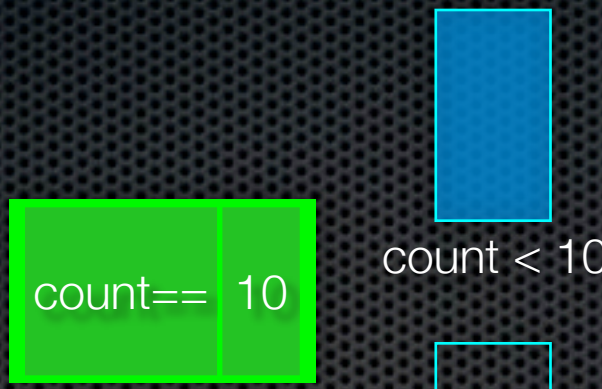

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II




```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



West ? buffer [rear]

...

count < 10

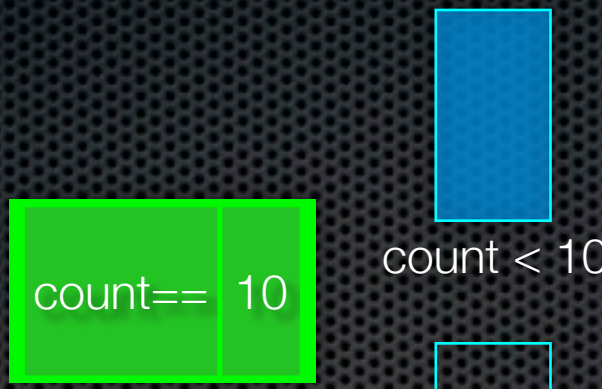
East ! buffer [front]

...

count > 0

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II



West ? buffer [rear]

...

count < 10

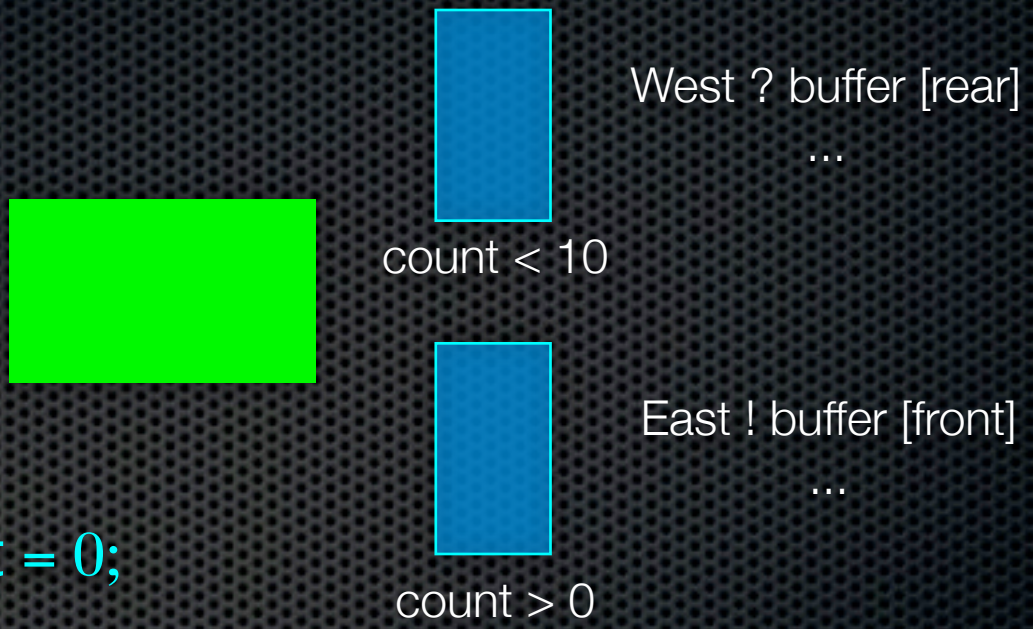
East ! buffer [front]

...

count > 0

```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Buffering II




```
process Copy {
    char buffer[10];
    int front = 0, rear = 0, count = 0;
    do count < 10; West ? buffer[rear] ->
            count = count+1; rear = (rear + 1) mod 10;
    [ ] count > 0; East ! buffer[front] ->
            count = count - 1; front = (front + 1) mod 10;
    od
}
```

Resource Allocator

```
process Allocator {
    int avail = MAXUNITS;
    set units = initial values;
    int index, unitid;
    do avail > 0; Client[*] ? acquire (index) ->
        avail--; remove (units, unitid);
        Client [index] ! reply (unitid);
    [ ] Client[*] ? release (index, unitid) ->
        avail++; insert (units, unitid);
    od
}
```

Sieve of Erastosthenes

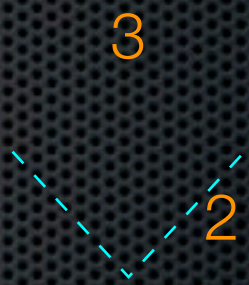
Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;   # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                # p is a prime
  do Sieve[i-1]?next ->        # receive next candidate
    if (next mod p) != 0 ->    # if it might be prime,
      Sieve[i+1]!next;         #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                # p is a prime
  do Sieve[i-1]?next ->        # receive next candidate
    if (next mod p) != 0 ->    # if it might be prime,
      Sieve[i+1]!next;         #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes



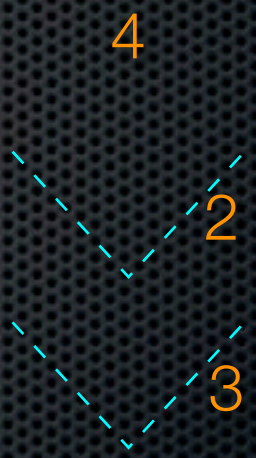

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                # p is a prime
  do Sieve[i-1]?next ->        # receive next candidate
    if (next mod p) != 0 ->    # if it might be prime,
      Sieve[i+1]!next;         #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                # p is a prime
  do Sieve[i-1]?next ->        # receive next candidate
    if (next mod p) != 0 ->    # if it might be prime,
      Sieve[i+1]!next;         #   pass it on
    fi
  od
}
```

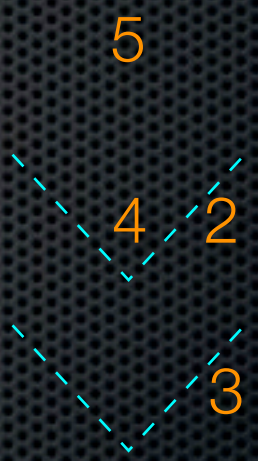


Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                # p is a prime
  do Sieve[i-1]?next ->        # receive next candidate
    if (next mod p) != 0 ->    # if it might be prime,
      Sieve[i+1]!next;         #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```



Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```

Sieve of Erastosthenes

```
process Sieve[1] {
  int p = 2;
  for [i = 3 to n by 2]
    Sieve[2]!i;    # pass odd numbers to Sieve[2]
}
process Sieve[i = 2 to L] {
  int p, next;
  Sieve[i-1]?p;                 # p is a prime
  do Sieve[i-1]?next ->         # receive next candidate
    if (next mod p) != 0 ->     # if it might be prime,
      Sieve[i+1]!next;          #   pass it on
    fi
  od
}
```