Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Producer / Consumer Problem

- Producer generate Items
- Consumer consume item
- What will happen if
  - Rate to producer > rate to consume?
  - Rate to producer < rate to consume?
  - Rate to produce == to Rate to consume?

Producer/Consumer Problem-> Solution

- Two Situation
  - Unbounded Buffer
  - Bounded buffer – limited size
- When buffer is full,
- producer wait until items are consumed.
- Rate of consumption > rate of production, result to empty buffer.
- Producer Consumer problems also known as **Bounded Buffer Problem.**

Producer/Consumer Problem-> Solution

- Solution to this must satisfy the following condition
  - A producer must not overwrite a full buffer
  - Consumer must not consume an empty buffer
  - Consumer and buffer must access buffer in a mutually exclusive manner
- Variable count keep track number of item (N) in buffer
- For Producer
  - If count = n -> buffer is full then producer go to sleep
  - If count ≠ n producer add item and increment count
- For Consumer
  - If count = 0 - > buffer empty, consumer go to sleep
  - If count ≠ 0 consumer remove an item and decrement counter

Bounded-Buffer Problem

- Assume pool consists of n buffers, each capable of holding one object.
- Shared data

  type *item* = …

  var *buffer* = …

  full, empty, mutex: semaphore;

  nextp, nextc: item;

  full :=0; *empty* := *n*; *mutex* :=1;

Bounded-Buffer Problem (Cont.)

- Producer process

repeat

…

produce an item in *nextp*

…

wait(*empty*);

wait(*mutex*);

…

signal(*mutex*);

signal(*full*);

until *false*;

Bounded-Buffer Problem (Cont.)

- 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*;

Readers-Writers Problem

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- Two types of processes by referring to the former as ***readers*** and to the latter as ***writers.***
- Problem:
  - allow multiple readers to read at the same time.
  - Only one single writer can access the shared data at the same time.

Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0
- Writer process

  wait(wrt);

  …

  writing is performed

  …

  signal(wrt);

Readers-Writers Problem (Cont.)

- Reader process

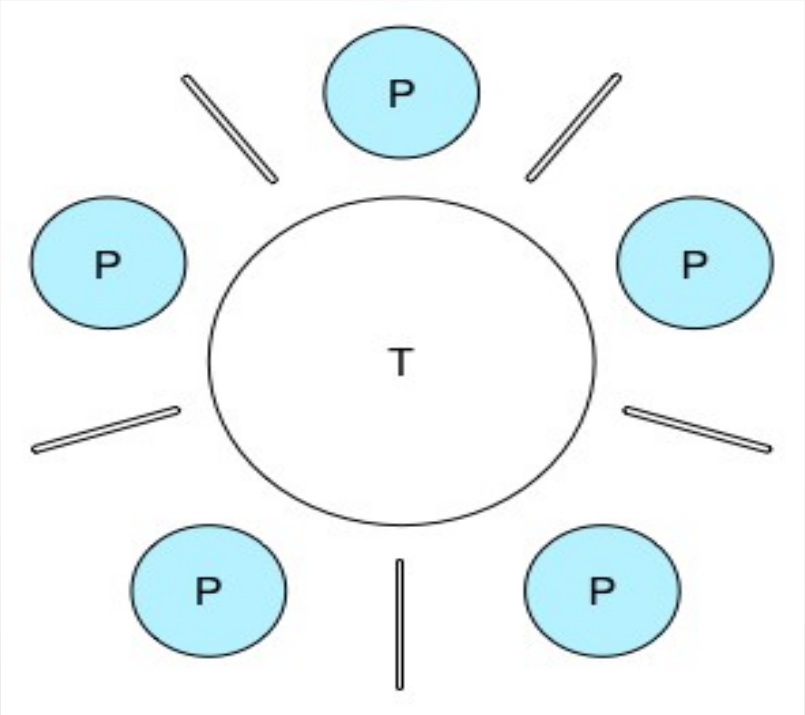
wait(*mutex*);
readcount := *readcount* +1;
if *readcount* = 1 **then** *wait*(*wrt*);
signal(*mutex*);
…
reading is performed
…
wait(*mutex*);
readcount := *readcount* – 1;
if *readcount* = 0 **then** *signal*(*wrt*);
signal(*mutex*):

Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time.
- Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
- When she is finished eating, she puts down both of her chopsticks and starts thinking again.

Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
    - Semaphore chopstick [5] (**array** [0..4]) initialized to 1



Dining-Philosophers Problem (Cont.)

- 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 Problem (Cont.)

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could ***create a deadlock***.
- Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
- All the elements of chopstick will now be equal to 0.
- When each philosopher tries to grab her right chopstick, she will be delayed forever.

Dining-Philosophers Problem (Cont.)

- Several possible remedies to the deadlock problem are listed next.
  - Allow at most four philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
  - Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick

Monitors

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- A monitor type is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

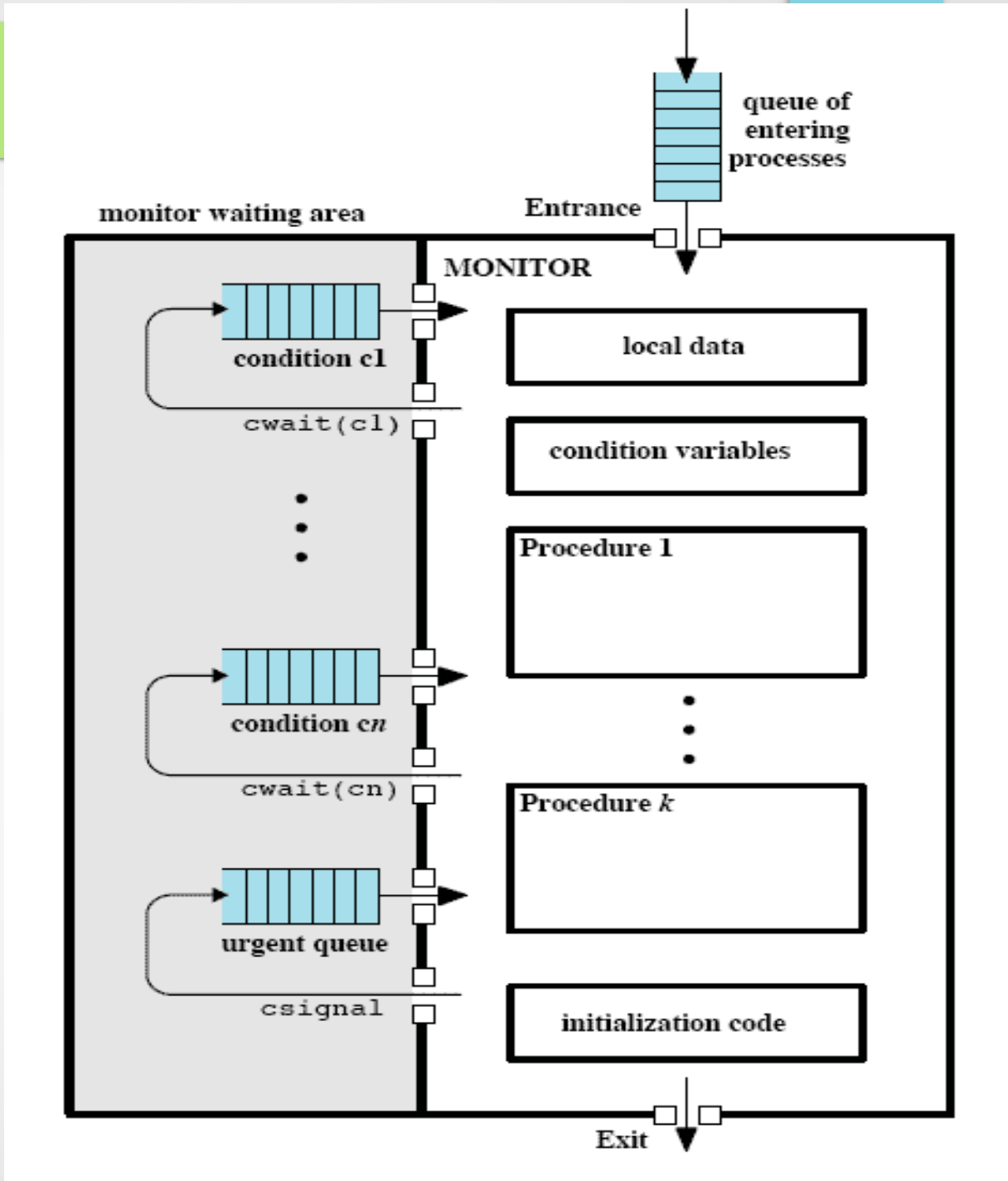
Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, and Java.

Chief characteristics

- Local data variables are accessible only by the monitor.
- Process enters monitor by invoking one of its procedures.
- Only one process may be executing in the monitor at a time.
- Synchronization achieved by condition variables within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - Cwait(c): Suspend execution of the calling process on condition c
  - Csignal(c) Resume execution of some process blocked after a cwait on the same condition

Structure


queue of entering processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Monitors

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

  var *x, y: condition*

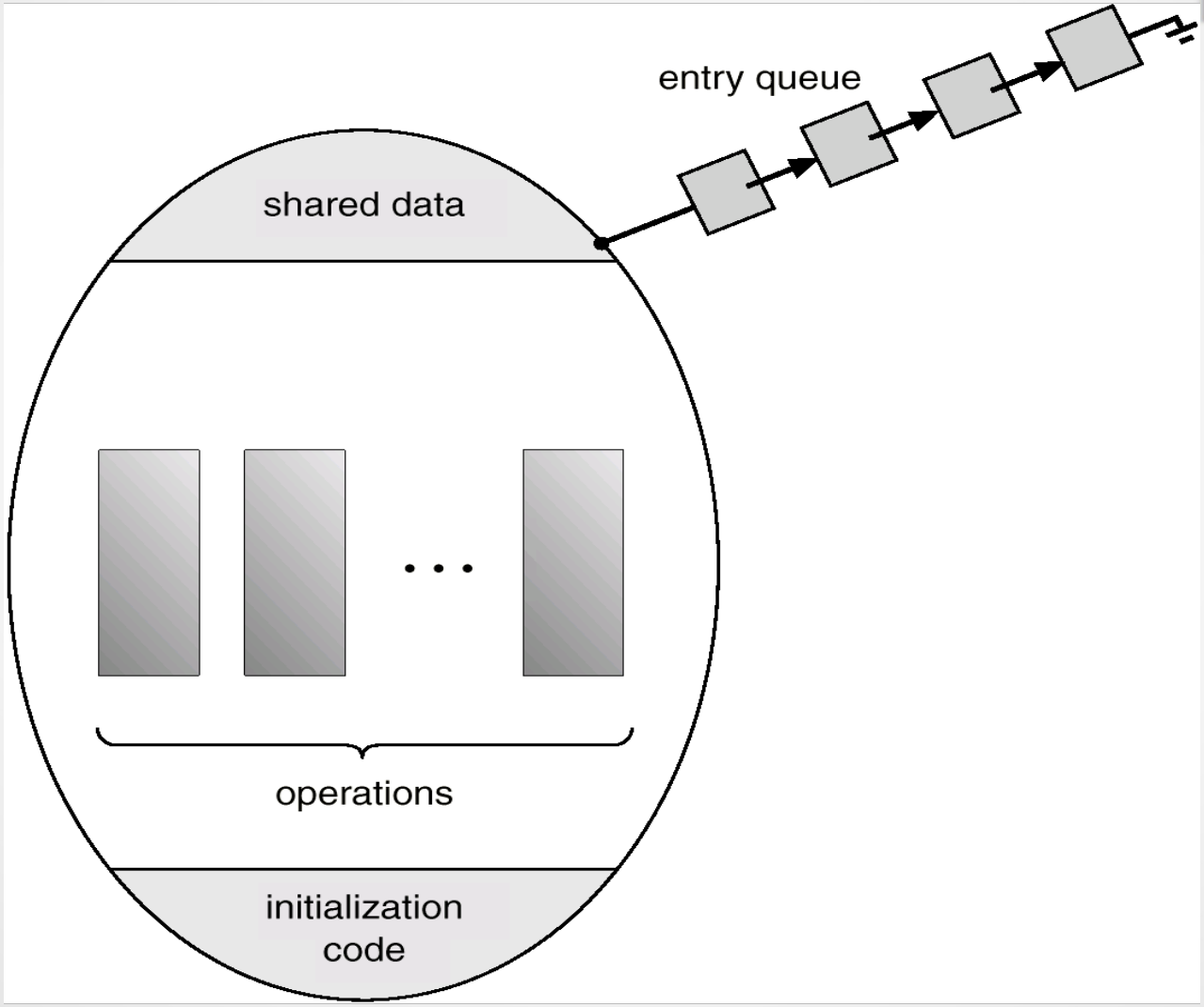
- Condition variable can only be used with the operations *wait* and *signal*.
  - The operation

    x.wait;

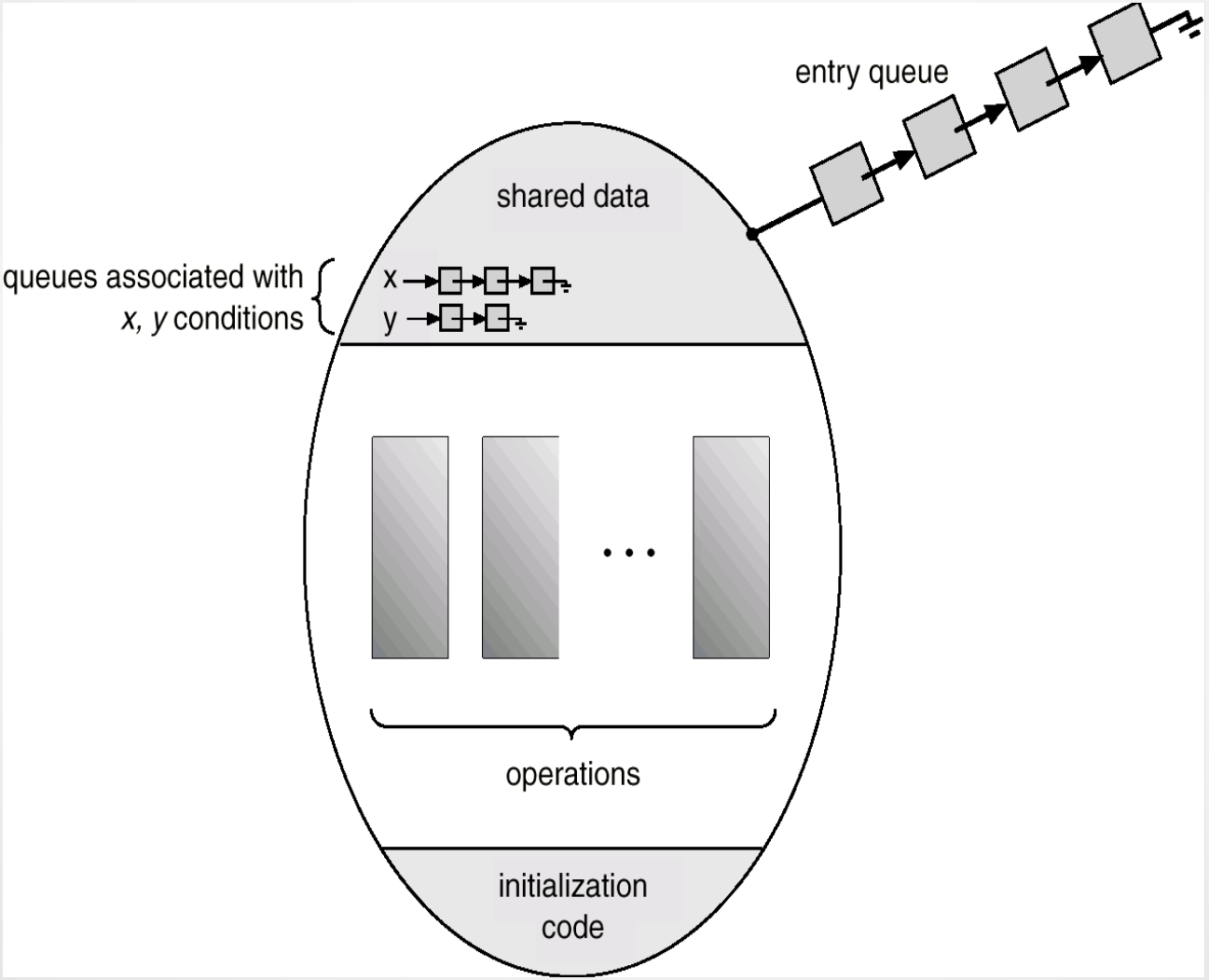
    means that the process invoking this opeation is suspended until another process invokes

    x.signal;
  - The *x.signal* operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Schematic view of a monitor



Monitor with condition variables



Dining Philosophers Example

```
type dining-philosophers = monitor
  var state : array [0..4] of :(thinking, hungry, eating);
  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], wait,
  end;
      procedure entry putdown (i: 0..4);
  begin
  state[i] := thinking;
  test (i+4 mod 5);
  test (i+1 mod 5);
  end;
```

Dining Philosophers (Cont.)

```
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.
```

Dining Philosophers (Cont.)

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }
```

Dining Philosophers (Cont.)

```
void test(int i) {
   if ((state[(i + 4) % 5] != EATING) &&
    (state[i] == HUNGRY) &&
    (state[(i + 1) % 5] != EATING)) {
      state[i] = EATING;
      self[i].signal();
   }
}

initialization_code() {
   for (int i = 0; i < 5; i++)
      state[i] = THINKING;
 }
}
```