Monitors

- A *monitor* is a programming language construct that supports controlled access to shared data
  - synchronization code is added by the compiler
    - why does this help?
- A monitor encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent threads that invoke those procedures
- Data can only be accessed from within the monitor, using the provided procedures
  - protects the data from unstructured access
- Addresses the key usability issues that arise with semaphores

Characteristics of Monitors.

1. Inside the monitors, we can only execute one process at a time.

2 . Monitors are the group of procedures, and condition variables that are merged together in a special type of module.

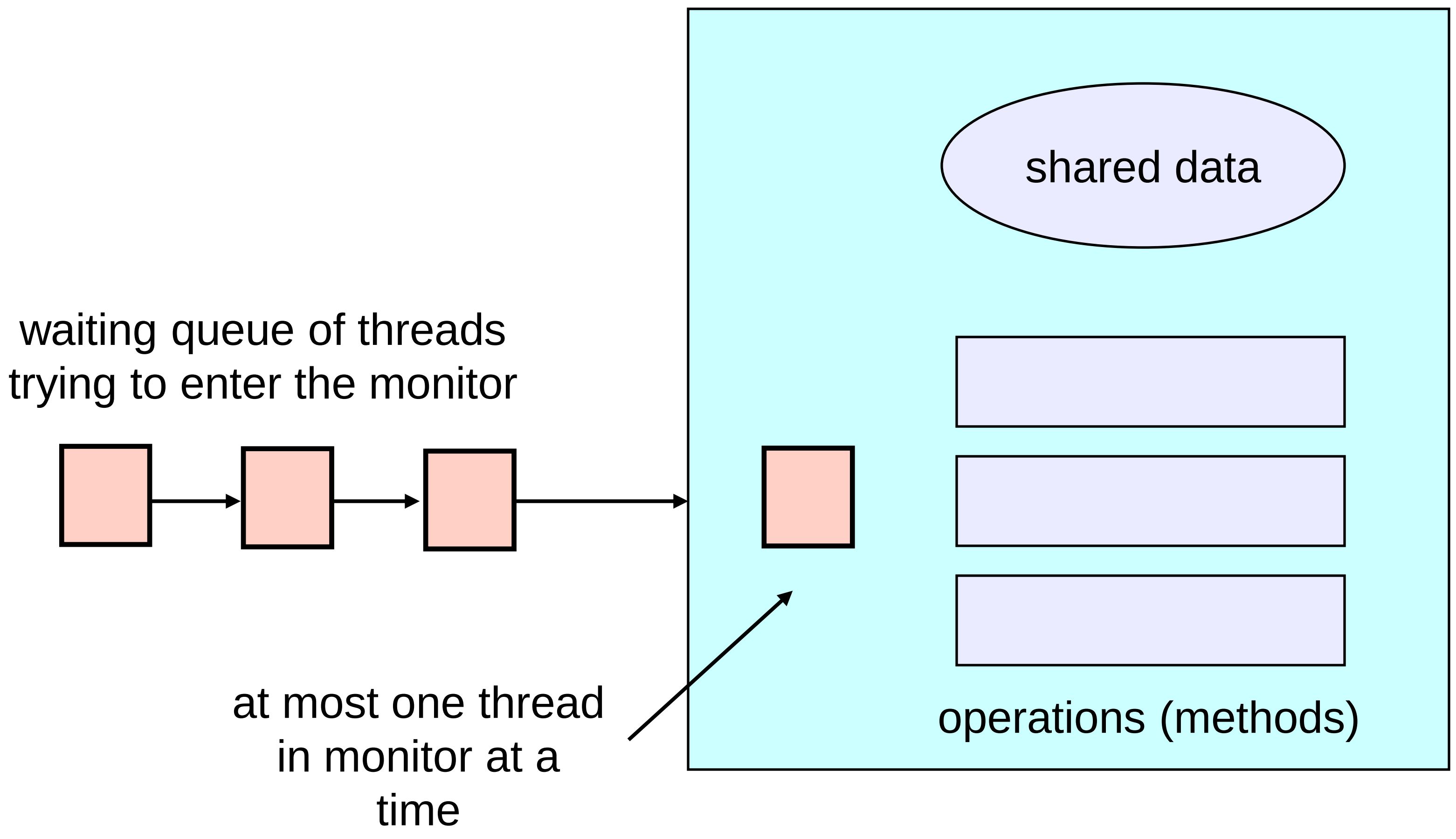
3. If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.

4. Monitors offer high-level of synchronization

5. Monitors were derived to simplify the complexity of synchronization problems.

6. There is only one process that can be active at a time inside the monitor.

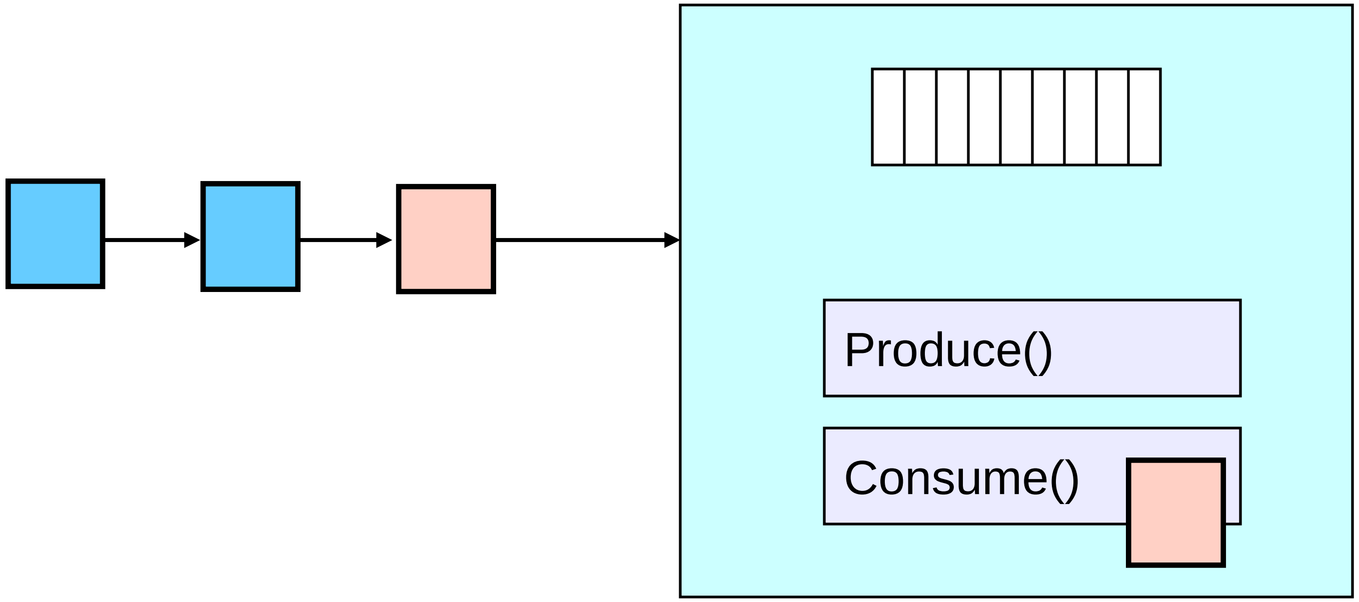
A monitor


shared data
waiting queue of threads trying to enter the monitor
at most one thread in monitor at a time
operations (methods)

Monitor facilities

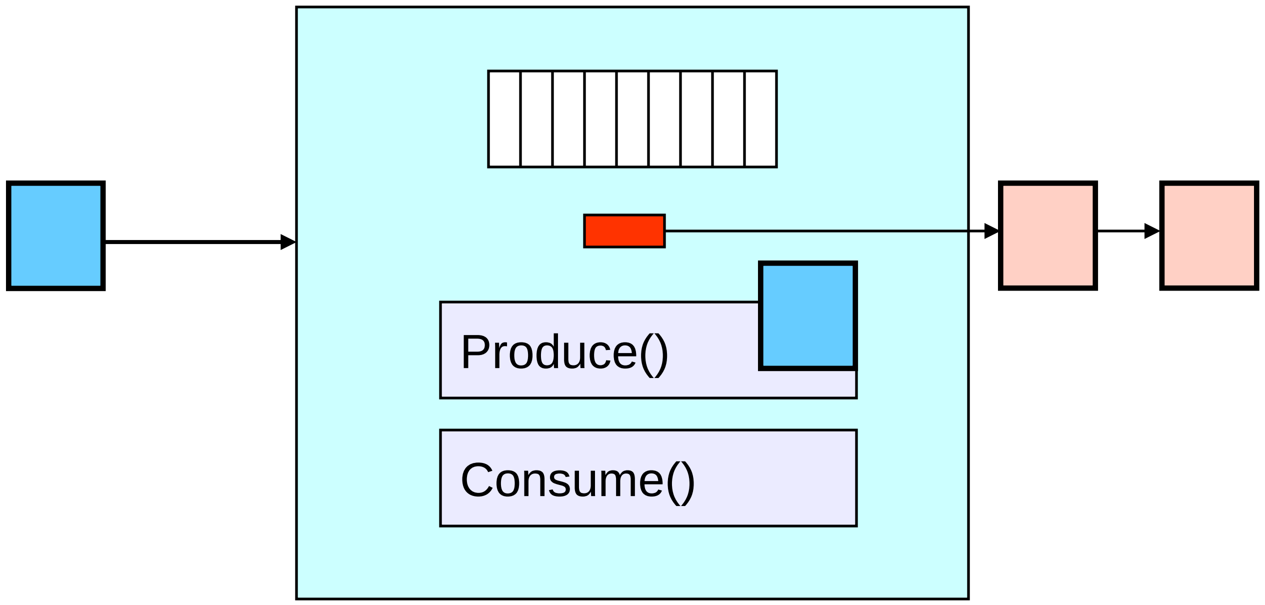
- “Automatic” mutual exclusion
  - only one thread can be executing inside at any time
    - thus, synchronization is implicitly associated with the monitor – it “comes for free”
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
    - more restrictive than semaphores
    - but easier to use (most of the time)
- But, there’s a problem…

Example: Bounded Buffer Scenario



- Buffer is empty
- Now what?

Example: Bounded Buffer Scenario



- Buffer is empty
- Now what?

- **Components of Monitor**
- There are four main components of the monitor:
- Initialization
- Private data
- Monitor procedure
- Monitor entry queue
- **Initialization: –** Initialization comprises the code, and when the monitors are created, we use this code exactly once.
- **Private Data:** – Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.

- **Monitor Procedure:** – Monitors Procedures are those procedures that can be called from outside the monitor.
- **Monitor Entry Queue: –** Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.

Condition variables

- Three operations on condition variables

  wait(c)

  - The process that performs wait operation on the condition variables are suspended and locate the suspended process in a block queue of that condition variable.
  - if the resource is currently not availabe.current process put to sleep.it release a lock of monitor.

  signal(c)

  - wake up at most one waiting thread
  - If a signal operation is performed by the process on the condition variable, then a chance is provided to one of the blocked processes.
  - if no waiting threads, signal is lost
    - this is different than semaphores: no history

  broadcast(c)

  - wake up all waiting threads

Bounded buffer using monitors

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

  procedure add_entry(resource x) {                          EnterMonitor
    if (array "resources" is full, determined maybe by a count)
      wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);                                       ExitMonitor
  }
  procedure get_entry(resource *x) {                         EnterMonitor
    if (array "resources" is empty, determined maybe by a count)
      wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);                                        ExitMonitor
  }
```

Runtime system calls for monitors

- Enter Monitor(m) {guarantee mutual exclusion}
  - if m occupied, insert caller into queue m
  - else mark as occupied, insert caller into ready queue
  - choose somebody to run
- Exit Monitor(m) {hit the road, letting someone else run}
  - if queue m is empty, then mark m as unoccupied
  - else move a thread from queue m to the ready queue
  - insert caller in ready queue
  - choose someone to run

Runtime system calls for monitors (cont’d)

- Wait(c) {step out until condition satisfied}
  - if queue m is empty, then mark m as unoccupied
  - else move a thread from queue m to the ready queue
  - put the caller on queue c
  - choose someone to run
- Signal(c) {if someone’s waiting, step out and let him run}
  - if queue c is empty then put the caller on the ready queue
  - else move a thread from queue c to the ready queue, and put the caller into queue m
  - choose someone to run

Monitor Summary

- Language supports monitors
- Compiler understands them
  - compiler inserts calls to runtime routines for
    - monitor entry
    - monitor exit
    - signal
    - Wait
  - Language/object encapsulation ensures correctness
    - Sometimes! With conditions you STILL need to think about synchronization
- Runtime system implements these routines
  - moves threads on and off queues
  - *ensures mutual exclusion!*