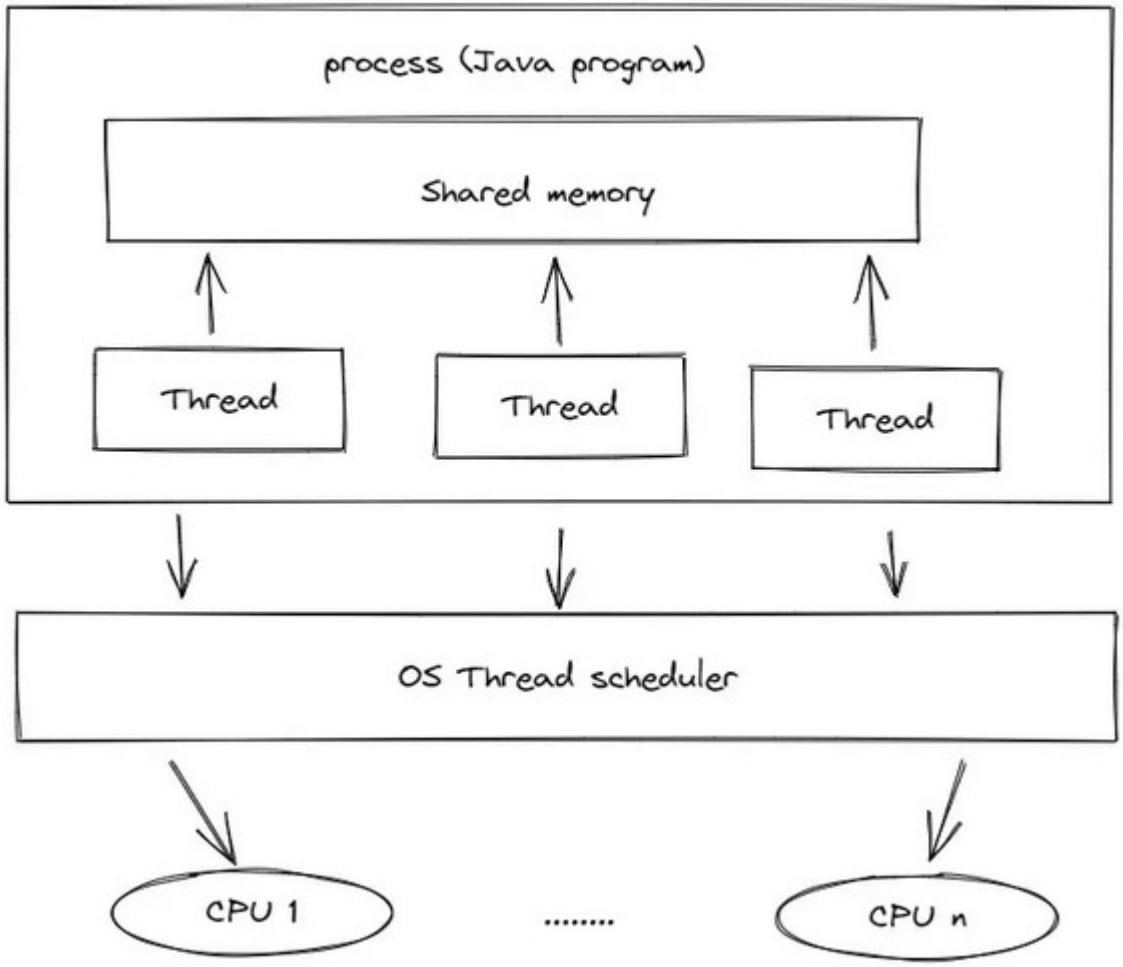
CHƯƠNG 2:
SONG SONG DỰA TRÊN LUỒNG

MÔ HÌNH ĐA LUỒNG (THE MULTITHREAD MODEL)

- Thread được mô tả đơn giản như một chương trình bao gồm nhiều chương trình con
  - Chương trình chính được lập lịch để chạy trên máy với những yêu cầu cần thiết
  - Chương trình chính thực hiện nối tiếp một vài công việc và lập lịch để cho các task (các thread) thực thi một cách đồng thời



TỔNG QUAN

- Một luồng thực thi độc lập, có thể được thực hiện song song và đồng thời với các luồng khác trong hệ thống.
- Nhiều luồng có thể chia sẻ dữ liệu và tài nguyên với luồng khác có cùng tiến trình.
- Mỗi luồng bao gồm ba yếu tố: bộ đếm chương trình (program counter), thanh ghi (register) và ngăn xếp (stack).
- Trạng thái thực thi của một luồng: sẵn sàng (ready), đang chạy (running) và bị chặn (blocked).
- Lập trình đa luồng ưu tiên phương thức giao tiếp giữa các luồng sử dụng không gian thông tin được chia sẻ -> vấn đề chính cần được giải quyết là quản lý không gian đó.

MODULE “THREADING” TRONG PYTHON

- Các thành phần chính:
  - Luồng (thread)
  - Lock
  - Rlock
  - Semaphore
  - Điều kiện (condition)
  - Sự kiện (event)

MODULE “THREADING” TRONG PYTHON

```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={})
```

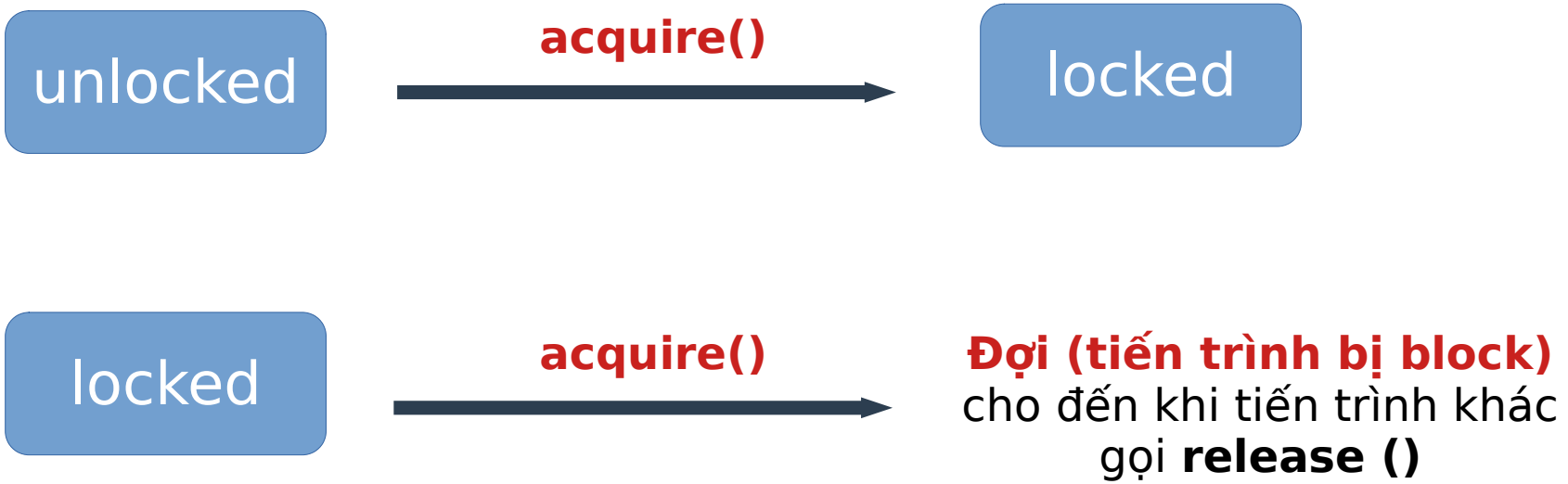
- **group**: thường là “None”; được dành riêng cho việc triển khai trong tương lai.
- **target**: hàm sẽ được thực thi khi bắt đầu một hoạt động của luồng.
- **name**: tên của luồng; theo mặc định, một tên duy nhất có dạng **Thread-N** được gán cho nó.
- **args**: bộ (tuple) đối số được truyền cho **target.**
- **kwargs**: từ điển (dictionary) các đối số từ khóa (keyword) sẽ được sử dụng cho hàm **target**.

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

- Khi hai hoặc nhiều operations thuộc các luồng đồng thời cố gắng truy cập vào bộ nhớ dùng chung và ít nhất một trong số đó có quyền thay đổi trạng thái của dữ liệu mà không có cơ chế đồng bộ hóa phù hợp → lỗi
- Cơ chế đồng bộ **Lock** trong Python
  - Thread muốn truy cập vào một phần của bộ nhớ dùng chung → phải có khóa (Lock) trên phần đó trước khi sử dụng.
  - Sau khi thực hiện xong phép toán → thread phải giải phóng lock đã nhận được trước đó → một phần bộ nhớ dùng chung sẵn sàng cho bất kỳ luồng nào khác muốn sử dụng nó.
  - Tại mỗi thời điểm, chỉ có 1 thread sở hữu lock

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

- 02 **trạng thái** cơ bản của lock: **locked** và **unlocked**
- 02 **phương thức** thao tác với lock: **acquire()**và **release()**



ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

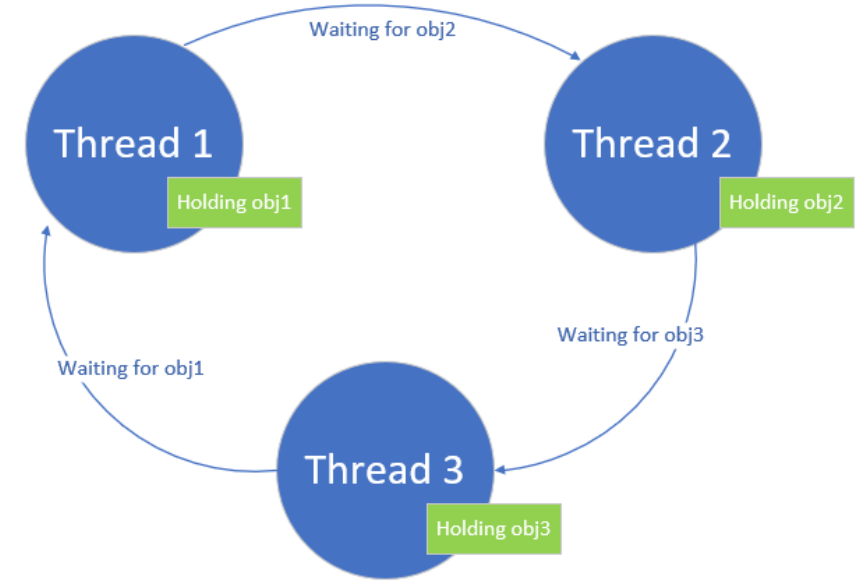
- 02 **trạng thái** cơ bản của lock: **locked** và **unlocked**
- 02 **phương thức** thao tác với lock: **acquire()**và **release()**

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

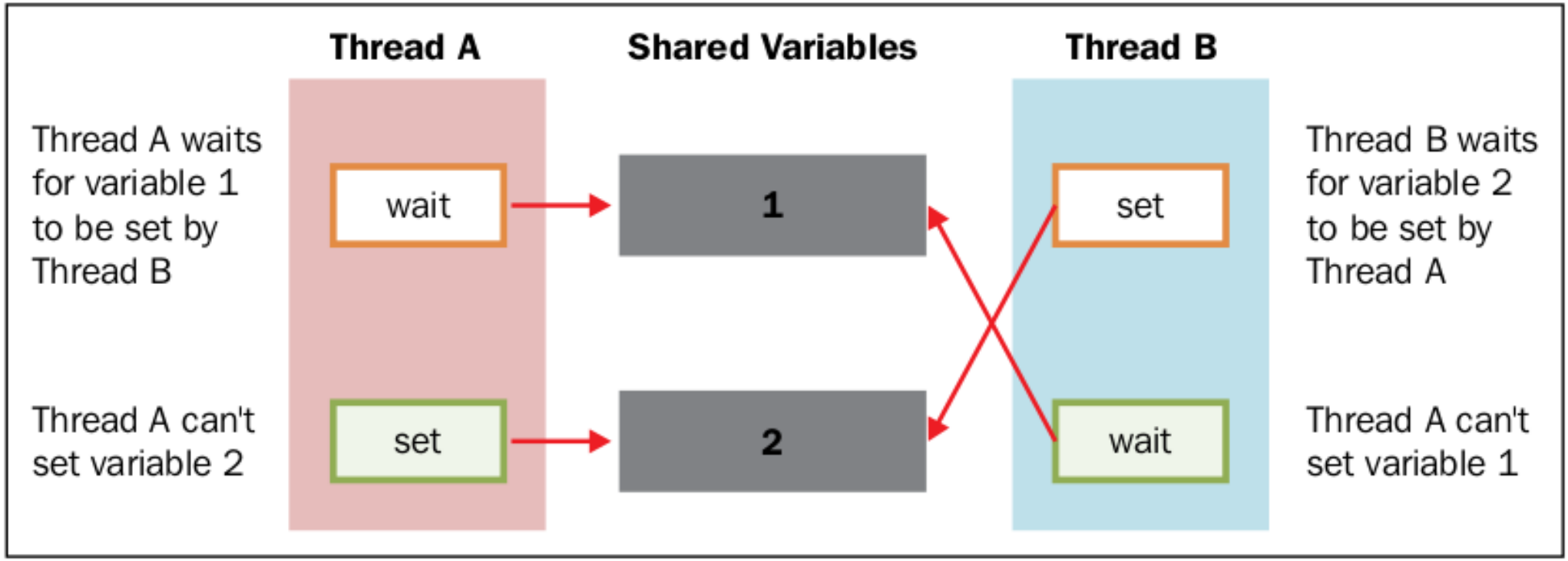
- **lock overhead**:
  - Là tài nguyên bổ sung cần để sử dụng lock
    - → Không gian bộ nhớ được phân bổ cho lock;
    - → CPU time để khởi tạo và hủy lock;
    - → Thời gian để accuire() hoặc release() lock.
  - Một chương trình sử dụng càng nhiều lock thì càng cần nhiều overhead.
- **lock contention (xung đột/tranh chấp)**:
  - xuất hiện khi 1 process hoặc thread cố gắng giành được (acquire) lock đang được giữ bởi 1 process hoặc thread khác.
  - Các lock càng fine-grain (mịn, nghĩa là càng chi tiết) thì càng ít khả năng 1 process/thread sẽ yêu cầu 1 lock đang giữ bởi 1 process/thread khác.

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

- **deadlock**:
  - Nhiều luồng (ít nhất 2) đang đợi lock mà luồng khác đang giữ.
  - Xuất hiện khi 1 process/ thread bước vào trạng thái waiting bởi vì yêu cầu đến tài nguyên hệ thống đang được giữ bởi 1 waiting process khác (waiting process khác này lại đang waiting cho 1 tài nguyên khác khác đang được giữ bởi 1 waiting process khác)
  - Nếu 1 process mãi mãi (vô hạn - indefinitely) không thể thay đổi trạng thái của nó bởi vì các tài nguyên nó đang yêu cầu đang được sử dụng bởi 1 waiting process khác thì hệ thống được xem như là bị deaklock.



ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK



Deadlock

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

- **Luồng A**: yêu cầu tài nguyên **1**, **luồng B**: yêu cầu tài nguyên **2**

  → 02 luồng yêu cầu **lock riêng** → **bình thường**

- Giả sử: trước khi giải phóng lock, **luồng A** yêu cầu tài nguyên **2**, **luồng B** yêu cầu tài nguyên **1**

  → 2 tài nguyên 1 và 2 đều đang ở trạng thái **locked** → 2 luồng đều **bị chặn (blocked) và đợi nhau**.

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

- **Hạn chế**:
  - Đưa ra các **overhead** không cần thiết.
  - Giới hạn khả năng **đọc và mở rộng code**.
  - Mâu thuẫn với nhu cầu có thể áp đặt **quyền ưu tiên truy cập** vào bộ nhớ được chia sẻ bởi các tiến trình khác nhau.
  - Gây khó khăn khi tìm kiếm lỗi (debugging)

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

- Nếu muốn lock chỉ được giải phóng (release) với thread sở hữu nó → sử dụng đối tượng (object) **Rlock (Re-entrant lock)**.
- Hữu ích khi muốn có quyền truy cập an toàn theo luồng từ bên ngoài lớp và sử dụng các phương thức tương tự từ bên trong lớp.
- 02 **phương thức** thao tác với Rlock: **acquire()**và **release().**
- Sử dụng khái niệm **"owning thread" (tiến trình sở hữu)** và **"recursion level"** (có thể gọi phương thức **acquire() nhiều lần**).

ĐỒNG BỘ HÓA LUỒNG VỚI LOCK VÀ RLOCK

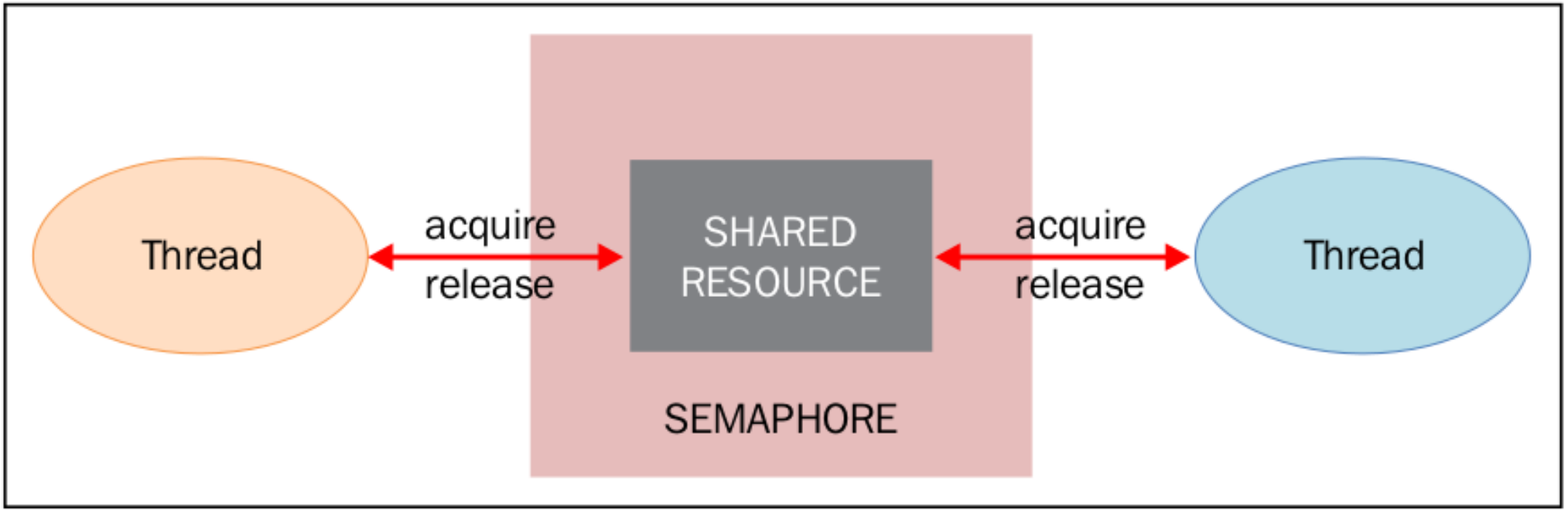
- **Owning thread**:
  - Rlock lưu giữ định danh (ID) của luồng sở hữu nó → luồng đó có thể gọi acquire() nhiều lần mà không bị blocked
  - Lock chỉ chuyển sang trạng thái unlocked khi luồng sở hữu nó gọi release()
  - Số lần gọi release() bằng số lần gọi acquire() trước đó.
- **Recursion level**:
  - Khi 1 thread giành được khóa → thread đó có thể gọi acquire() nhiều lần mà không bị blocked.
  - Hữu ích cho các hàm đệ quy.

ĐỒNG BỘ HÓA LUỒNG VỚI SEMAPHORE

- Là phương pháp đồng bộ hóa cổ điển được đưa ra bởi Dijkstra (1965) nhằm **bảo vệ dữ liệu chia sẻ giữa các luồng hay tiến trình**.
- Được sử dụng để đồng bộ hóa quyền **truy cập tuần tự của nhiều luồng đến tài nguyên và dữ liệu chia sẻ**/ vùng code nào đó trong chương trình.
- Semaphore thường được dùng để giới hạn số lượng luồng đồng thời truy cập vào tài nguyên.
- Một semaphore được cấu thành từ một biến đếm (internal variable) xác định số lượng truy cập đồng thời vào tài nguyên mà nó được liên kết.

ĐỒNG BỘ HÓA LUỒNG VỚI SEMAPHORE

- 02 **phương thức** thao tác với lock: **acquire()**và **release()**
- Semaphore duy trì một **biến đếm** được truyền vào khi khởi tạo một đối tượng Semaphore



Thread synchronization with semaphores

ĐỒNG BỘ HÓA LUỒNG VỚI SEMAPHORE

- Luồng muốn **truy cập một tài nguyên** được liên kết với một semaphore → gọi **acquire()** → giá trị của **biến đếm giảm**
  - Nếu giá trị biến đếm *không âm* → cho phép *truy cập tài nguyên*
  - Nếu giá trị biến đếm bằng *0* → blocked luồng gọi và đợi cho đến khi luồng khác gọi *release()* → làm *tăng giá trị biến đếm lên 1*.
  - Nếu giá trị biến đếm *âm* → luồng sẽ bị *treo* và việc giải phóng tài nguyên bởi một luồng khác sẽ bị tạm dừng.
- Luồng sử dụng xong dữ liệu hoặc tài nguyên được chia sẻ → **giải phóng tài nguyên** thông qua **release()** → giá trị **biến đếm tăng** → luồng chờ đầu tiên trong hàng đợi của Semaphore có quyền truy cập vào tài nguyên được chia sẻ

ĐỒNG BỘ HÓA LUỒNG VỚI SEMAPHORE

- 02 luồng thực thi đồng thời, thao tác chờ trên một semaphore, biến đếm có giá trị 1.
- Giả sử: sau khi luồng đầu tiên có semaphore, biến đếm giảm từ 1 xuống 0, điều khiển chuyển sang luồng thứ hai, luồng này giảm từ 0 xuống -1 → biến đếm âm.
- Tại thời điểm này, với điều khiển trả về luồng đầu tiên, semaphore có giá trị âm và do đó, luồng đầu tiên cũng chờ.

ĐỒNG BỘ HÓA LUỒNG VỚI ĐIỀU KIỆN (CONDITION)

- Một điều kiện xác định sự thay đổi trạng thái trong ứng dụng.
- Đây là một cơ chế đồng bộ hóa trong đó một luồng **chờ một điều kiện cụ thể** và một luồng khác thông báo rằng điều kiện này đã diễn ra.
- Khi điều kiện diễn ra, luồng sẽ có được khóa để có quyền **truy cập độc quyền** vào tài nguyên được chia sẻ.
- Các luồng khác không được phép truy cập vào tài nguyên khi luồng này chưa release khóa.

ĐỒNG BỘ HÓA LUỒNG VỚI ĐIỀU KIỆN (CONDITION)

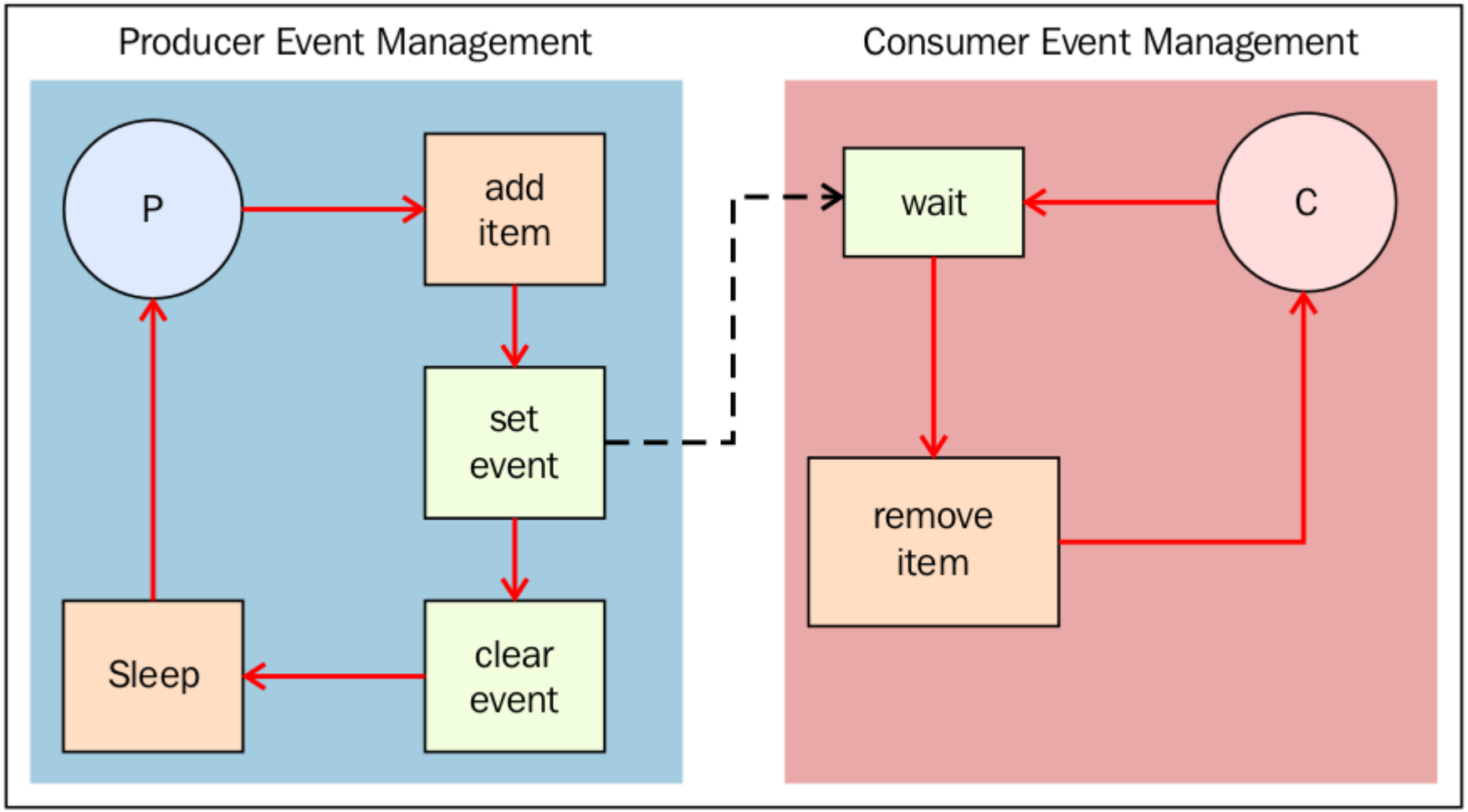
- **Ví dụ:** bài toán producer/consumer (sản-xuất/tiêu-thụ).
  - Luồng producer ghi (sản-xuất) vào bộ đệm tại những thời điểm ngẫu nhiên, miễn là nó không đầy.
  - Luồng consumer lấy (tiêu-thụ) dữ liệu từ bộ đệm đó, miền là bộ đệm đầy.
  - Producer sẽ thông báo cho consumer rằng bộ đệm không trống, trong khi consumer sẽ báo cáo cho producer rằng bộ đệm không đầy.

ĐỒNG BỘ HÓA LUỒNG VỚI SỰ KIỆN (EVENT)

- Sự kiện (Event) là các đối tượng được sử dụng để liên lạc giữa các luồng.
- Một luồng chờ tín hiệu trong khi một luồng khác xuất tín hiệu đó hay một luồng sẽ phát ra một sự kiện (event) và các luồng khác đợi sự kiện đó.
- Về cơ bản, một đối tượng sự kiện quản lý một cờ có thể được đặt thành **true** bằng phương thức **set()** và đặt lại thành **false** bằng phương thức **clear().**
- Tiến trình gọi phương thức wait() bị block cho đến khi cờ này có giá trị True.

ĐỒNG BỘ HÓA LUỒNG VỚI SỰ KIỆN (EVENT)

- **Ví dụ:** bài toán producer/consumer (sản-xuất/tiêu-thụ)



Thread synchronization with event objects

ĐỒNG BỘ HÓA LUỒNG VỚI SỰ KIỆN (EVENT)

- **Ví dụ:** bài toán producer/consumer (sản-xuất/tiêu-thụ) → sử dụng event thay vì condition
  - Producer ghi vào bộ đệm → thiết lập set() sự kiện (event) - cờ true → thông báo consumer + clear () sự kiện - cờ false.
  - Consumer cũng truyền một Event cho hàm khởi tạo → consumer sẽ bị block (khi gọi phương thức wait()), chờ cho đến khi sự kiện được phát ra - có dữ liệu mới được thêm vào.
  - **Lưu ý**: cờ được clear() - thiết lập false khi khởi tạo Event.

LƯU Ý

- Song Song dựa trên luồng
- Các cơ chế đồng bộ
  - Lock
  - Rlock
  - Semaphore
  - Điều kiện
  - Sự kiện