Synchronization

Peter J. Denning

CS471/CS571

What is synchronization?

- Requirement that one process stop to wait to pass a point until another process sends a signal.
- The waiting point represents a condition that must be true for subsequent execution to be valid.
- The signal represents the event of the condition becoming true.
- Semaphores directly implement the requirement.

Common Examples

- Process ordering
- Mutual exclusion
- Pool control
- Producer-consumer
- Readers-writers
- Private semaphore and I/O signalling
- Monitors

Process Ordering

- Precedence ordering: one process cannot begin execution until another has finished.
- Terminate the first process with a signal semaphore to the second.

```
P2sem: init c 0

P1: actions
    signal(p2sem)

P2: wait(p2sem)
    actions
```

Mutual Exclusion

- Allow only one of several processes in a critical section at the same time
- Prevent race conditions with shared data processed by the critical section.

```
mutex: init c 1

P1: wait(mutex)
    critical section
    signal(mutex)

P2: wait(mutex)
    critical section
    signal(mutex)
```

Pool Control

- Set of identical resource units
- h = GetUnit( ) -- wait until unit free
- ReturnUnit(h) -- allow waiter to go

```
GetUnit
wait(pool)
...
return h
```

```
ReturnUnit(h)
...
signal(pool)
return
```

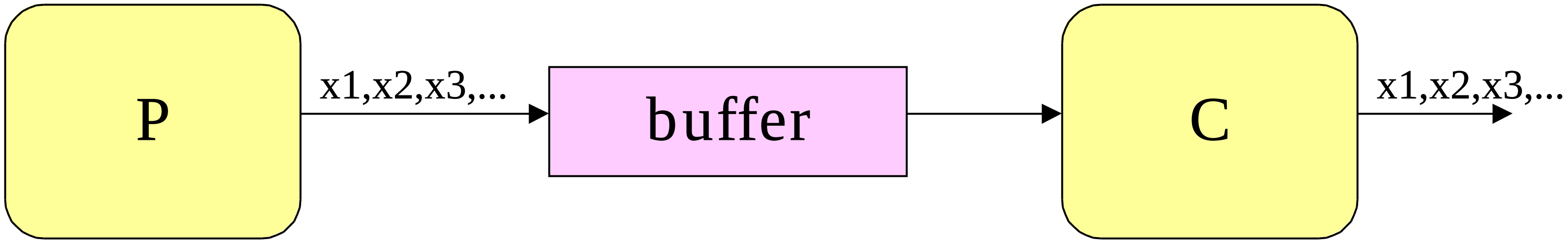
```
I(pool)=N
```

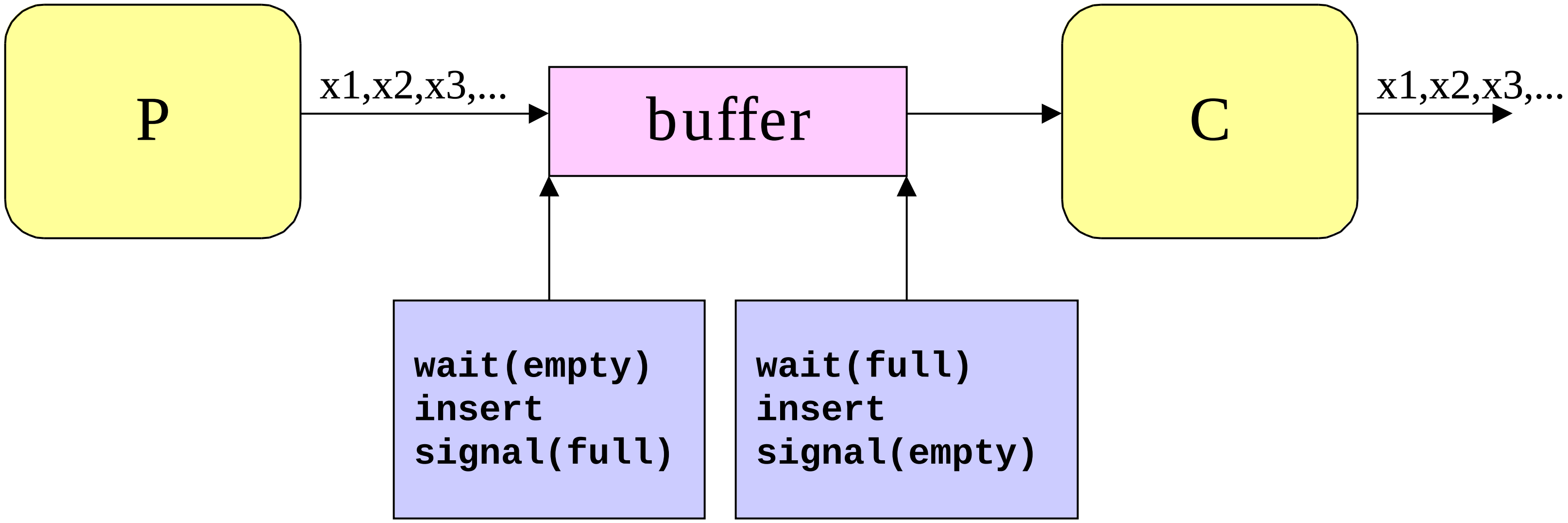
Producer-Consumer

- Process P produces sequence of items x1,x2,x3,....
- Items stored in order in a buffer
- Consumer C consumes items from the buffer in the same order, once each
- Correct operation: output of C identical to output of P (no duplicates, no losses)

Producer-Consumer

- Buffer is bounded, can hold up to N items.
- Stop P when buffer full.
- Stop C when buffer empty.
- Semaphores:
  - empty: counts number of empty buffer slots, initially N
  - full: counts number of full buffer slots, initially 0
- Stop P: wait(empty)
- Stop C: wait(full)
- After insert: P says signal(full)
- After removal: C says signal (empty)


P
x1,x2,x3,...
buffer
C
x1,x2,x3,...



```
wait(empty)
insert
signal(full)
```

```
wait(full)
insert
signal(empty)
```

Readers-Writers

- Shared file
- Multiple readers and writers
- Writers exclude readers and other writers
- Readers exclude writers but not other readers
- Preventing starvation under load
  - priority to readers?
  - priority to writers?
  - alternating?

Dining Philosophers

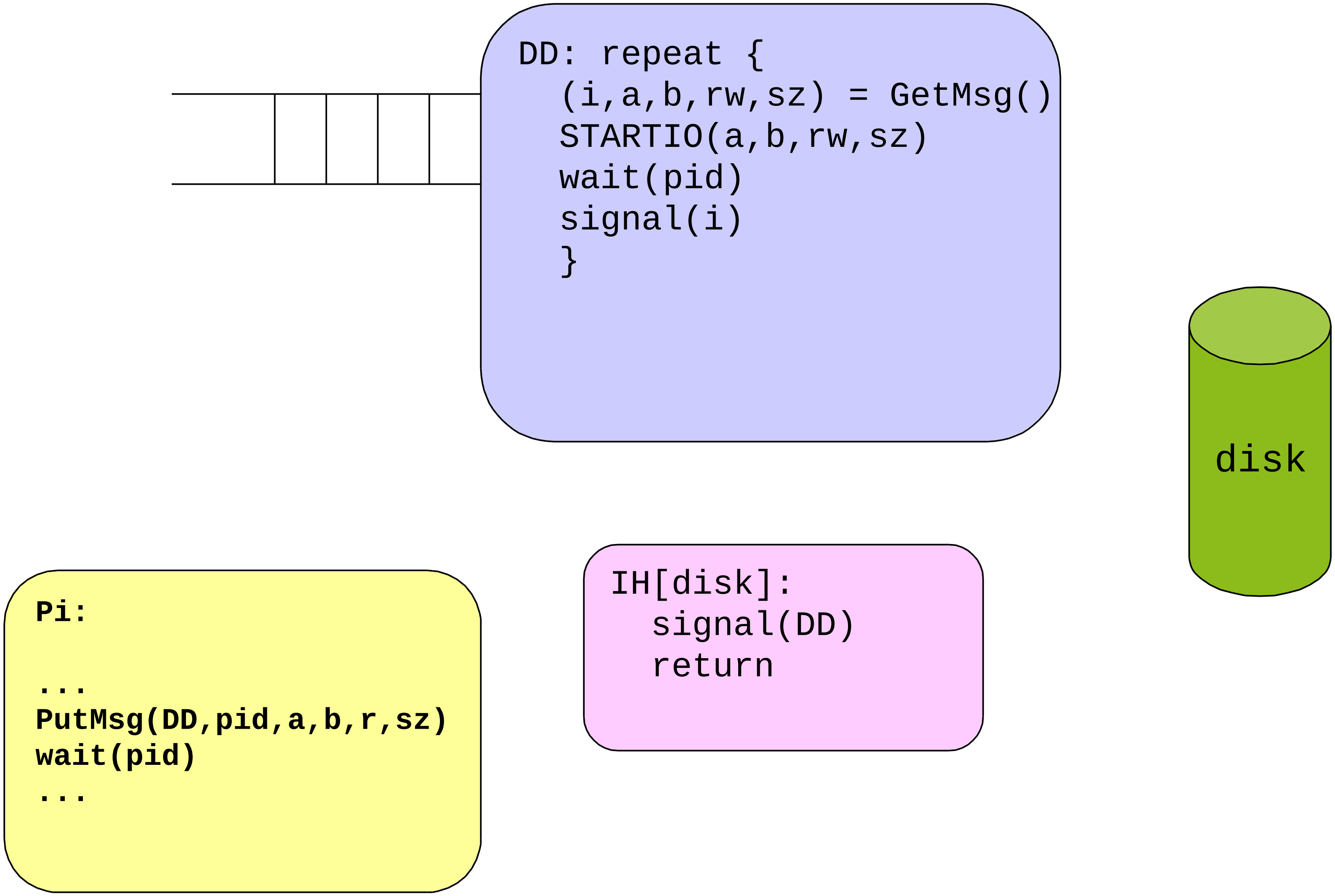
- Five philosophers, round table, five plates, five forks alternating (Dijkstra 1965)
- Philosopher comes to assigned place, eats, and departs at random times
- Philosopher needs left and right forks to eat
- All philosophers follow the same program
- How to prevent deadlock?
- Must monitor global “table state”

Private Semaphore

- Semaphore reserved for private waiting-use by a process
- Reserve semaphore indices j=1,...,N for private semaphores. Then j=N+1,...,M are sharable semaphores.
- Only process i is allowed to call wait(i)
- Private semaphores useful for synchronizing processes simulating procedure calls where process must wait for a return

Private Semaphore

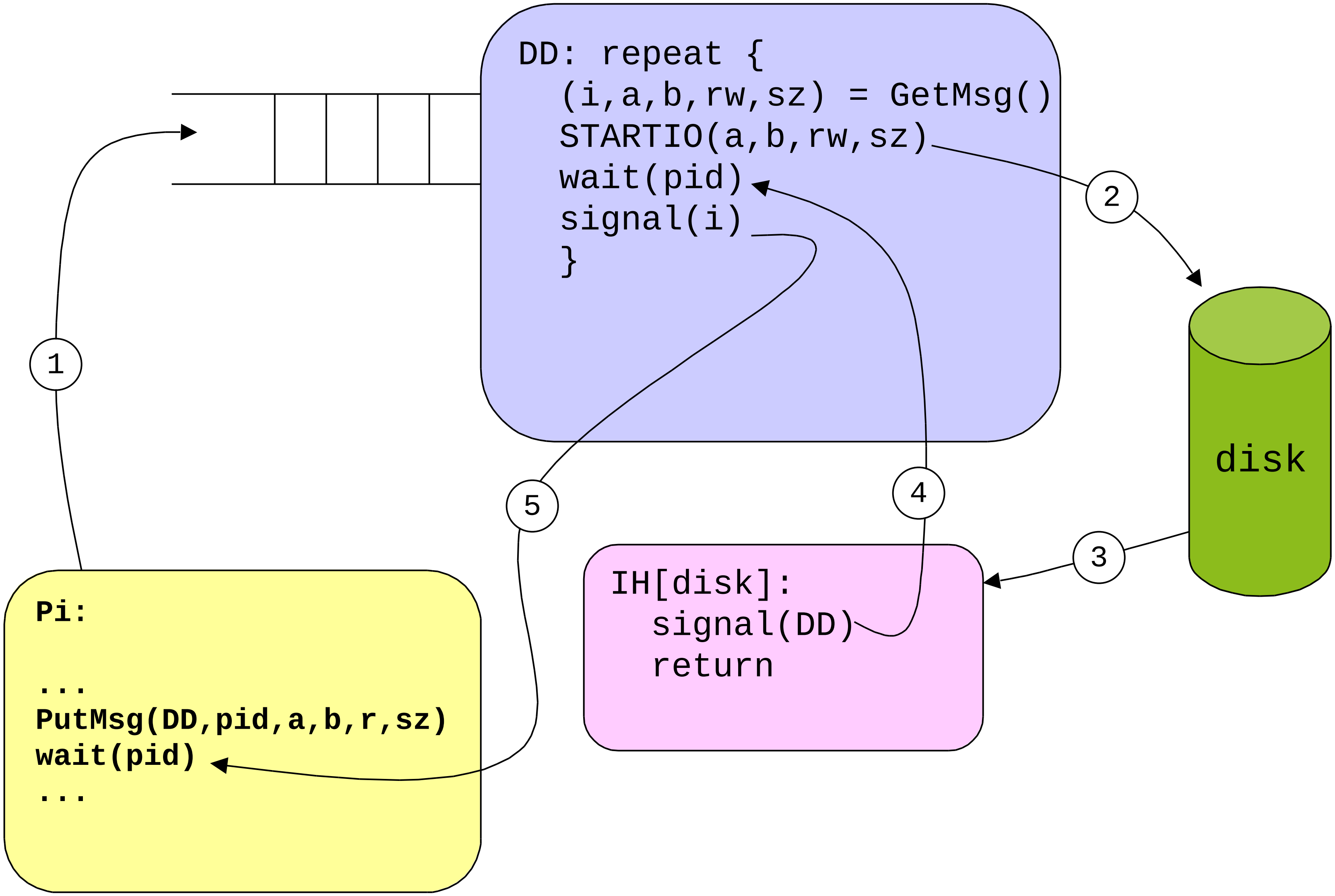
- Example of a disk driver process serving block-move requests from user processes
- Work queue on disk driver collects user requests, driver serves them one at a time
- driver uses STARTIO to pass task to disk
- disk uses completion interrupt to signal done
- disk interrupt handler signals driver to restart
- driver signals user process to restart



```
DD: repeat {
  (i,a,b,rw,sz) = GetMsg()
  STARTIO(a,b,rw,sz)
  wait(pid)
  signal(i)
  }
```

```
Pi:

...
PutMsg(DD,pid,a,b,r,sz)
wait(pid)
...
```

```
IH[disk]:
  signal(DD)
  return
```



```
DD: repeat {
  (i,a,b,rw,sz) = GetMsg()
  STARTIO(a,b,rw,sz)
  wait(pid)
  signal(i)
  }
```

```
IH[disk]:
  signal(DD)
  return
```

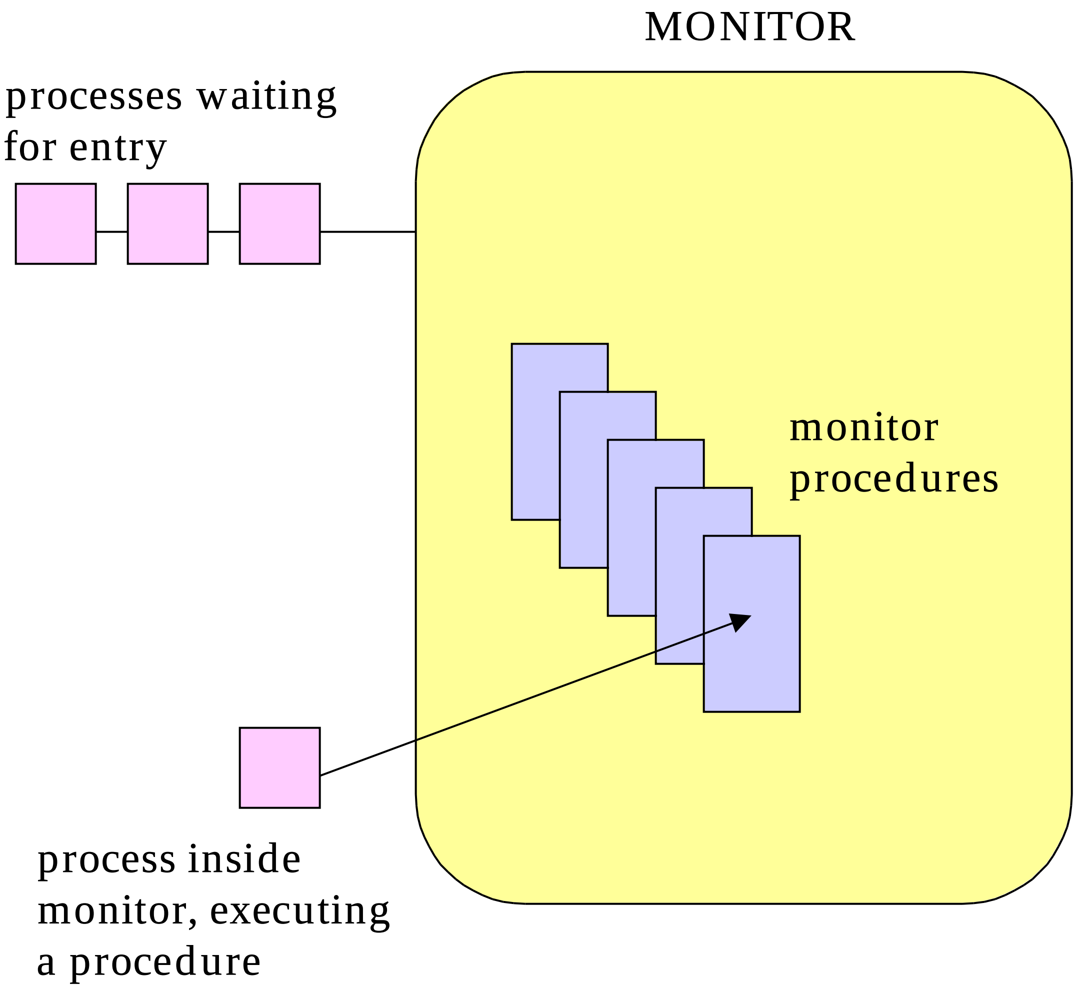
```
Pi:

...
PutMsg(DD,pid,a,b,r,sz)
wait(pid)
...
```

Monitors

- A high level language synchronization structure (Hoare 1978)
- Compiler translates monitor into proper semaphore patterns
- Much improved programming reliability

Monitors

- High level view: monitor is a package of procedures (and data structures); when process enters by calling one of the procedures, the entire monitor is locked to entry by other processes.
- Provides mutually indivisible set of operations on common data.



MONITOR

processes waiting
for entry

monitor
procedures

process inside
monitor, executing
a procedure

Monitors

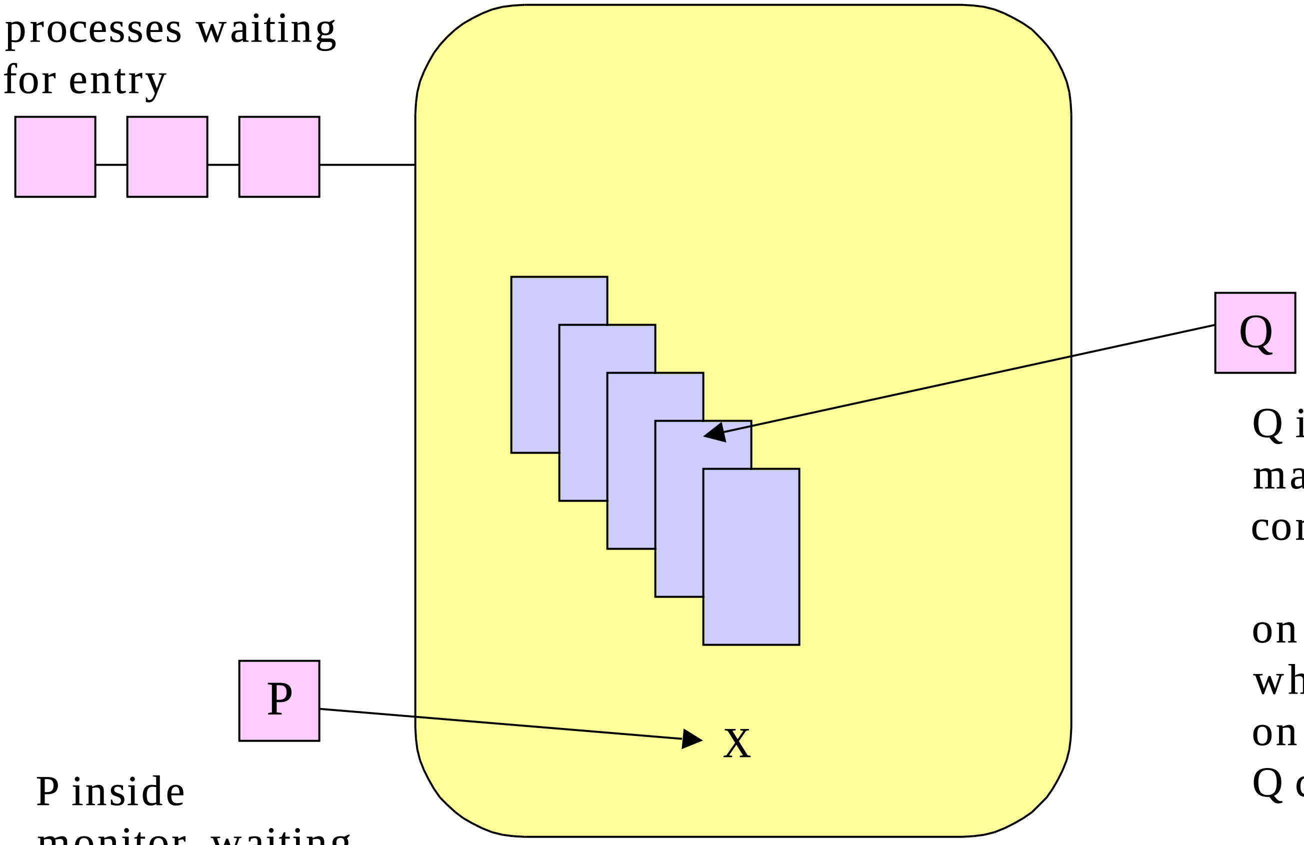
- What if process inside needs to stop and wait?
  - Ex: Pool manager monitor, process executes GetUnit when pool empty?
- How to release monitor exclusion and permit another process to enter and free the waiting one?
  - Ex: another process returns a unit, making it possible for waiting process to proceed.

Monitors

- Condition variable x: denotes a boolean condition
- x.wait -- stop and wait until the condition becomes true
- x.signal -- let a waiting process (if one exists) know that the condition is true

MONITOR

processes waiting
for entry



Q inside monitor,
making condition X
come true;

on X.signal, Q suspends
while P resumes, acts
on X’s being true before
Q can change X again

P inside
monitor, waiting
for condition X

```
monitor poolmgr

condition nonempty

GetUnit:{
  if poolsize=0 then nonempty.wait
  h = “remove unit from pool”
  return h }

ReturnUnit(h):{
  “link h back into pool”
  poolsize++
  if poolsize=1 then nonempty.signal
  return }

end monitor
```