Operating Systems (INFR09047)
2019/2020 Semester 2

Semaphores and Monitors

abarbala@inf.ed.ac.uk

Chapter 6 (6.6 – 6.7)

What About Busy Waiting Solutions?

- ## Disadvantages

  - Waste CPU time waiting
  - High priority jobs may hinder progress of low priority jobs (scheduling)
  - Protect a single resource

```
1 while (TRUE) {
2    enter_region:
3       TAS REGISTER,LOCK_ADDR
4       CMP REGISTER,#0
5       JNE enter_region
6    critical_region(); /* work */
7    leave_region:
8       MOV LOCK_ADDR,#0
9    noncritical_region();
10}
```

Process A and Process B
a) Process A enters its critical region
b) Process A descheduled, Process B scheduled (high priority)
c) Process B busy waits for the critical region (depends on A)
d) Process B will busy wait until Process A is scheduled back
e) A is scheduled back and exits its critical section

sleep() and wakeup()

- Instead of busy wait, let the process sleep
- Introduce new **Operating System** functions
- `sleep()`
  - Caller gives up the CPU for some duration of time
  - Until a thread/process wakes it up
- `wakeup()`
  - Caller wakes up some sleeping thread/process

- In practice
  - As `sleep()` a thread/process may call `yield()` syscall
  - A thread/process may be woken with a signal
  - Each has also a kernel-level implementation

Semaphore

- E.W. Dijkstra, 1965
  - Semaphore variable *value*

```
int value=N; /* protected resource */
```

function ENTER_CRITICAL
- **Proberen (value)** /* P, Wait */

```
value--;
if (value < 0)
  sleep();  /* sleep without completing wait */
```

function EXIT_CRITICAL
- **Verhogen (value)** /* V, Signal */

```
value++;
if (value <= 0)
  wakeup();  /* wakeup one sleeper with a policy */
```

Each function **executes atomically**

Semaphore Implementation

```c
typedef struct {
    int value;
    struct thread *list;
} semaphore;

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this thread to S->list;
        sleep();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a thread T from S->list;
        wakeup(T);
    }
}
```

- A list of TCBs
- The list kept in any **order** (e.g. FIFO)
- User- or kernel-level TCBs
- Each function should be atomic

How to Provide Atomicity?
Need a spinlock! ("real" mutual exclusion lock)

- – P/wait(sem)
  - • acquire "real" mutual exclusion lock
    - – if sem is "available" (>0), decrement sem; release "real" mutual exclusion lock; let thread continue
    - – otherwise, place thread on associated queue; release "real" mutual exclusion lock; run some other thread
- – V/signal(sem)
  - • acquire "real" mutual exclusion lock
    - – if thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
    - – if no threads are on the queue, sem is incremented
      - » the signal is "remembered" for next time P(sem) is called
  - • release "real" mutual exclusion lock
  - • the "V-ing" thread continues execution

Semaphores vs Spinlocks

- **With Semaphores**
- Threads/processes are blocked at the level of **program logic**
  - By the semaphore P/wait operation
  - Placed on queues, rather than busy-waiting
    - The scheduler is **aware** about thread/process **waiting**
    - **Other tasks** can execute
- Busy-waiting may be used for the "real" mutual exclusion lock
  - To implement P/wait and V/signal
  - These are very short critical sections – **independent of program logic**
 - They are not implemented by the application programmer
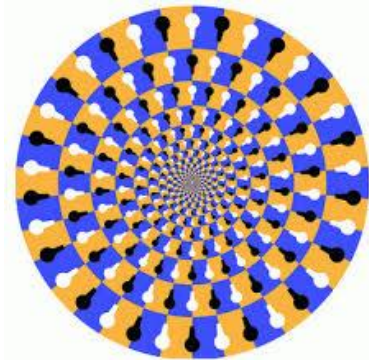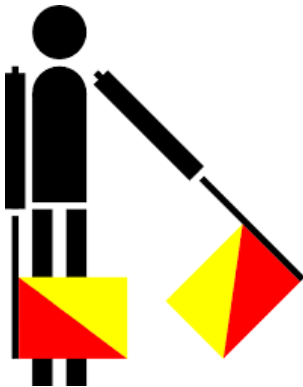
Semaphore Usage

- **Binary semaphore**
- Integer *value* **initialized and capped** to 1
  - 0 means **lock is held**
  - Same as a lock, but no busy waiting
    - `wait()` is `acquire()`
    - `signal()` is `release()`

- **Counting semaphore**
- Integer *value* **initialized and capped** to the # of protected resources
  - **Positive** value means still resource available
  - **Negative or zero** value, how many waiting for resources, no resources available

- **Sychronization semaphore**
- Integer *value* **initialized** to 0
  - **0** means no events pending
  - **Positive** value means there are events pending

Binary Semaphores in Real-life

A Semaphore with **one** resource
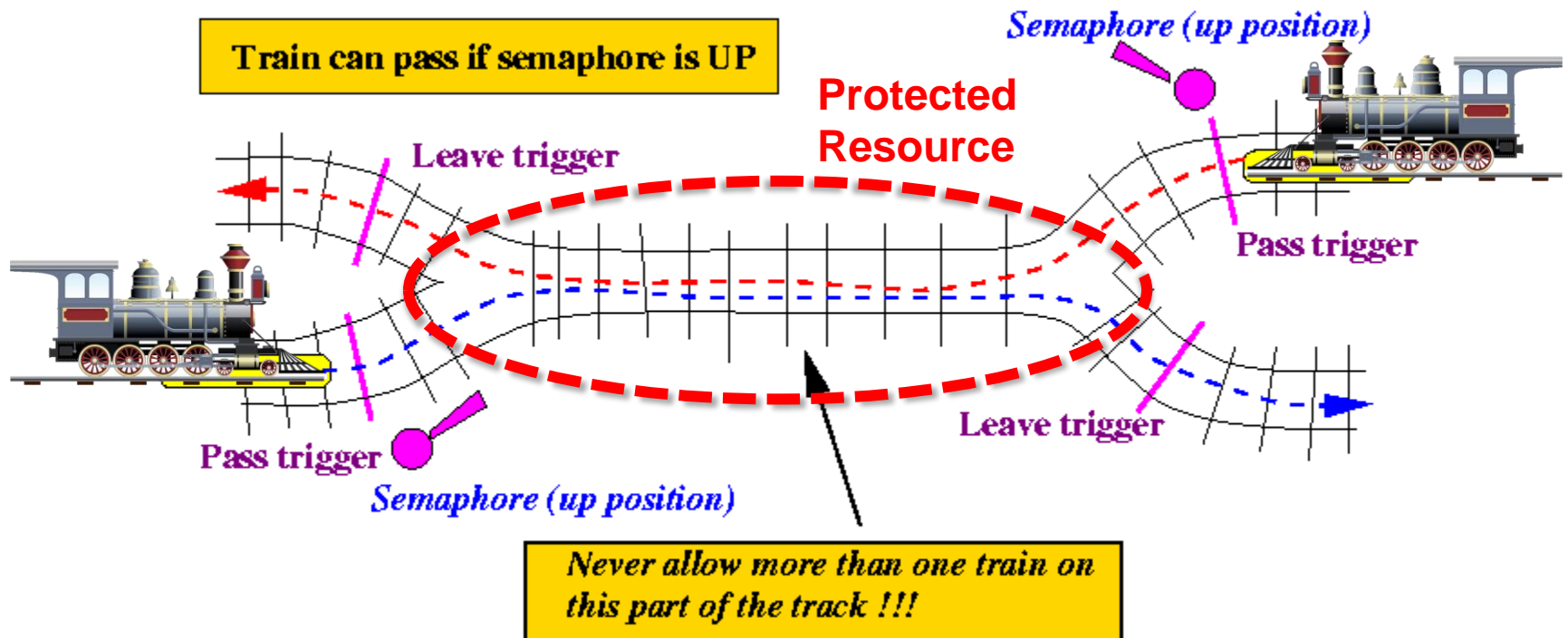
Counting Semaphores in Real-life

MAX PEOPLE IN MUSEUM = **N**

Threads Processes

Protected Resource

MUSEUM **ENTRANCE**

MUSEUM **EXIT**

Wait queue

Semaphore

Signal queue

Synchronization Semaphore

- P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0
  ```
  P1:
      S₁;
      signal(synch);
  P2:
      wait(synch);
      S₂;
  ```
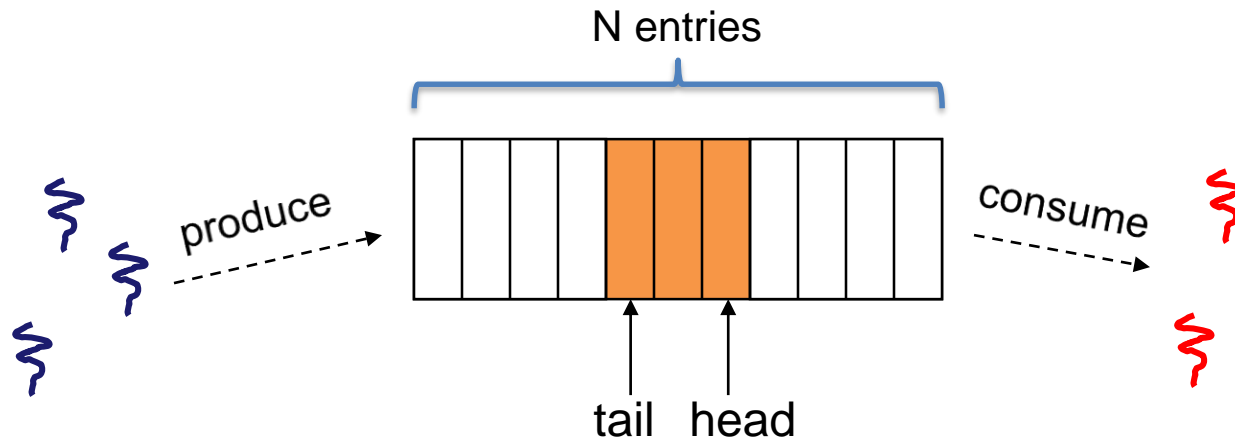
Producer-consumer Problem

- *AKA* "**Bounded-buffer**"
  - there is a circular buffer in memory with N entries (slots)
  - producer threads insert entries into it (one at a time)
  - consumer threads remove entries from it (one at a time)
- Threads/process are concurrent/parallel
  - Must use synchronization constructs to control access to shared variables describing buffer state

Producer-consumer Problem
Counting Semaphores (1)

Shared variables
const int N=100;
int count=0;

Previous code

Shared variables
const int N=100;
semaphore empty=N, full=0;

New code

- In producer-consumer problem ***count*** controls two conditions
  - Buffer empty
    - count == 1 or count == 0
  - Buffer full
    - count == N or count == N-1
- A semaphore controls a single condition on ***value***
  - Two semaphores are needed

Producer-consumer Problem
Counting Semaphores (2)

Previous code

```
Shared variables
const int N=100;
int count=0;

Producer
while (1){
  produce an item A;
  if(count==N) sleep();
  insert item;
  count++;
  if(count==1) wakeup(consumer);
}

Consumer
while (1){
  if(count==0) sleep();
  remove item;
  count--;
  if(count==N-1) wakeup(producer);
  consume an item;
}
```

Full?

Empty ?

Previous code

New code

```
Shared variables
const int N=100;
semaphore full=N, empty=0;

Producer
while (1){
  produce an item A;
  wait(full);
  insert item;
  signal(empty);
}

Consumer
while (1){
  wait(empty);
  remove item;
  signal(full);
  consume an item;
}
```
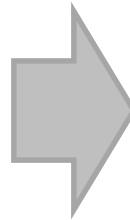
New code

Producer-consumer Problem
Counting and Binary Semaphores (1)

Shared variables
`const int N=100;`
int count=0;

Previous code

Shared variables
`const int N=100;`
semaphore empty=N, full=0;
semaphore mux=1;

New code

- Multiple threads/process trying to add/remove different items on the buffer
  - Operations on the buffer are critical operation
- Mutex to protect buffer operations
  - Avoid races

Producer-consumer Problem
Counting and Binary Semaphores (2)

```
Shared variables
const int N=100;
int count=0;

Producer
while (1){
  produce an item A;
  if(count==N) sleep();
  insert item;
  count++;
  if(count==1) wakeup(consumer);
}

Consumer
while (1){
  if(count==0) sleep();
  remove item;
  count--;
  if(count==N-1) wakeup(producer);
  consume an item;
}
```
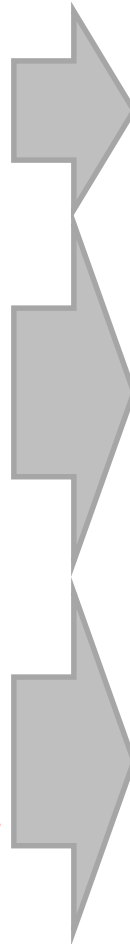
sleep()&wakeup() code

```
Shared variables
const int N=100;
semaphore full=N, empty=0;
semaphore mux=1;
Producer
while (1){
  produce an item A;
  wait(full);
  wait(mux); // acquire(mux)
  insert item;
  signal(mux); // release(mux)
  signal(empty);
}
Consumer
while (1){
  wait(empty);
  wait(mux); // acquire(mux)
  remove item;
  signal(mux); // release(mux)
  signal(full);
  consume an item;
}
```

New code

Example: Readers/Writers

- Description:
  - A single object is shared among several threads/processes
  - Sometimes a thread just reads the object
  - Sometimes a thread updates (writes) the object

  - **We can allow multiple readers at a time**
    - **Do not change state – no race condition**

  - **We can only allow one writer at a time**
    - Change state- race condition

Readers

Writers

Readers/Writers Using Semaphores

```
var mutex: semaphore = 1      ; controls access to readcount
    wrt: semaphore = 1    ; control entry for a writer or first reader
    readcount: integer = 0       ; number of active readers
```

```
writer:
    P(wrt)              ; any writers or readers?
        <perform write operation>
    V(wrt)              ; allow others
```

```
reader:
    P(mutex)                           ; ensure exclusion
      readcount++                      ; one more reader
      if readcount == 1 then P(wrt)      ; if we're the first, synch with writers
    V(mutex)
        <perform read operation>
    P(mutex)                           ; ensure exclusion
      readcount--                      ; one fewer reader
      if readcount == 0 then V(wrt)      ; no more readers, allow a writer
    V(mutex)
```

Readers/Writers Notes

- Notes
  - the first reader blocks on P(wrt) if there is a writer
    - any other readers will then block on P(mutex)

  - if a waiting writer exists, the last reader to exit signals the waiting writer
    - A new reader cannot get in while a writer is waiting

  - When writer exits, if there is both a reader and writer waiting, which one goes next?

Problems with Semaphores and Locks

- Solve any of the **traditional synchronization problems**
- But it is easy to make mistakes
  - Like shared global variables
    - Can be accessed from anywhere (bad software engineering)
  - No connection between the synchronization variable, and the data being controlled
  - No control over their use, no guarantee of proper usage
    - Semaphores: will there ever be a V()?
    - Locks: did you lock when necessary? Unlock at the right time? At all?
- Prone to bugs
  - We can reduce the chance of bugs by "stylizing" the use of synchronization
  - Language help is useful for this

Problem Example

- The sequence of wait and signal operations
  - Producer – wait(full), wait(mux), signal(mux), signal(empty)
  - Consumer – wait(empty), wait(mux), signal(mux), signal(full)
- **Is order of *wait/signal* important?**

Starts with full=N, empty=0, mux=1

```
Producer                           Consumer
while (1){  4                      while (1){           1
  produce an item A;  5              wait(mux);         2
  wait(full);  6                     wait(empty);       3  Blocked!
  wait(mux);  7  Blocked!            remove item;
  insert item;                       signal(mux);
  signal(mux);                       signal(full);
  signal(empty);                     consume an item;
}                                  }
```

Both may block forever!
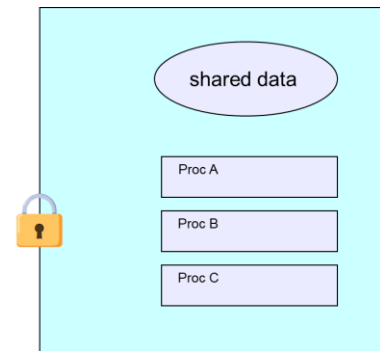
Semaphore Summary

- Solve lock problems
  - No busy-waiting
  - Waiting is controlled by the scheduler
    - Improved scheduling
      - No wasted CPU time
      - Respect task priorities
  - Control multiple resources
- Different usages
  - Binary
  - Counting
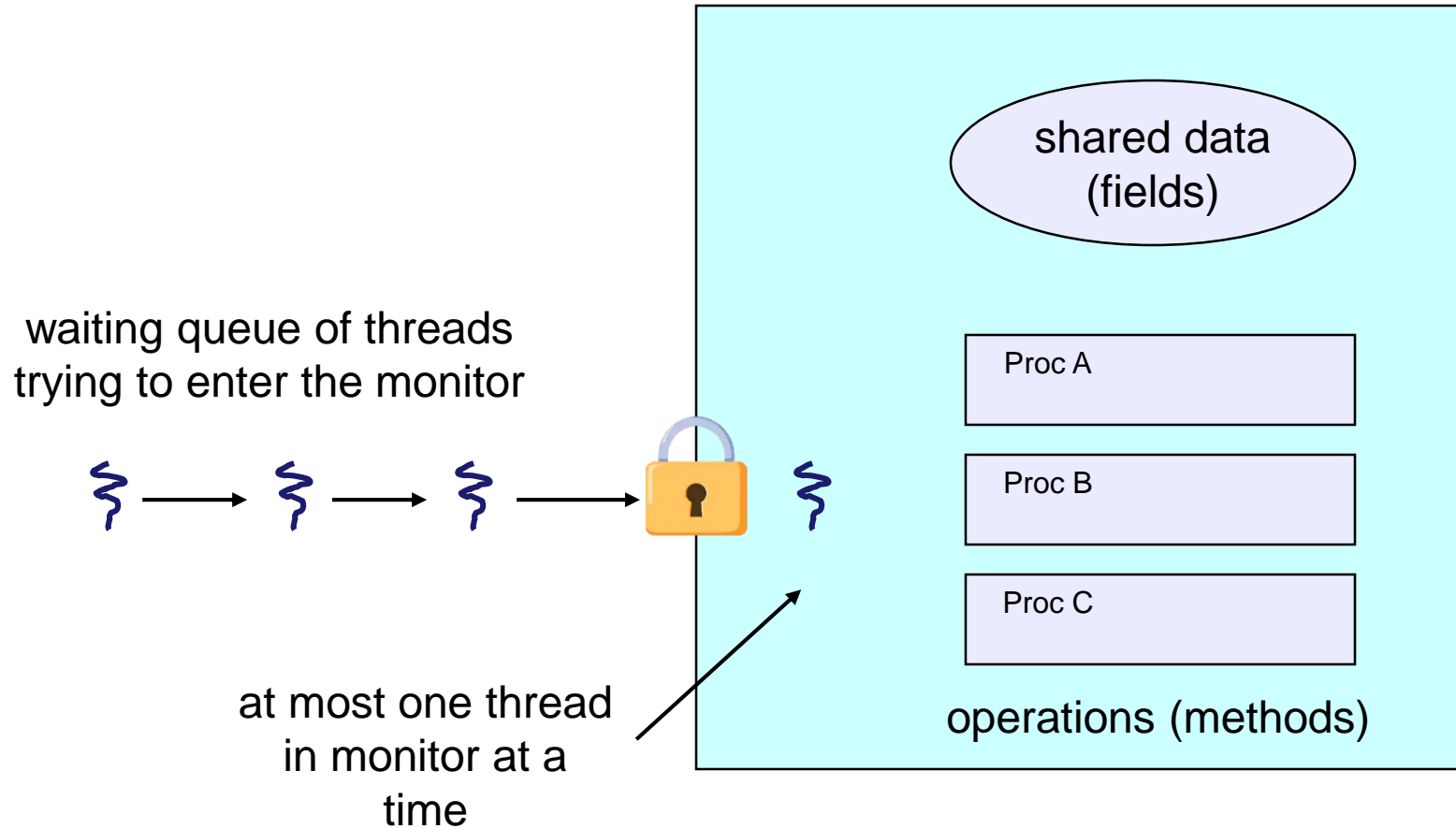 - Synchronization
- Easy to introduce bugs

Monitors

Monitors

- Address the key usability issues with semaphores
- A **programming language construct** to support **controlled shared data access**
  - Synchronization code is added by the compiler/language VM

- An **abstract data type/class** in which every method **automatically**
  - acquires a lock on entry
  - releases the lock on exit
- Includes
  - **shared data** structures (object fields)
  - **procedures** that operate on the shared data (object methods)
  - **synchronization** between concurrent execution flows that invoke those procedures

shared data

Proc A

Proc B

Proc C

- Data can only be accessed **from within**
  - Protects the data from unstructured access
  - Prevents ambiguity about synchronization

A monitor

shared data
(fields)

waiting queue of threads
trying to enter the monitor

Proc A

Proc B

Proc C

at most one thread
in monitor at a
time

operations (methods)

Monitor facilities

- **Automatic** mutual exclusion
  - Only **one thread** can be executing inside at any time
    - Synchronization is implicitly associated with the monitor
      - it "comes for free"
  - If a second thread tries to execute a monitor procedure it **blocks** until the first has left the monitor
    - More restrictive than semaphores
    - Easier to use (most of the time)

- However, there's a problem…

Problem: Producer-consumer Scenario
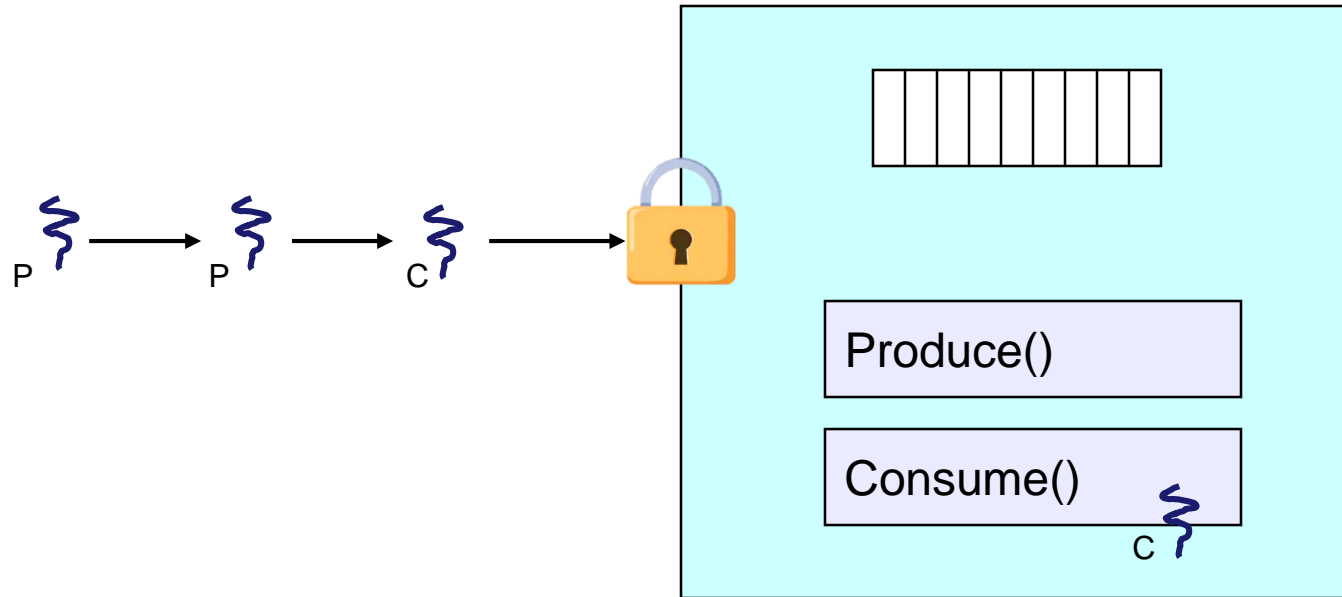
- Buffer is **empty**
- Now what?

Problem: Producer-consumer Scenario
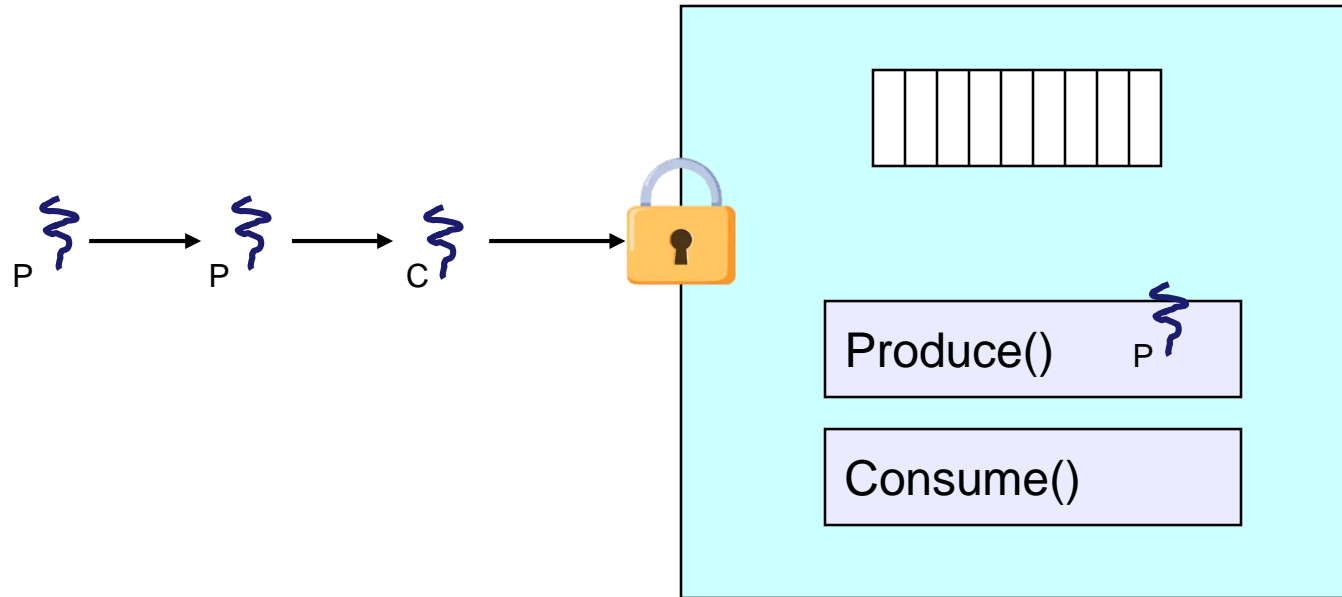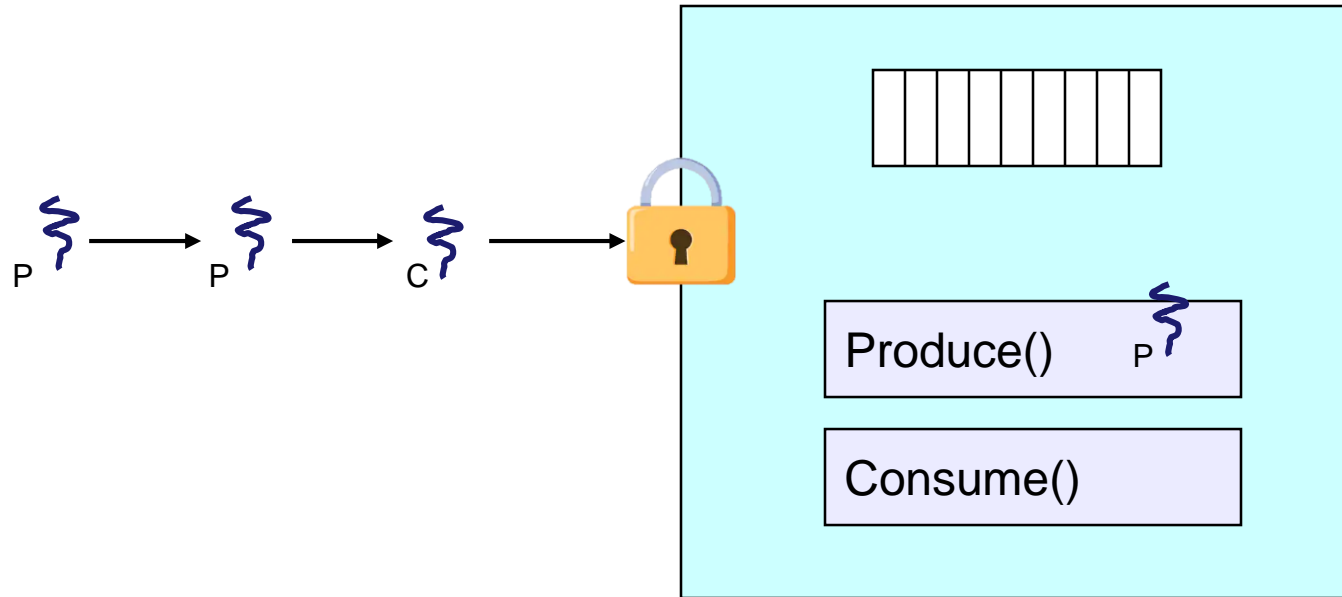

- Buffer is **full**
- Now what?

Solution?

- **Condition variables**
- Operations on condition variables
  - **wait(c)**
    - Release monitor lock, so somebody else can get in
    - Wait for somebody else to signal condition
    - Condition variables have associated wait queues
  - **signal(c)**
    - Wake up at most one waiting thread
      - "Hoare" monitor: wakeup immediately, signaller steps outside
    - If no waiting threads, signal is lost
      - this is different than semaphores: no history!
  - **broadcast(c)**
    - Wake up all waiting threads

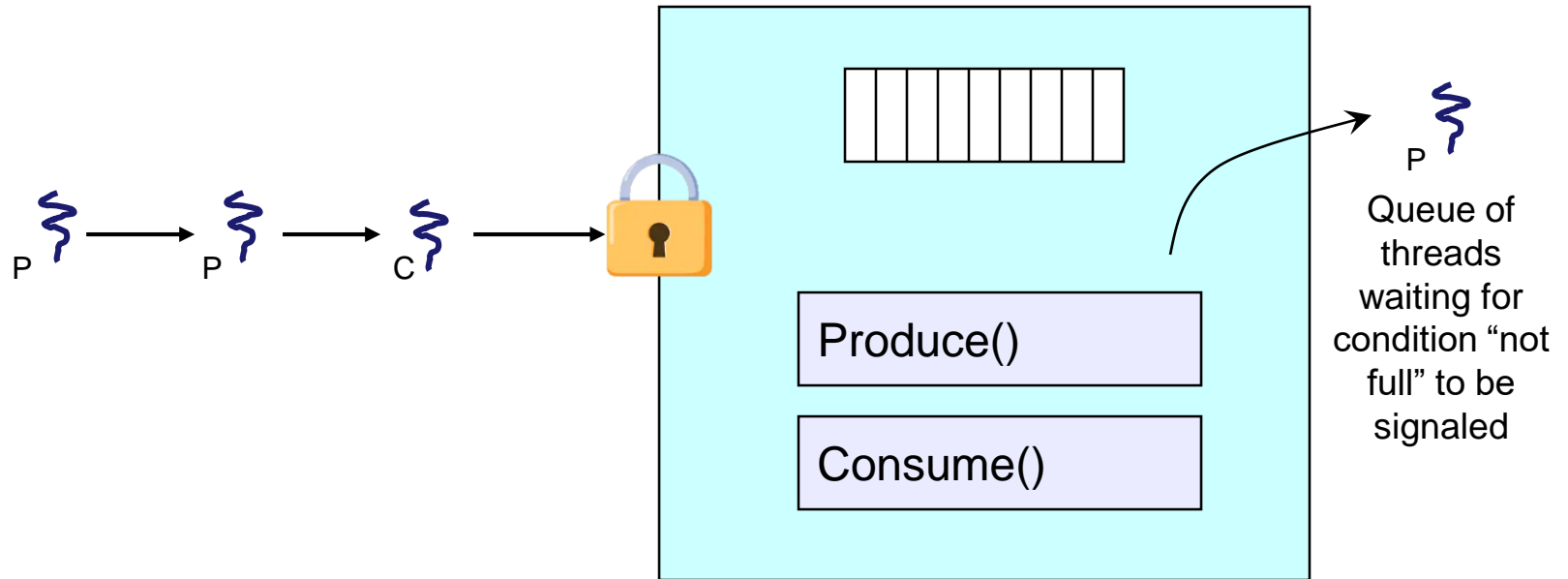# Producer-consumer using (Hoare) monitors

```
Monitor producer_consumer {
  buffer resources[N];
  condition not_full, not_empty;

produce(resource x) {
    if (array "resources" is full, determined maybe by a count)
        wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);
  }

 consume(resource *x) {
    if (array "resources" is empty, determined maybe by a count)
        wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);
  }
}
```

Problem: Producer-consumer Scenario

- Buffer is **full**
- Now what?

Producer-consumer Scenario with Conditional Variable

P   P   C

Produce()

Consume()

P

Queue of threads waiting for condition "not full" to be signaled

- Buffer is **full**
- Now what?

Runtime Functions for (Hoare) Monitors

- EnterMonitor(m)
  - guarantee mutual exclusion
- ExitMonitor(m)
  - hit the road, letting someone else run
- Wait(c)
  - step out until condition satisfied
- Signal(c)
  - if someone's waiting, step out and let him run

- EnterMonitor and ExitMonitor are inserted automatically by the compiler
- This guarantees mutual exclusion for code inside of the monitor

Producer-consumer Using (Hoare) Monitors

```
Monitor producer_consumer {
  buffer resources[N];
  condition not_full, not_empty;

produce(resource x) {
```
.... EnterMonitor(m)
```
    if (array "resources" is full, determined maybe by a count)
        wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);
```
.... ExitMonitor(m)
```
  }

  consume(resource *x) {
```
.... EnterMonitor(m)
```
    if (array "resources" is empty, determined maybe by a count)
        wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);
```
.... ExitMonitor(m)
```
  }
}
```

34

Monitor Summary

- Language supported
- Compiler understands them
  - Compiler inserts calls to
    - monitor entry
    - monitor exit
  - Programmer inserts calls to
    - signal
    - wait
  - Language/object encapsulation ensures correctness
    - With conditions, you *still* need to think about synchronization
- Runtime system implements these routines
  - moves threads on and off queues
  - *ensures mutual exclusion!*