

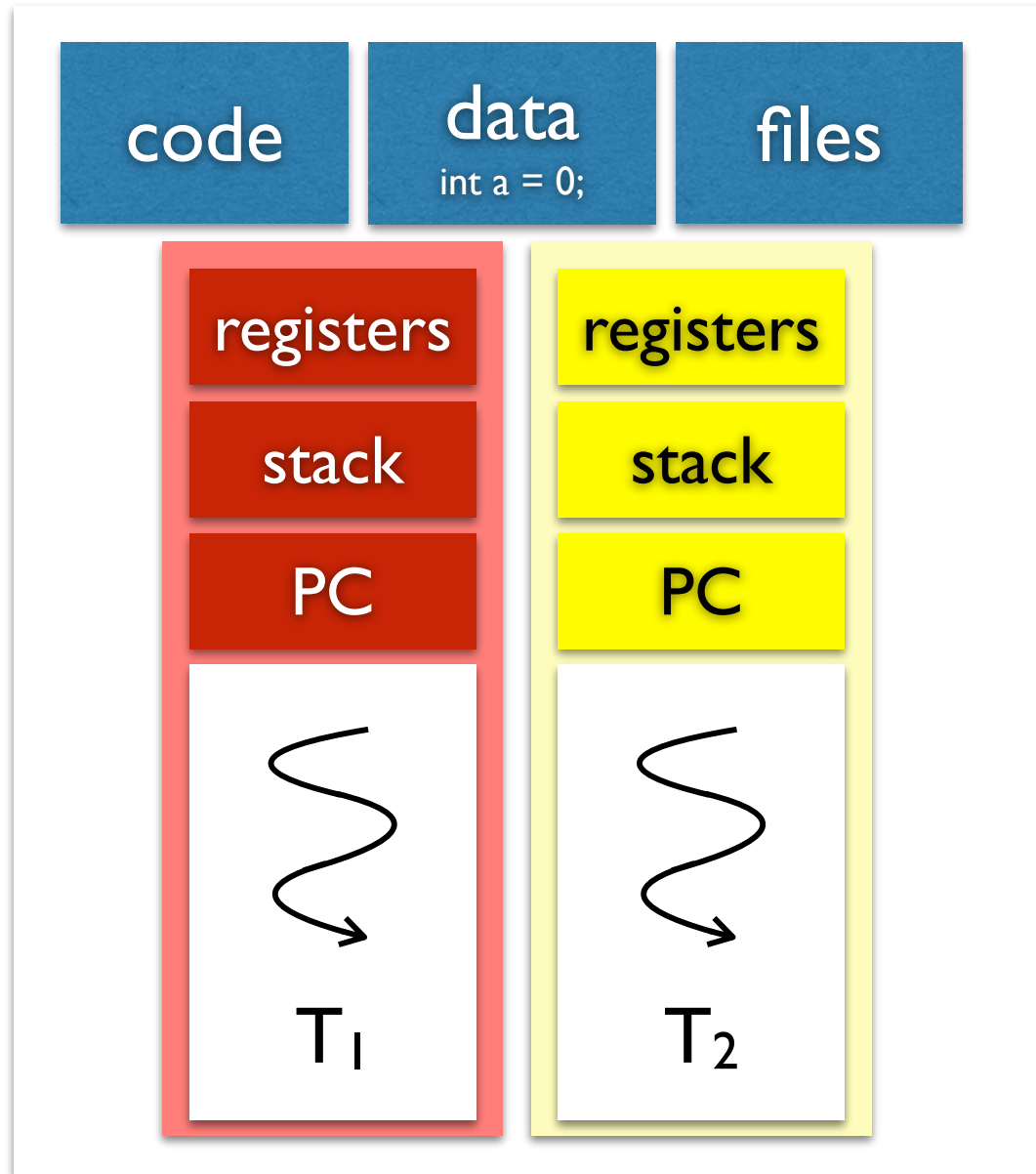
Synchronization

CSCI 315 Operating Systems Design
Department of Computer Science

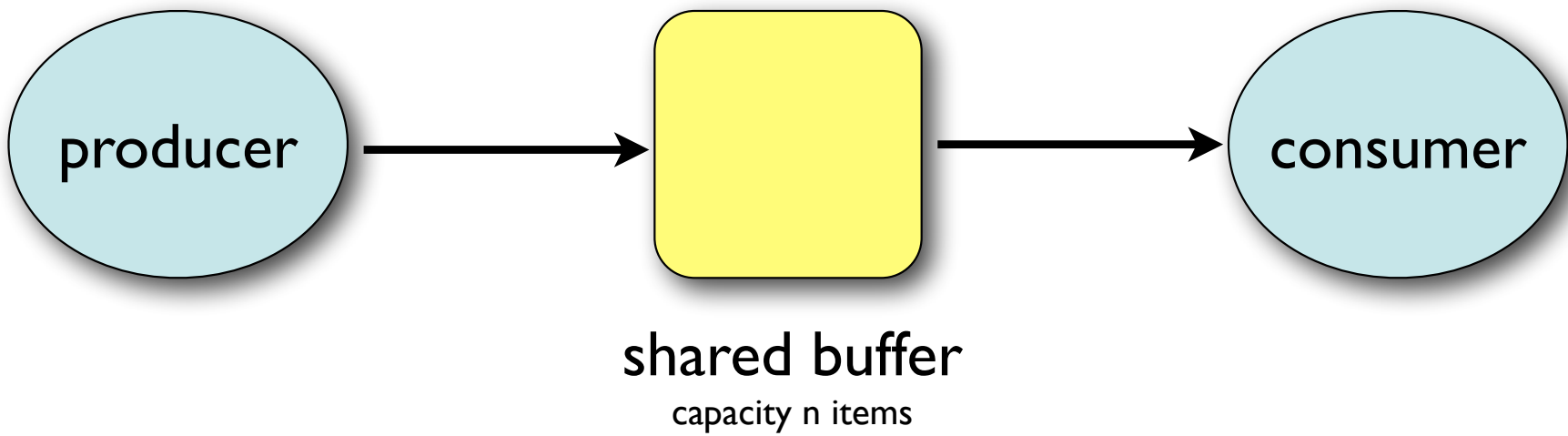
Notice: The slides for this lecture are based on those for *Operating Systems Concepts, 10th ed.*, by Silberschatz, Galvin, and Gagne. Many of the illustrations contained in this presentation come from this source.



A Tale of a Shared Variable



multithreaded process



Race Condition

A **race** occurs when the correctness of a program depends on one thread reaching point X in its control flow before another thread reaches point y .

Races usually occurs because programmers assume that threads will take some particular trajectory through the execution space, forgetting **the golden rule that threaded programs must work correctly for any feasible trajectory.**

Computer Systems

A Programmer's Perspective

Randal Bryant and David O'Hallaron

The Synchronization Problem

- **Concurrent access to shared data** may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the **“orderly” execution of cooperating processes.**

The Critical-Section Problem Solution

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (Assume that each process executes at a nonzero speed. No assumption concerning relative speed of the N processes.)

Typical Process P_i

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Peterson's Solution

```
int turn;  
boolean flag[2];
```

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    critical section  
    flag[i] = FALSE;  
    remainder section  
} while (TRUE);
```

Using Locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

Atomic



1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Synchronization Hardware

- Many systems provide hardware support for critical section code.
- Uniprocessors (could disable interrupts):
 - Currently running code would execute without preemption.
 - Generally too inefficient on multiprocessor systems.
 - Operating systems using this not broadly scalable.
- Modern machines provide special **atomic** hardware instructions:
 - Test memory word and set value.
 - Swap the contents of two memory words.



TestAndSet

```
boolean TestAndSet(boolean *target)
{
    boolean ret_val = *target;
    *target = TRUE;
    return ret_val;
}
```

Lock with TestAndSet

```
boolean lock = FALSE;  
  
do {  
    while (TestAndSet(&lock));  
    critical section  
    lock = FALSE;  
    remainder section  
} while (TRUE);
```

CompareAndSwap

```
int CompareAndSwap (int *value,  
    int expected, int new_value){  
  
    int temp = *value;  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

Lock with CompareAndSwap

```
int lock = 0;

do {
    while(CompareAndSwap(&lock, 0, 1) != 0);
    critical section
    lock = 0;
    remainder section
} while (TRUE);
```

How are we meeting requirements?

Do the solutions above provide:

- Mutual exclusion?
- Progress?
- Bounded waiting?

Semaphores

- **Counting semaphore** – integer value can range over an unrestricted domain.
- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement (also known as **mutex** locks).
- Provides **mutual exclusion**:

```
semaphore S(I); // initialized to I
```

```
wait(S); // or acquire(S) or P(S)
```

```
criticalSection();
```

```
signal(S); // or release(S) or V(S)
```

Semaphore Implementation

```
typedef struct {  
    int value;  
    struct process  
    *list;  
} semaphore;
```

Semaphore Implementation

Counting

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

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add process to S->list  
        block();  
    }  
}
```



Semaphore Implementation

Counting

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add process to S->list  
        block();  
    }  
}
```

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



Semaphore Implementation

Binary

```
signal(binary_semaphore *S) {  
    if (!empty(S->list)) {  
        remove a process P from S->list  
        wakeup(P)  
    } else {  
        S->value = 1;  
    }  
}
```

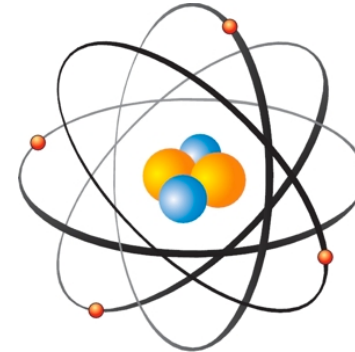
```
wait(binary_semaphore *S) {  
    if (S->value == 1) {  
        S->value = 0;  
    } else {  
        add process to S->list  
        block();  
    }  
}
```



Semaphore Implementation

Binary

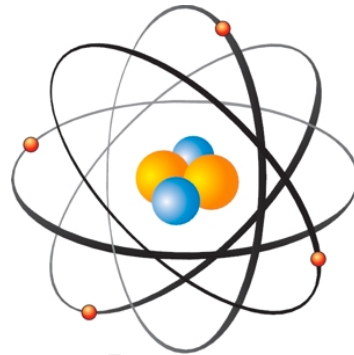
```
wait(binary_semaphore *S) {  
    if (S->value == 1) {  
        S->value = 0;  
    } else {  
        add process to S->list  
        block();  
    }  
}
```



```
signal(binary_semaphore *S) {  
    if (!empty(S->list)) {  
        remove a process P from S->list  
        wakeup(P)  
    } else {  
        S->value = 1;  
    }  
}
```

Semaphore Implementation

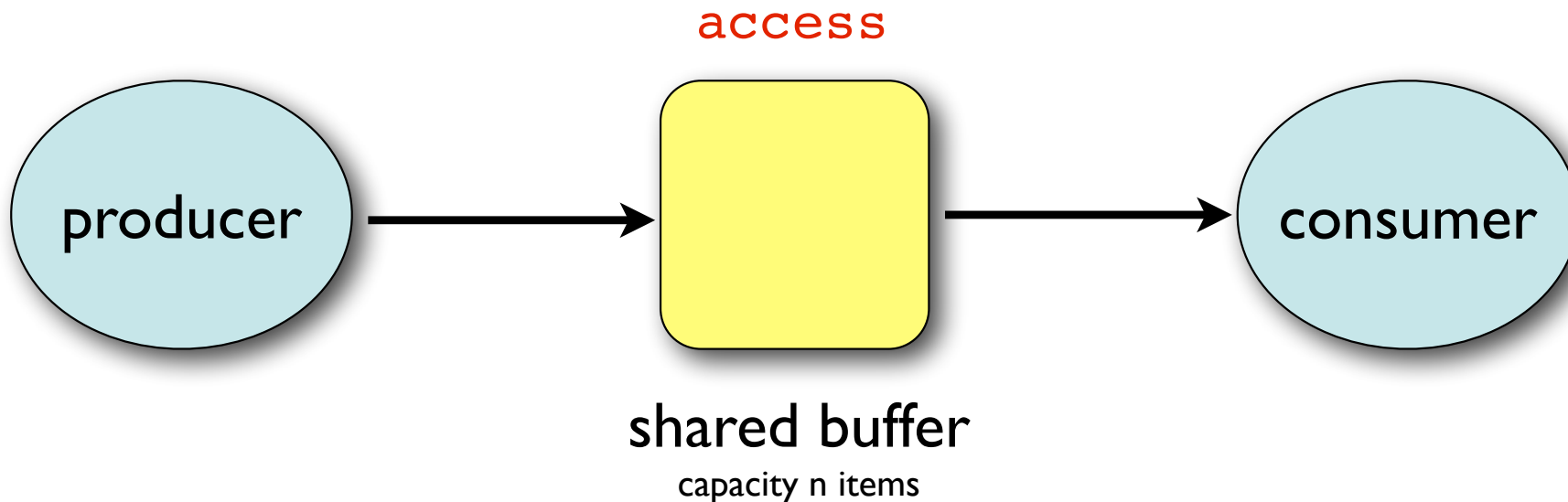
- Must guarantee that no two processes can execute **signal()** and **wait()** on the same semaphore “at the same time.”



- **The implementation becomes the critical section problem:**
 - Could now have busy waiting in critical section implementation
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
 - Applications may spend lots of time in critical section

The Bounded-Buffer Problem

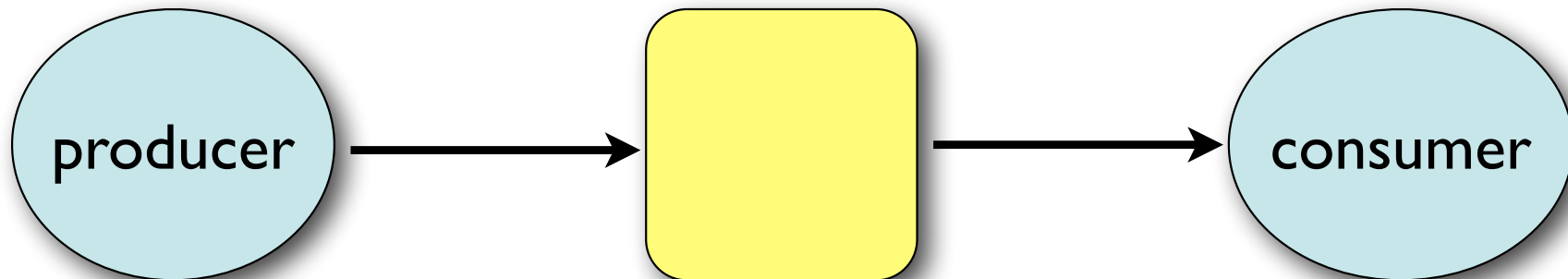
```
int n;  
mutex access;          init(&access,1);  
semaphore empty;      init(&empty,n);  
semaphore full;       init(&full,0);
```



The Bounded-Buffer Problem

```
do { // produce item and save
    wait(&empty);
    wait(&access);
    // add item and save
    signal(&access);
    signal(&full);
} while (true);
```

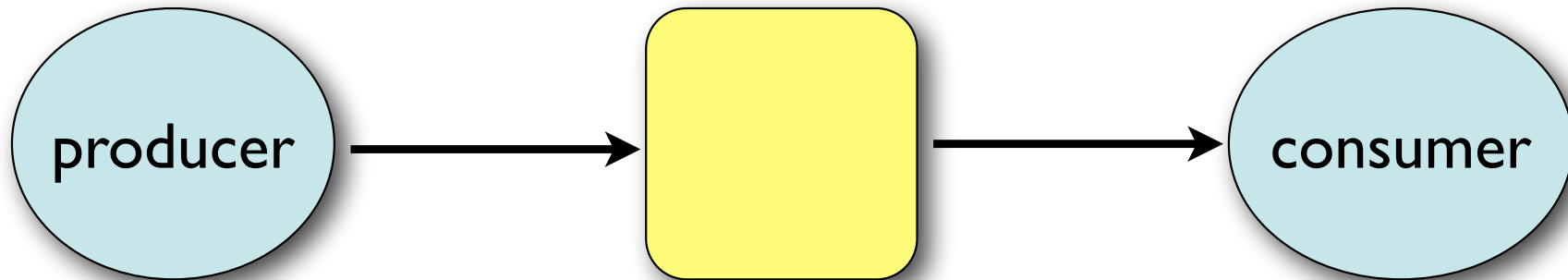
Producer



The Bounded-Buffer Problem

```
do { // produce item and save  
    wait(&empty);  
    wait(&access);  
    // add item and save  
    signal(&access);  
    signal(&full);  
} while (true);
```

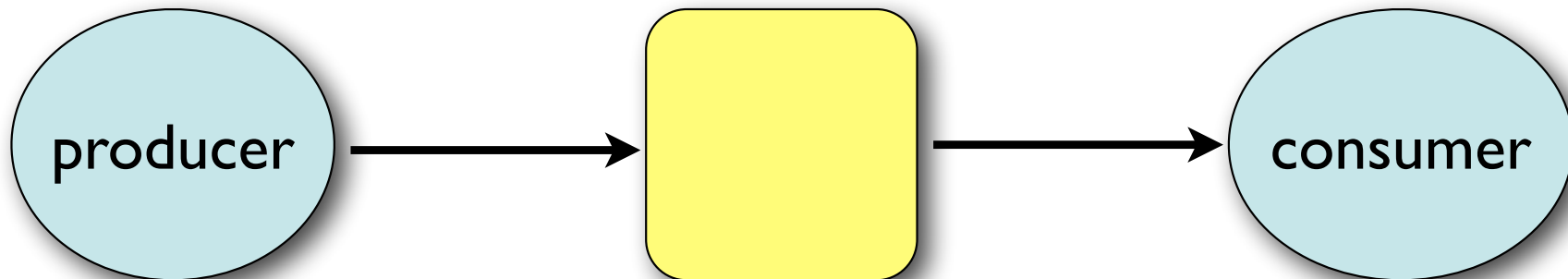
critical section



The Bounded-Buffer Problem

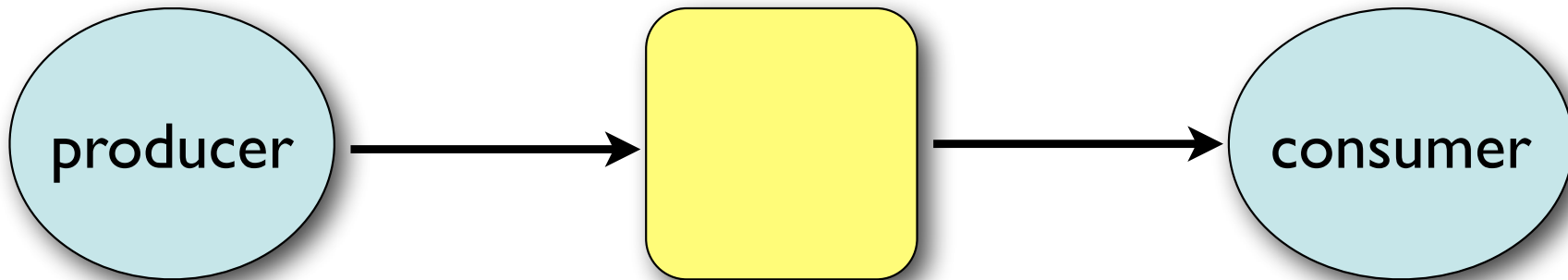
Consumer

```
do { wait(&full);  
    wait(&access);  
    // remove item and save  
    signal(&access);  
    signal(&empty);  
    // consume save item  
} while (true);
```



The Bounded-Buffer Problem

```
do { wait(&full);  
critical section  
    wait(&access);  
    // remove item and save  
    signal(&access);  
    signal(&empty);  
    // consume save item  
} while (true);
```



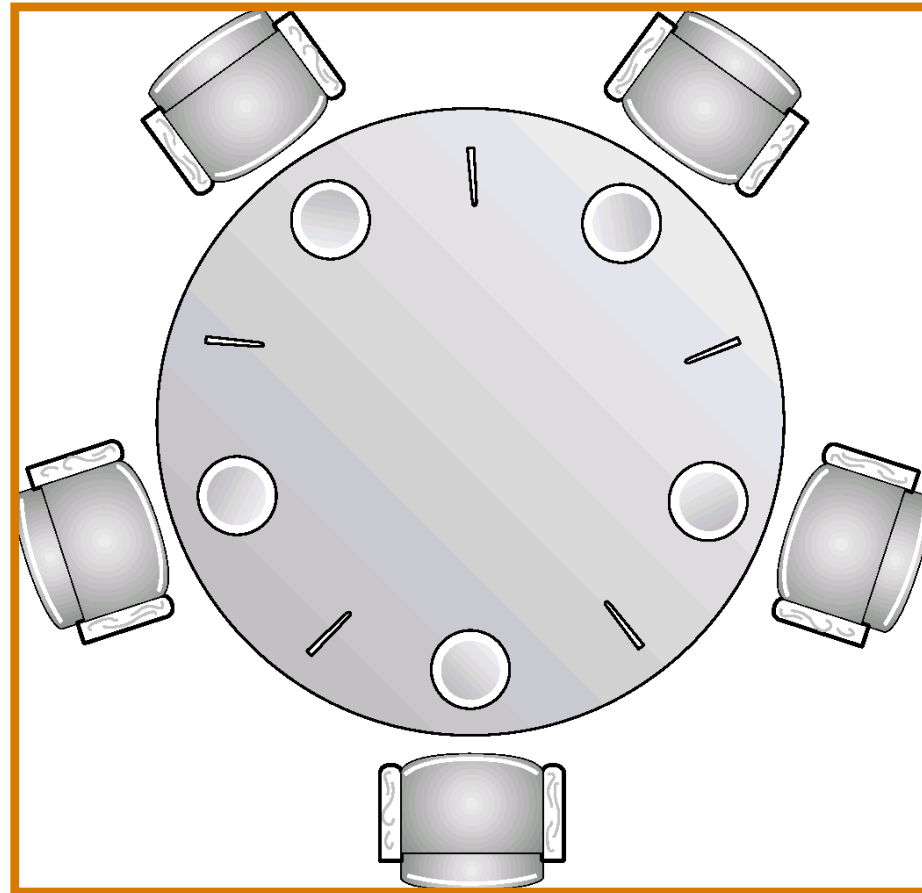
Deadlock and Starvation

- **Deadlock** – two or more processes are indefinitely blocked, waiting for an event that can be caused by only one of the waiting processes.
- Let **S** and **Q** be two semaphores initialized to **1**

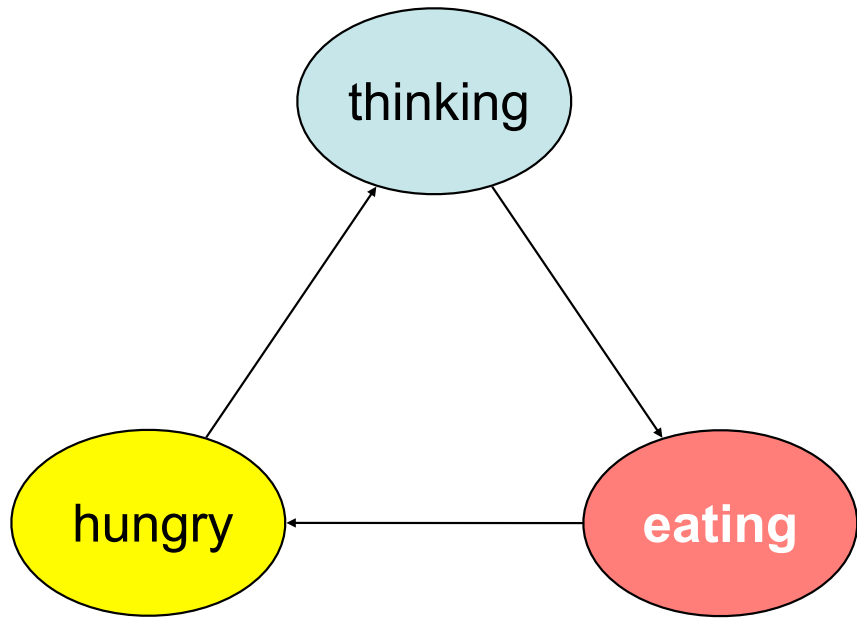
P_0	P_1
acquire(S);	acquire(Q);
acquire(Q);	acquire(S);
•	•
•	•
•	•
release(S);	release(Q);
release(Q);	release(S);

- **Starvation** – a process is trying to access a resource but its turn never comes.

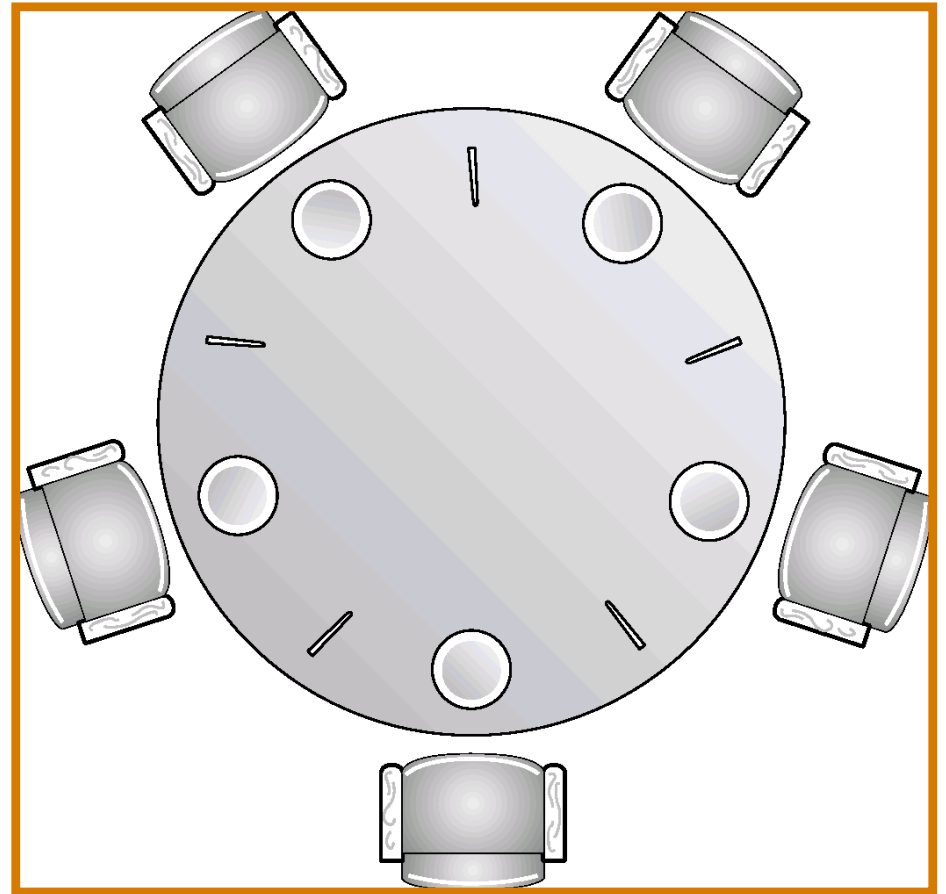
The *Dining-Philosophers* Problem



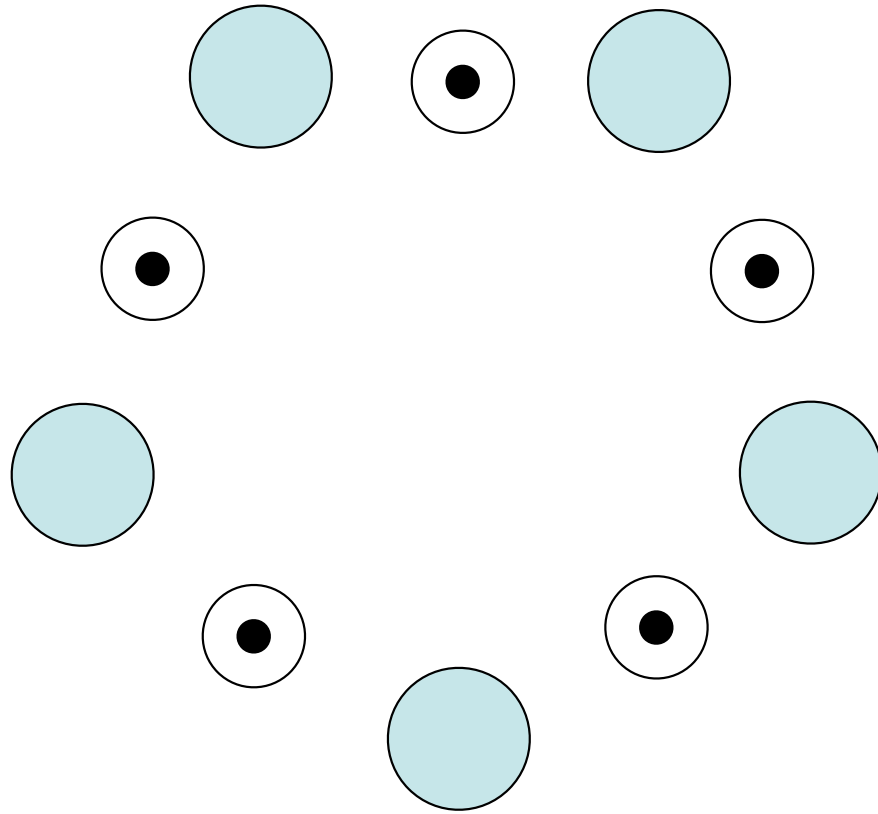
The *Dining-Philosophers* Problem



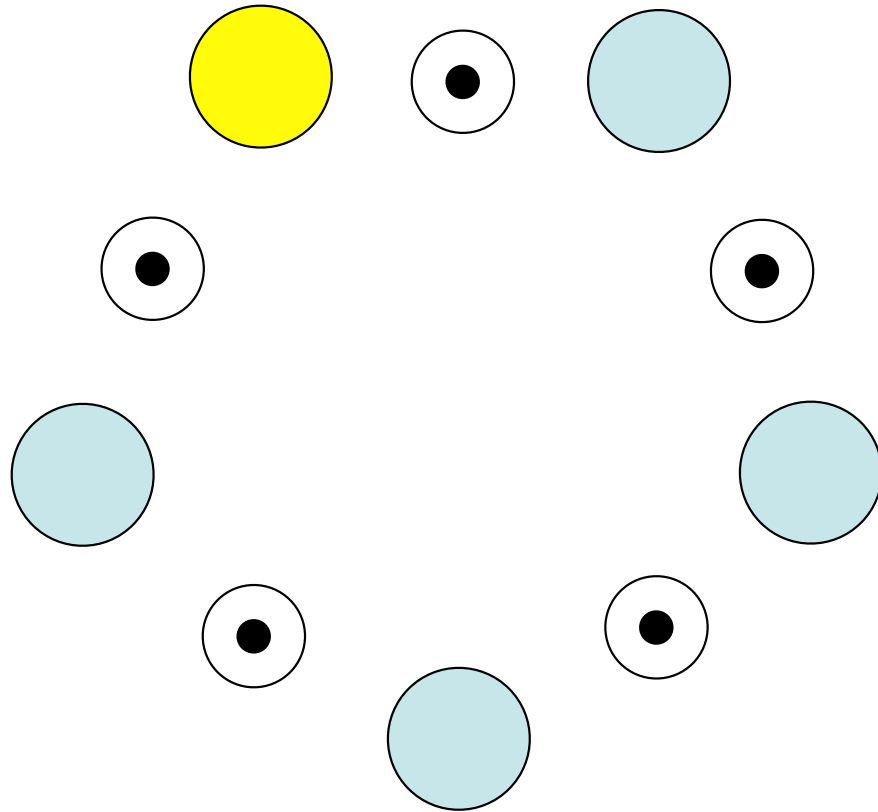
State diagram for a philosopher



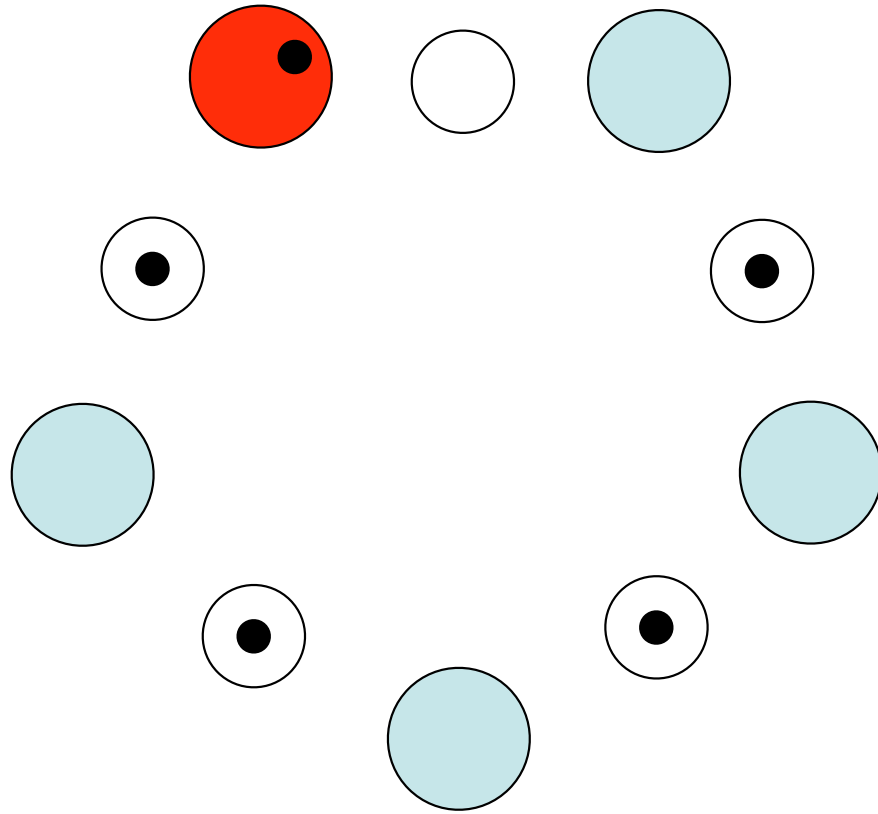
The *Dining-Philosophers* Problem



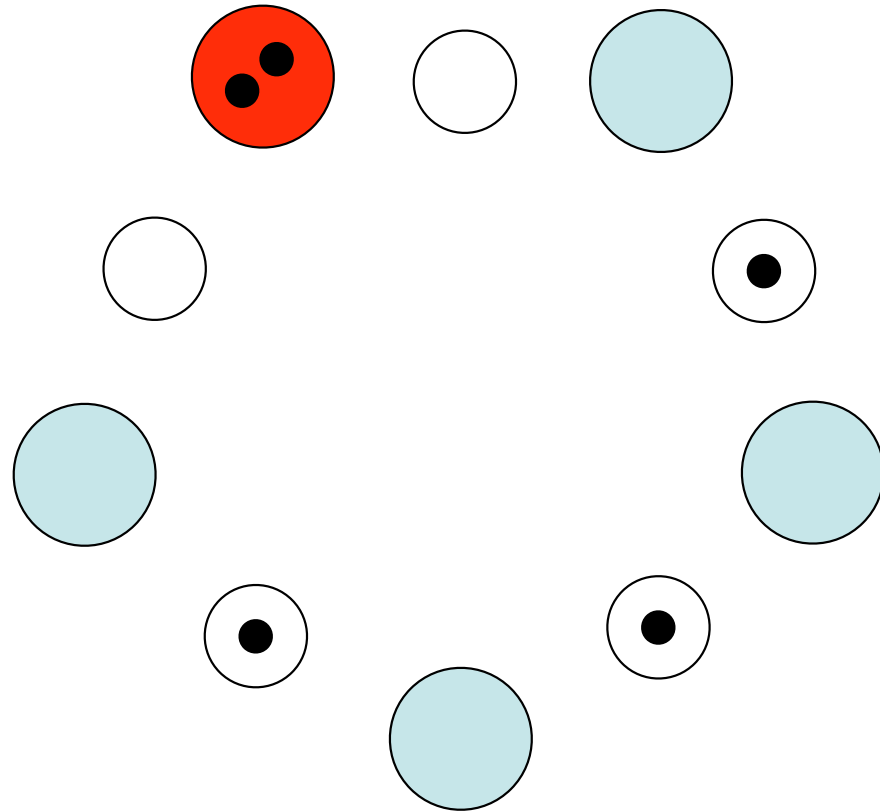
The *Dining-Philosophers* Problem



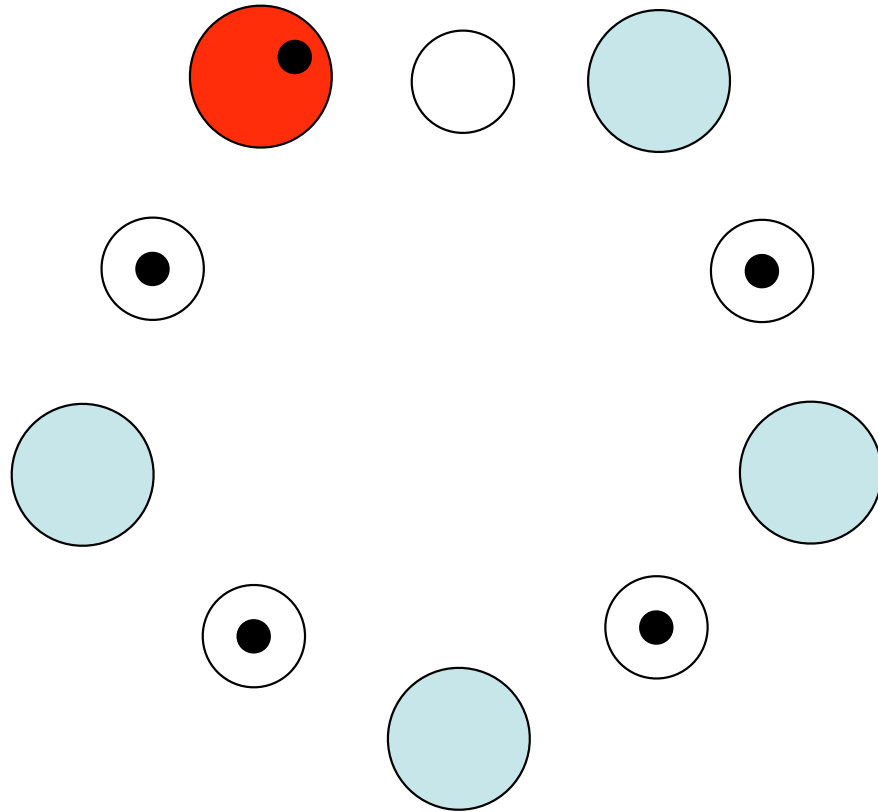
The *Dining-Philosophers* Problem



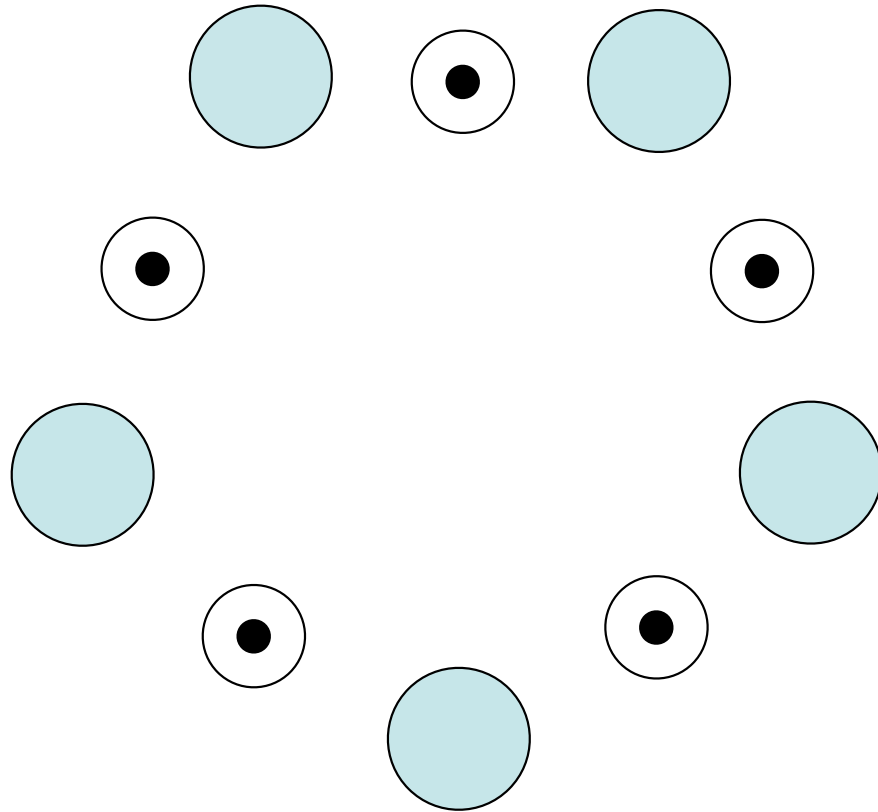
The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem



Limit to Concurrency

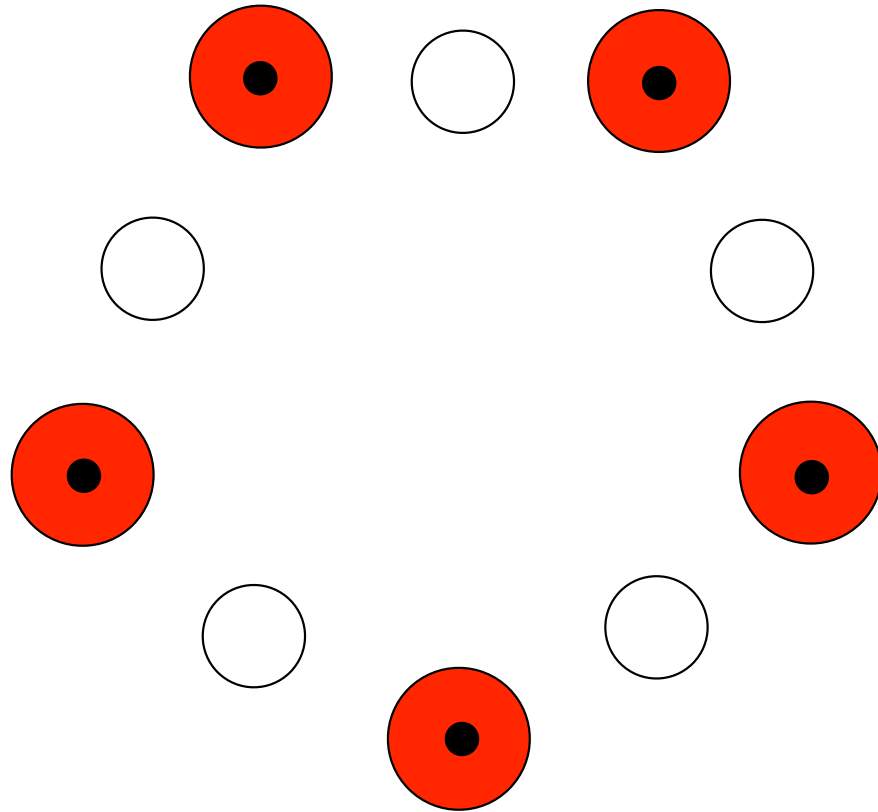
What is the maximum number of philosophers that can be eating at any point in time?

Philosopher's Behavior

- Grab chopstick on left
- Grab chopstick on right
- Eat
- Put down chopstick on right
- Put down chopstick on left

How well does this work?

The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem

Question: How many philosophers can eat at once? How can we generalize this answer for n philosophers and n chopsticks?

Question: What happens if the programmer initializes the semaphores incorrectly? (Say, two semaphores start out a zero instead of one.)

Question: How can we formulate a solution to the problem so that there is no deadlock or starvation?

Synonyms

wait	signal
P	V
acquire	release
down	up
lock	unlock

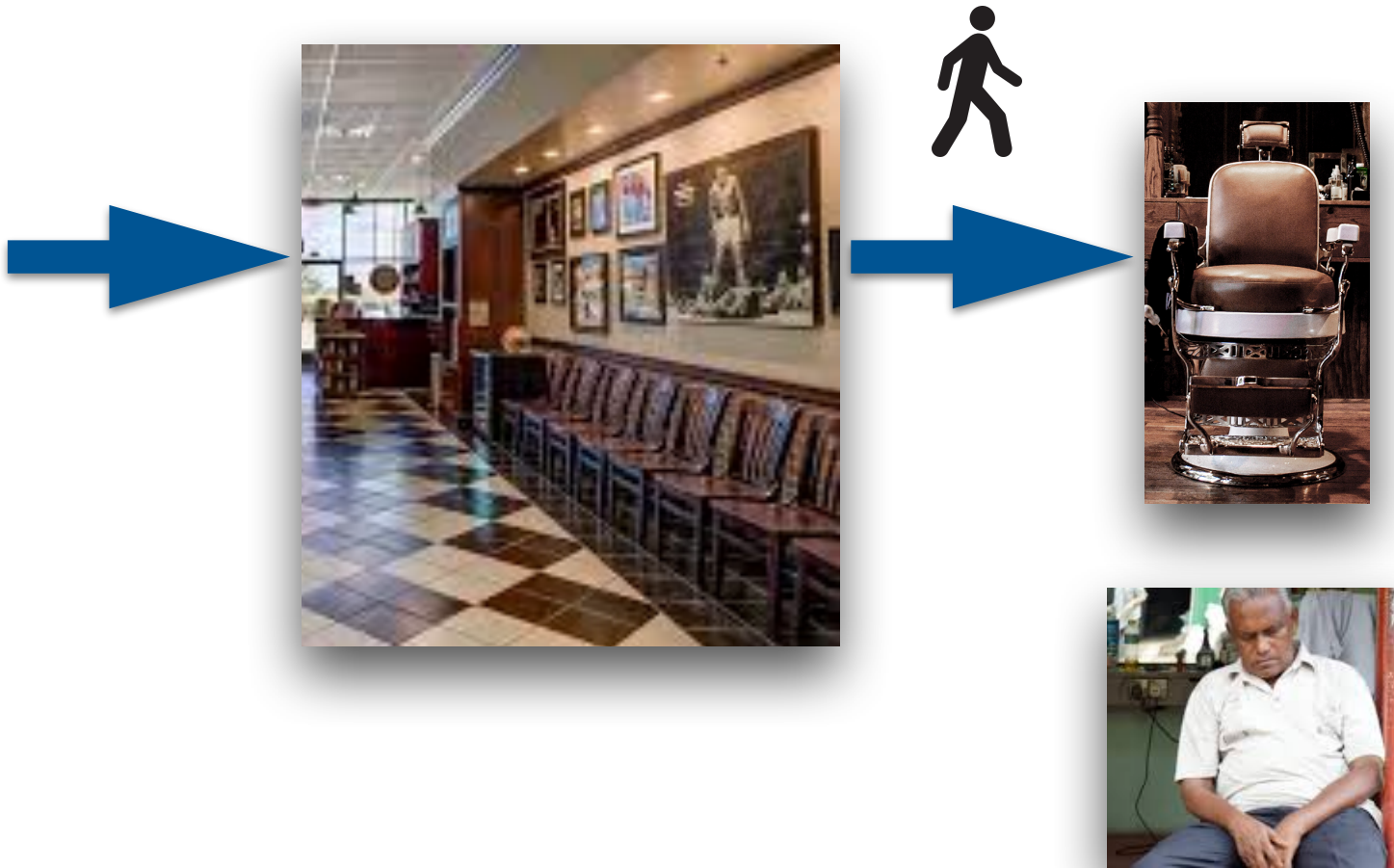
The Sleeping Barber Problem



The Sleeping Barber Problem



The Sleeping Barber Problem



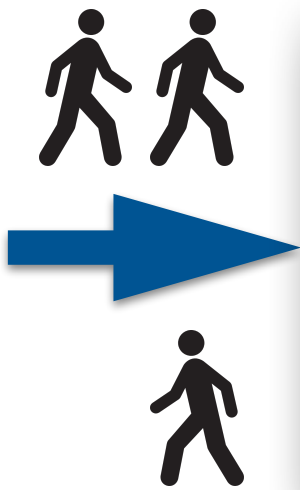
The Sleeping Barber Problem



The Sleeping Barber Problem



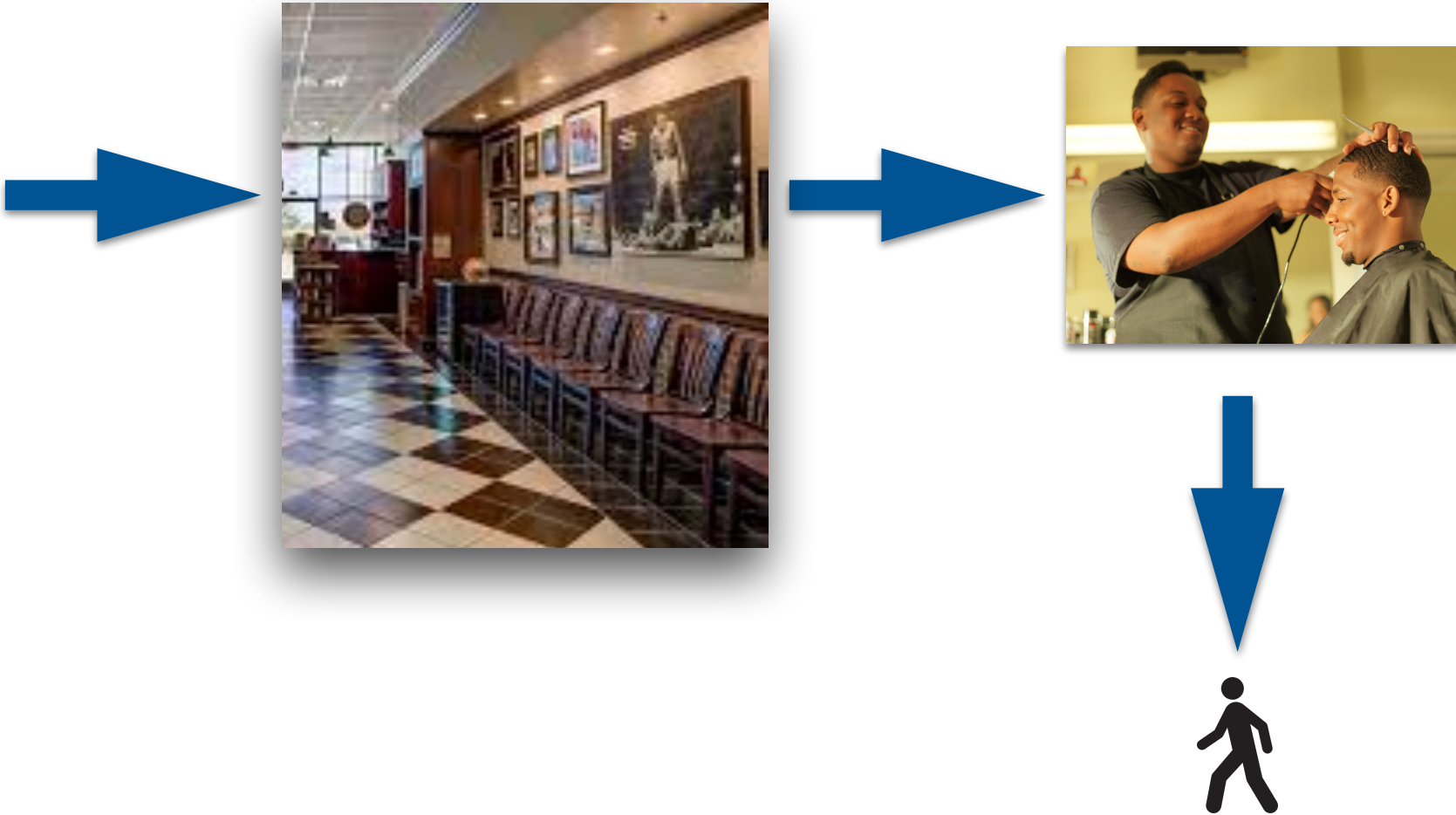
The Sleeping Barber Problem



The Sleeping Barber Problem



The Sleeping Barber Problem



The Sleeping Barber Problem



Solution

```
#define CHAIRS 5
semaphore customers = 0;
semaphore barbers = 0;
mutex access = 1;
int waiting = 0;
```

```
void barber(void)
{
    while (TRUE) {
        wait(&customers);
        wait(&access);
        waiting = waiting - 1;
        signal(&barbers);
        signal(&access);
        cut_hair();
    }
}
```

Solution

```
#define CHAIRS 5
semaphore customers = 0;
semaphore barbers = 0;
mutex access = 1;
int waiting = 0;
```

```
void customer(void)
{
    down(&access);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        signal(&customers);
        signal(&access);
        wait(&barbers);
        get_haircut();
    } else {
        signal(&access);
    }
}
```

Solution

```
void barber(void)
{
    while (TRUE) {
        wait(&customers);
        wait(&access);
        waiting = waiting - 1;
        signal(&barbers);
        signal(&access);
        cut_hair();
    }
}
```

```
void customer(void)
{
    wait(&access);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        signal(&customers);
        signal(&access);
        wait(&barbers);
        get_haircut();
    } else {
        signal(&access);
    }
}
```

The Sleeping Barber Problem

What entities are we modeling?

What behaviors are we modeling?
(How do we synchronize behaviors?)

What are the shared structures?

The Sleeping Barber Problem

What entities are we modeling?

- Customers
- Barber
- Barber chair
- Waiting area

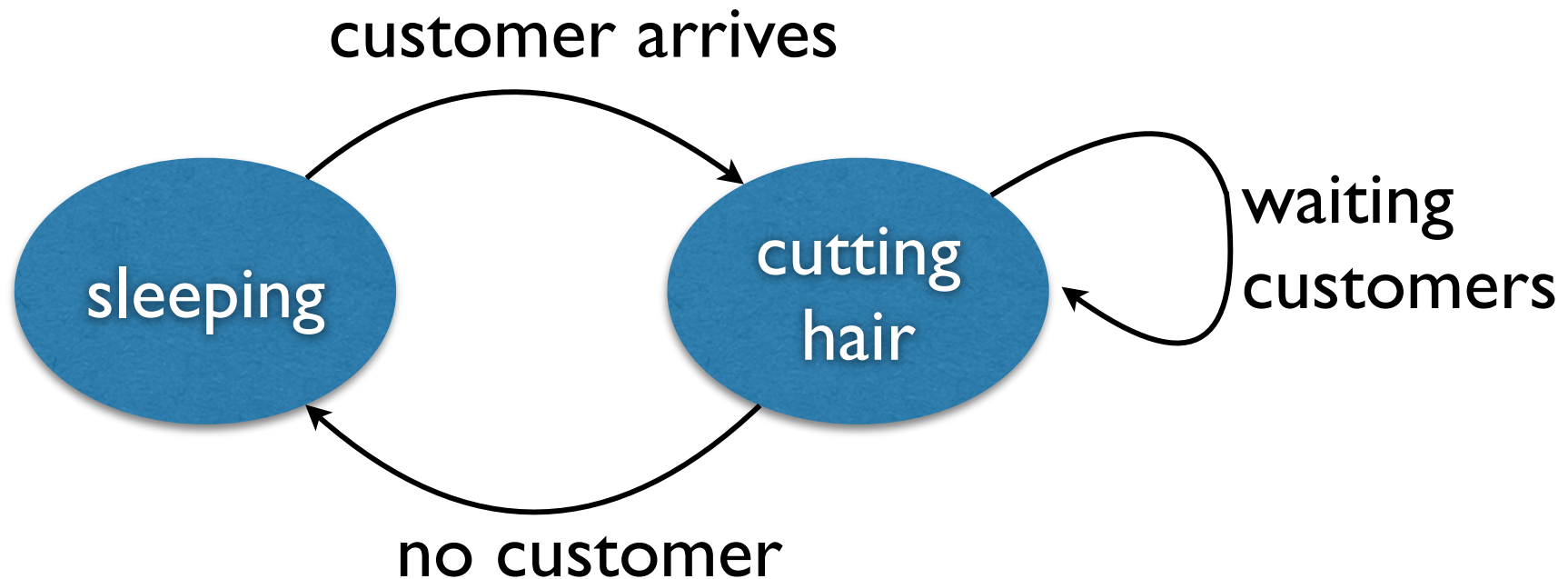
The Sleeping Barber Problem

What entities are we modeling?

Entity	Representation
Customer	process
Barber	process
Barber Chair	data
Waiting Area	data

The Sleeping Barber Problem

What behaviors are we modeling?
(How do we synchronize behaviors?)



Monitor

- Semaphores are low-level synchronization resources.
- A programmer's honest mistake can compromise the entire system (well, that is almost always true). We should want a solution that reduces risk.
- The solution can take the shape of high-level language constructs, as the **monitor** type:

```
monitor mName {  
    // shared variables  
    declaration  
    function P1 (...) {  
        ...  
    }  
    function Pn (...) {  
        ...  
    }  
    init code (...) {  
        ...  
    }  
}
```

A **function** can access only local variables defined within the monitor.

There cannot be concurrent access to procedures within the monitor (only one process/thread can be **active** in the monitor at any given time).

Condition variables: queues are associated with variables. Primitives for synchronization are **wait** and **signal**.

Monitor

- The *monitor* is a “*high level of abstraction*” construct.
- Placing shared data in a monitor guarantees mutually exclusive access to that data.
- Access to the data inside the monitor happens exclusively through the monitor’s procedures.

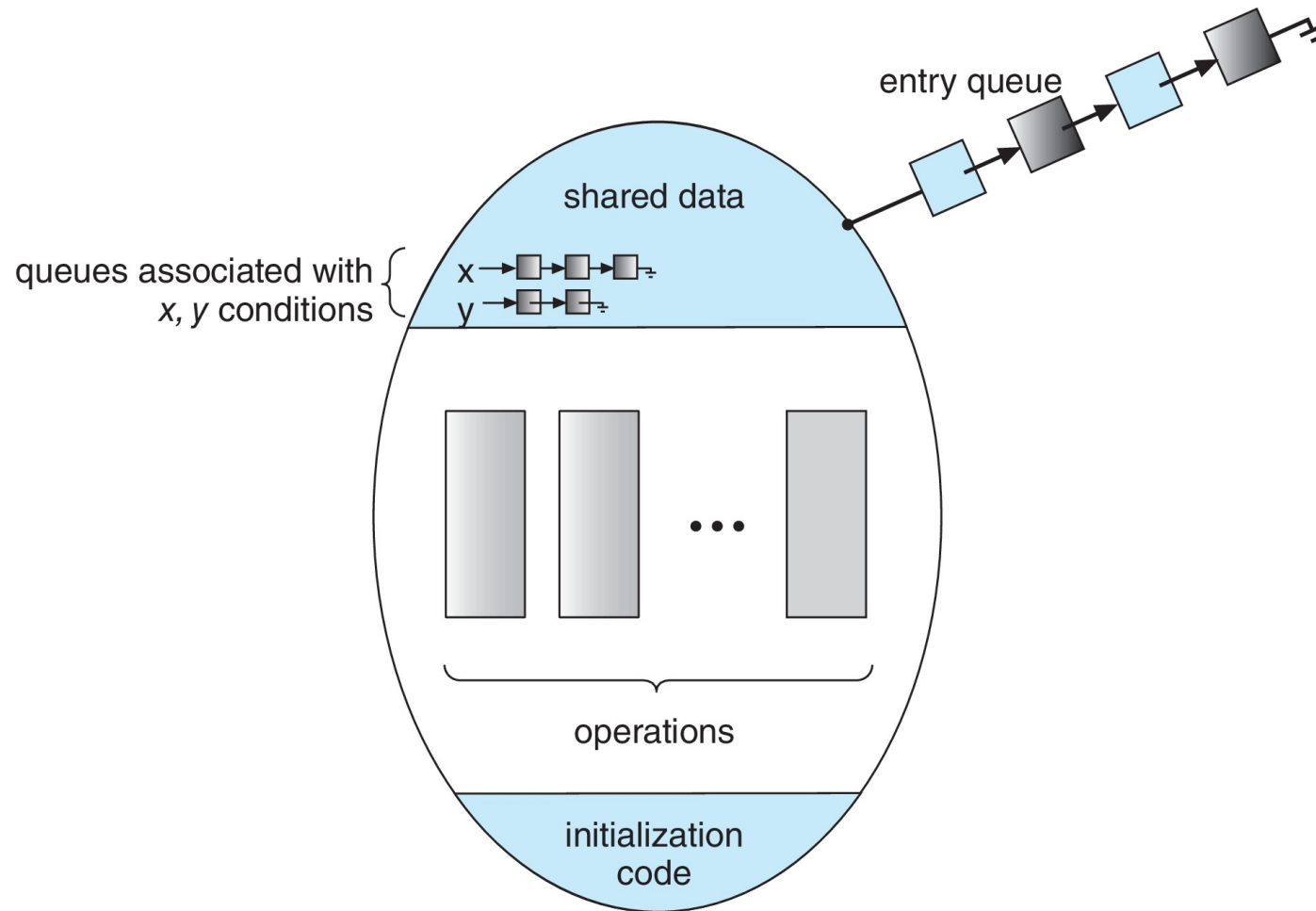
Benefit

The programmer is protected from mistakes in coding solutions to the critical section problem.

Restriction

The monitor doesn’t directly solve all synchronization problems. For that, the programmer needs to use condition variables.

Monitor

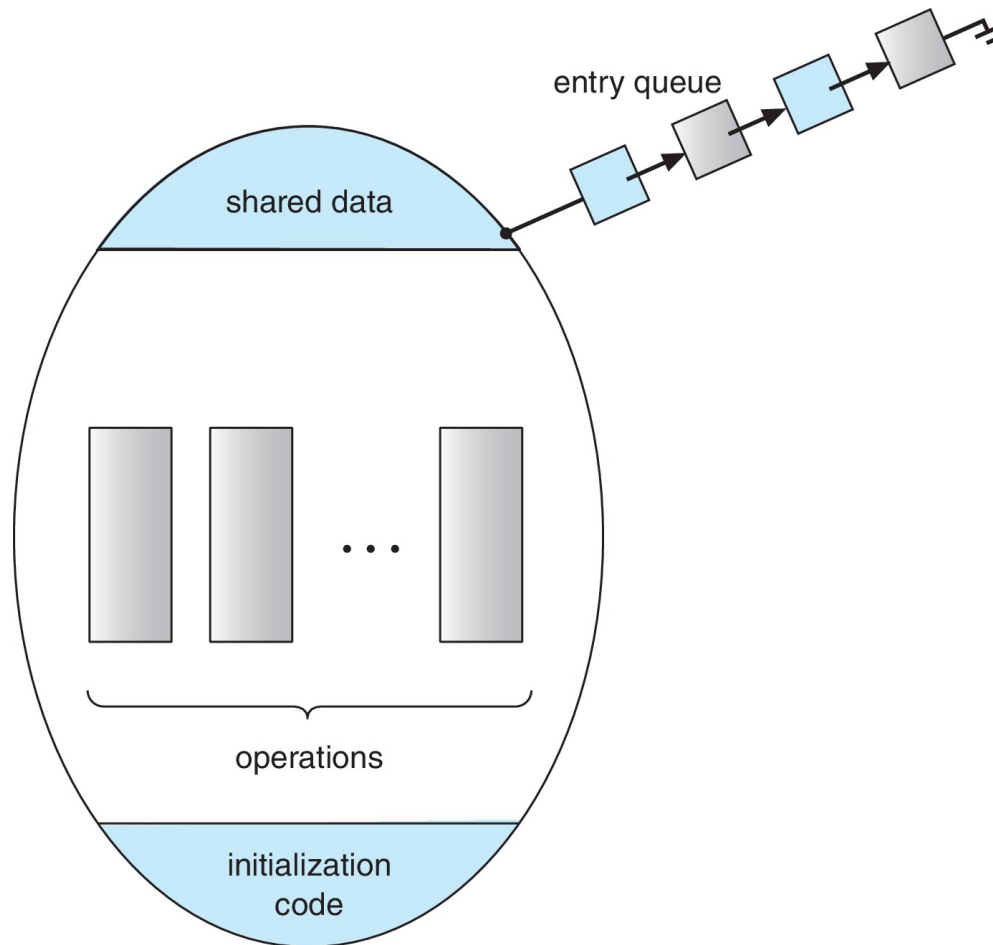


Condition Variables

```
condition x, y;
```

- Two operations are allowed on a condition variable:
 - ➔ **x.wait()** – a process that invokes the operation is suspended until x.signal()
 - ➔ **x.signal()** – resumes one of processes (if any) that invoked x.wait()
 - If no **x.wait()** on the variable, then it has no effect on the variable

Monitor



Practical Applications of Classical Problems

Producers and Consumers

Message Queues

Producers = Services publishing messages
Consumers = Services subscribing to them
Shared buffer = Middleware (Kafka, RabbitMQ, ZeroMQ, etc.)

This is the heart of modern cloud architectures. Producers and consumers are decoupled, can scale independently, and synchronize through message queues.

Data Acquisition in IoT / Sensor Systems

Producers = Sensor sampling threads
Consumers = Data analysis/storage threads
Shared buffer = Memory queue

Sensors may generate data at different rates than analysis/storage threads can process. The buffer smooths mismatches.

Practical Applications of Classical Problems

The Sleeping Barber

Job Scheduling in an OS

Barber = CPU core (or processor)

Chairs = Ready queue

Customers = Processes needing CPU time

When a process is ready:

If the CPU is idle, it runs immediately.

If the CPU is busy, it waits in the ready queue.

If the ready queue is full (in embedded systems, real-time systems, or hardware schedulers), new jobs are blocked or discarded.

Thread Pool in a Web Server

Barber = Worker thread

Chairs = Task queue slots

Customers = Incoming client requests

When HTTP requests come in:

If a thread is free, it processes the request immediately.

If all threads are busy, the request waits in a bounded queue. If the queue is full, requests may be dropped or asked to retry. This is almost a direct implementation of the barber problem solution.

When a process is ready:

If the CPU is idle, it runs immediately. If the CPU is busy, it waits in the ready queue. If the ready queue is full (in embedded systems, real-time systems, or hardware schedulers), new jobs are blocked or discarded.