Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

a.a. 2026/2027

Marco Raoul Marini

marini@di.uniroma1.it

Gabriele Tolomei

tolomei@di.uniroma1.it

Lecture 3.4: Concurrency IV

Semaphores: Binary and Counting

Opening Thought

> *"Dijkstra and colleagues invented the semaphore as a single primitive for all things related to synchronization; as you will see, one can use semaphores as both locks and condition variables."*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 31*

So far we have used two different tools: **locks** (mutual exclusion) and **condition variables** (ordering). Today we meet the **semaphore**, introduced by Edsger Dijkstra in 1965: one abstraction that can express both, and much more.

Lecture Roadmap

This lecture is organized into **8 major sections**:

1. **Recap & Motivation** — Two primitives for two problems; a counter with a wait queue
2. **Semantics and the POSIX API** — `sem_wait()`, `sem_post()`, the invariant, `sem_t`
3. **Binary Semaphores** — Initial value 1: mutual exclusion
4. **Semaphores for Ordering** — Initial value 0: signaling that is remembered
5. **Counting Semaphores** — Initial value N: pools of resources
6. **Producer-Consumer** — Three semaphores and the `wait()`-order trap (*deadlock preview*)
7. **Dining Philosophers** — The naive solution deadlocks (*deadlock preview*); one fix
8. **The Zemaphore** — Building a semaphore from a mutex and a CV

> *Extended material (named semaphores, Dijkstra's negative values, reader-writer lock, second Dining Philosophers fix, Tanenbaum, Zemaphore details, two exercises) is in the **Appendix** (A.1–A.5); section dividers point to it.*

Prerequisite Check

Before proceeding, ensure you recall:

- **Race conditions, critical sections, mutex API** (Lecture 3.1): why unprotected shared state is dangerous.
- **Locks** (Lecture 3.2): `lock()` / `unlock()`; a blocked thread sleeps instead of spinning.
- **Condition variables** (Lecture 3.3): `wait()`, `signal()`; Mesa semantics; the `while` loop.
- **Bounded buffer with CVs** (Lecture 3.3): `cv_empty`, `cv_full`, one mutex.
- **Thread states** (Lecture 3.1): RUNNING, READY, BLOCKED.

New today: semaphore, P/V (`wait` / `post`), binary and counting semaphore, Dining Philosophers, Zemaphore.

§ 1 — Recap & Motivation

Goal: see why a single counter-based primitive is attractive.

Recap: Two Primitives for Two Problems

	Mutex (3.2)	Condition variable (3.3)
Solves	mutual exclusion	ordering: wait until a condition holds
API	`lock()`, `unlock()`	`wait(&cv,&m)`, `signal(&cv)`, `broadcast(&cv)`
Needs	nothing else	a mutex **and** shared state, tested in a `while` loop
Memory	owner and locked/unlocked state	**none**: a `signal()` with no waiter is lost

Real programs need both: `pthread_mutex_t m` + `pthread_cond_t cv` + an `int state` that records what happened.

> ***Question:*** *can one simpler primitive replace the pair, and remember what happened?*

The Idea: A Counter with a Wait Queue

Think of a semaphore as a **box of tickets**:

- `wait`: take a ticket. If the box is empty, **sleep** until someone returns one.
- `post`: put a ticket back (and wake a sleeper, if any).
- Dijkstra's names: **P** = *proberen* ("to try"), **V** = *verhogen* ("to increase").

The **initial number of tickets** decides which problem it solves:

Initial tickets	Meaning	Role	Section
1	one thread at a time	mutual exclusion	§ 3
0	"not yet happened"	ordering	§ 4
N	N identical resources	resource pool	§ 5

§ 2 — Semantics and the POSIX API

Goal: define `wait`/`post` precisely and know the POSIX calls.

📎 *Extended material → Appendix A.1 (named semaphores), A.2 (Dijkstra's negative values)*

Semaphore: Definition

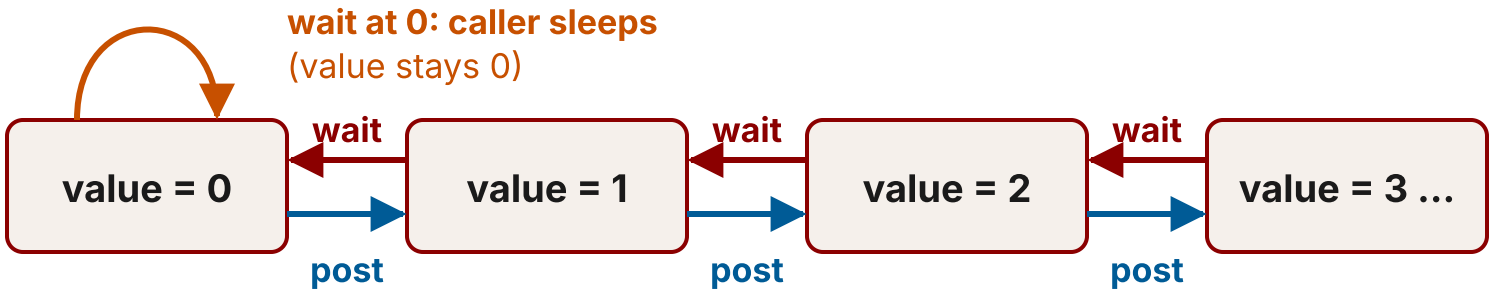
A semaphore is an integer $S \geq 0$ plus a wait queue, changed only by two atomic operations.

```
sem_wait(S):  // P / down              sem_post(S):  // V / up
    while (S == 0)                         S = S + 1
        sleep on S's queue                 if queue not empty: wake one sleeper
    S = S - 1                              // never blocks
```

- **Invariant:** $S = \text{init} + \#\text{posts} - \#\text{completed waits} \geq 0$.
- A sleeping thread is `BLOCKED` (no CPU); `post` makes it `READY`; it then **re-checks** `S`.
- Which sleeper wakes is **unspecified** (not necessarily FIFO). Any thread may call `post`.

Convention used here: the POSIX one, S never negative. Dijkstra's negative-value model: Appendix A.2.

Semaphore Value: What Each Operation Does



wait, value > 0:
value decreases by 1,
the caller continues

wait, value = 0:
the caller sleeps in the
queue; value stays 0

post:
value increases by 1;
if someone sleeps, wake one

The value is "tickets left". It only reaches 0 by `wait`; while it is 0, every further `wait` sleeps.

POSIX API: `sem_t`

```c
#include <semaphore.h>

sem_t s;
int sem_init(sem_t *sem, int pshared, unsigned value);
                  // pshared = 0: threads of one process; value = initial S
int sem_wait(sem_t *sem);     // P: block while value is 0, then decrement
int sem_post(sem_t *sem);     // V: increment, wake one waiter if any
int sem_getvalue(sem_t *sem, int *sval);  // debugging only: stale at once
int sem_destroy(sem_t *sem);
```

- All return 0 on success, -1 on error (`errno` set). Compile with `-pthread`.
- Unlike `pthread_mutex_unlock`, `sem_post` has **no owner check**.

✅ Checkpoint 1 — Semantics and the POSIX API

Test your understanding:

1. In the ticket picture, what do `wait` and `post` do? What do P and V stand for?
2. What does `sem_wait()` do if the value is 0? If it is 3? What does `sem_post()` do if nobody sleeps?
3. Initial value 5, three threads call `sem_wait()`. Value afterwards? Does anyone block?
4. Initial value 2, five threads call `sem_wait()`. How many proceed, how many block? What is the value?
5. State the invariant. Can the value ever be negative in POSIX?

§ 3 — Binary Semaphores: Mutual Exclusion

Goal: use a semaphore as a lock.

Binary Semaphore: Initial Value 1

> ***Problem:*** *keep two threads out of the critical section together (as in 3.1/3.2).* ***Idea:*** *one ticket: whoever holds it is inside.*

Step	Event	Value	Effect
0	initialization	1	lock free
1	A: `wait`	$1 \to 0$	A enters
2	B: `wait`	stays 0	B **sleeps**
3	A: `post`	$0 \to 1$	B woken
4	B retakes the ticket	$1 \to 0$	B enters
5	B: `post`	$0 \to 1$	lock free

> *Between steps 3 and 4 a third thread could grab the ticket first; B would simply sleep again (the `while` of the definition).*

Using a Binary Semaphore as a Mutex

```c
#include <semaphore.h>   // plus stdio.h, pthread.h
int counter = 0;
sem_t mutex;

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        sem_wait(&mutex);    // acquire (lock)
        counter++;           // critical section
        sem_post(&mutex);    // release (unlock)
    }
    return NULL;
}

int main(void) {
    sem_init(&mutex, 0, 1);  // init to 1: binary semaphore
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);  pthread_join(t2, NULL);
    printf("counter = %d\n", counter);   // always 2000000
}
```

✅ Checkpoint 2 — Binary Semaphores

Test your understanding:

1. Binary semaphore (init 1), threads T1, T2, T3 call `sem_wait()` in this order. Value after each call? Who blocks?
2. What happens if the initial value is 0? If it is 2?
3. Thread A calls `sem_wait()`, thread B calls `sem_post()`. Valid? Would it be valid with `pthread_mutex_t`?
4. Write the critical-section pattern with a semaphore and compare it to the `pthread_mutex_t` version.

§ 4 — Semaphores for Ordering

Goal: make one thread wait for an event in another, with no lost signals.

Ordering: Parent-Child Join

> ***Problem:*** *the parent must continue only* ***after*** *the child has finished. In 3.3 this needed a mutex, a CV and a* `done` *flag.* ***Idea:*** *a semaphore initialized to* ***0****: "no ticket until the event happens".*

- Initialize `done` to **0**.
- Child calls `sem_post(&done)` when finished: *"I am done."*
- Parent calls `sem_wait(&done)`: *"Wait until the child is done."*
- Spinning on `while (!done);` would waste CPU; a mutex alone cannot wait for an event.

Unlike a CV `signal()`, **the post is remembered** as a ticket, even if nobody waits yet.

Full Implementation: Semaphore-Based Join

```c
sem_t done;

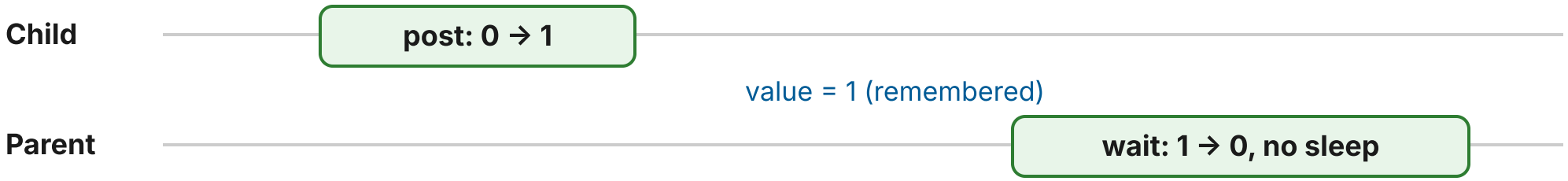
void *child(void *arg) {
    printf("child: working\n");
    sem_post(&done);          // "I am done": value 0 -> 1
    return NULL;
}

int main(void) {
    sem_init(&done, 0, 0);   // 0 tickets: the event has not happened yet
    pthread_t tid;
    pthread_create(&tid, NULL, child, NULL);
    sem_wait(&done);          // sleeps until the child posts
    printf("parent: child done\n");
    pthread_join(tid, NULL); // reclaim the thread
    sem_destroy(&done);
}
```

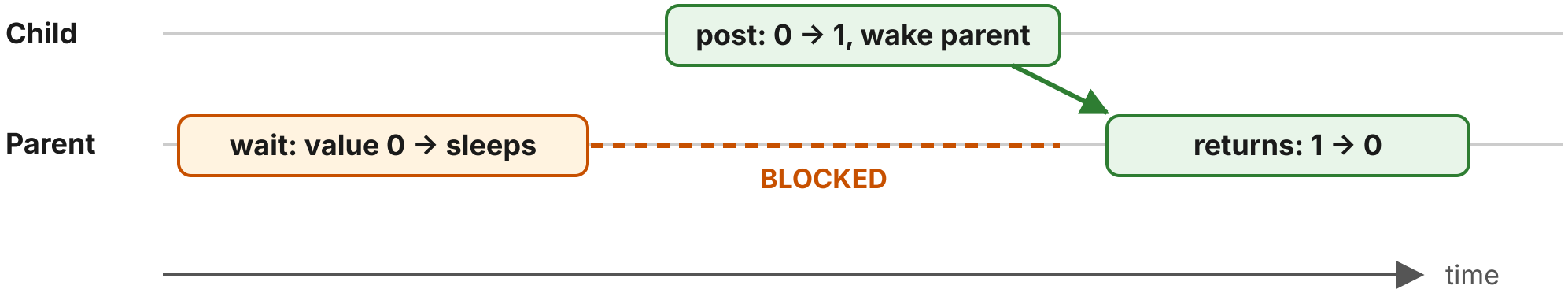
`pthread_join` *already orders thread termination; the pattern matters for events such as "data ready".*

Ordering: Both Interleavings Work

A. Child finishes first: the post is remembered



B. Parent reaches wait first: the parent sleeps



✅ Checkpoint 3 — Semaphores for Ordering

Test your understanding:

1. Why initialize to 0 for ordering, rather than 1?
2. Child finishes before the parent reaches `sem_wait()`. Does the parent block? Why not?
3. What does the semaphore remember that a CV does not?
4. Thread A must compute a value before thread B uses it. Write the pattern with only a semaphore.
5. Value after the parent returns from `sem_wait()` in the child-first case? In the parent-first case?

§ 5 — Counting Semaphores: Resource Management

Goal: allow up to N threads, not just one.

Counting Semaphore: Initialized to N

__Problem:__ a resource has N identical units (connections, workers). __Idea:__ start with N tickets; up to N threads pass `sem_wait()`, the $(N+1)$-th sleeps. A binary semaphore is the case $N = 1$.

```c
#define MAX_CONCURRENT 5
sem_t limiter;                              // sem_init(&limiter, 0, MAX_CONCURRENT);

void *do_work(void *arg) {
    sem_wait(&limiter);    // take one slot
    /* ... at most 5 threads here at once ... */
    sem_post(&limiter);    // give it back
    return NULL;
}
```

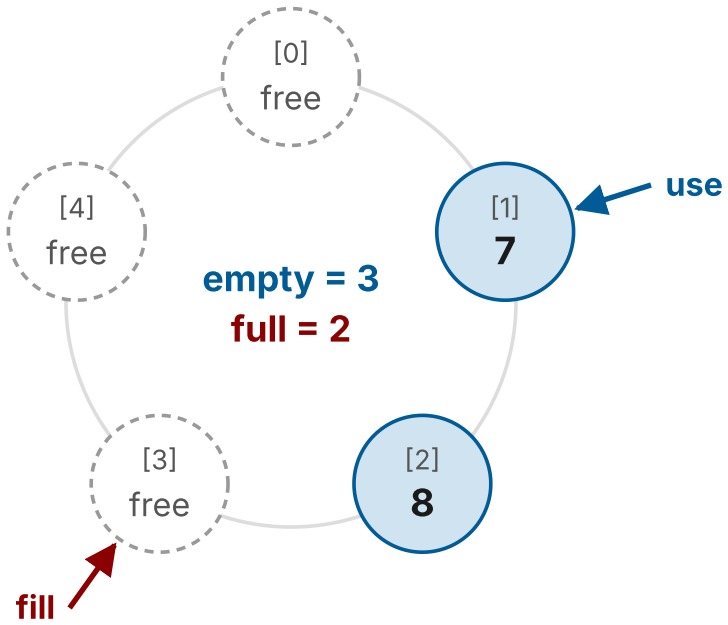
- **Trace, 8 threads:** T1–T5 pass (value $5 \to 0$); T6–T8 **sleep** (value stays 0). T1 posts: $0 \to 1$, one sleeper (say T6; the wake order is unspecified) wakes and retakes it: $1 \to 0$.
- Also used for **bounded buffers** (next section). Dijkstra's negative-value view: Appendix A.2.

§ 6 — Producer-Consumer with Semaphores

Goal: rebuild the bounded buffer of 3.3 with semaphores, and see how a wrong `wait()` order deadlocks.

Bounded Buffer: Which Counters Do We Need?

> ***Problem (as in 3.3):*** *producers block when the buffer is full, consumers when it is empty.* ***Idea:*** *let two counting semaphores be the two conditions.*



empty (init MAX)
free slots. Producer waits on it.

full (init 0)
items ready. Consumer waits on it.

mutex (init 1)
protects buffer, fill, use.

`fill` / `use`: next slot to write/read. `empty + full = MAX` when nobody is mid-operation.

Bounded Buffer: Setup

```
#define MAX 5
int buffer[MAX], fill = 0, use = 0;
sem_t empty, full, mutex;

void init(void) {
    sem_init(&empty, 0, MAX);   // all MAX slots free
    sem_init(&full,  0, 0);     // no items yet
    sem_init(&mutex, 0, 1);     // binary: protects buffer, fill, use
}
void put(int v) { buffer[fill] = v; fill = (fill + 1) % MAX; }
int  get(void)  { int v = buffer[use]; use = (use + 1) % MAX; return v; }
```

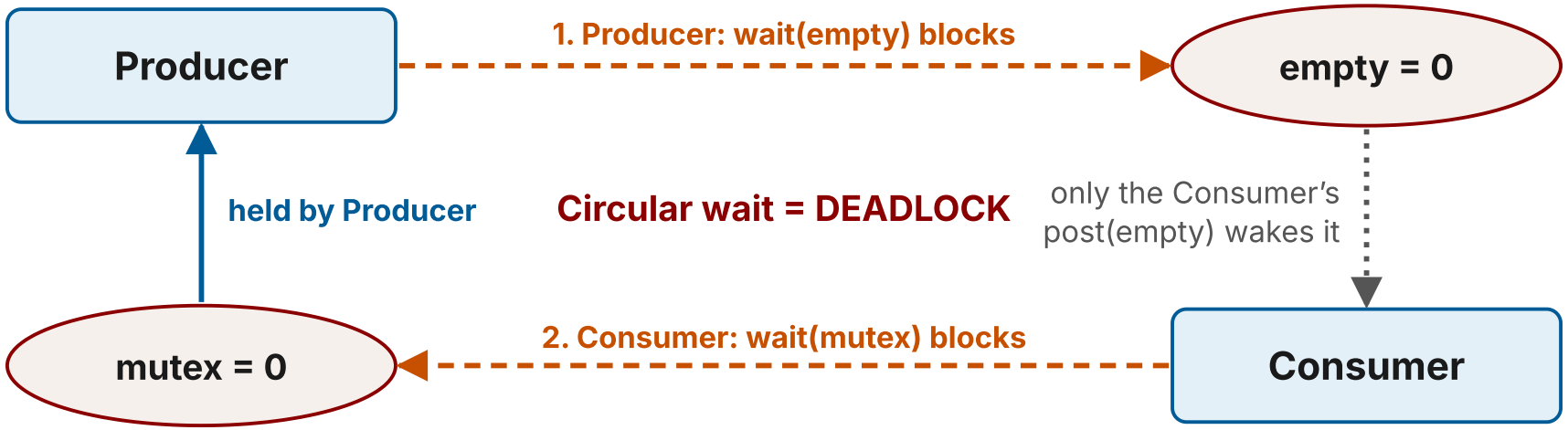
- `empty` and `full` **count** and make threads sleep; `mutex` **excludes**.
- `mutex` is still needed: two producers could otherwise both write `buffer[fill]`.

Bounded Buffer: Producer and Consumer (Correct Order)

```c
void *producer(void *arg) {
    for (int i = 0; i < 100; i++) {
        sem_wait(&empty);    // 1. reserve a free slot (may sleep)
        sem_wait(&mutex);    // 2. enter the critical section
        put(i);
        sem_post(&mutex);    // 3. leave it
        sem_post(&full);     // 4. one more item
    }
    return NULL;
}
void *consumer(void *arg) {
    for (int i = 0; i < 100; i++) {
        sem_wait(&full);     // reserve an item (may sleep)
        sem_wait(&mutex);
        int v = get();
        sem_post(&mutex);
        sem_post(&empty);    // one more free slot
        /* ... process v ... */
    }
    return NULL;
}
```

Swap the Two Waits? Preview of 3.5: Deadlock

Bridge: *is reserve, then lock arbitrary? Try the producer as* `wait(mutex); wait(empty); put(); post(mutex); post(full)` *with the buffer* ***full*** *(*`empty = 0`*): it sleeps on* `empty` ***still holding*** `mutex`*. The consumer passes* `wait(full)`*, then sleeps on* `mutex`*.*



Rule: *never sleep on a resource semaphore while holding a mutex that the other side needs. Lecture 3.5 names this a circular wait.*

✅ Checkpoint 4 — Producer-Consumer with Semaphores

Test your understanding:

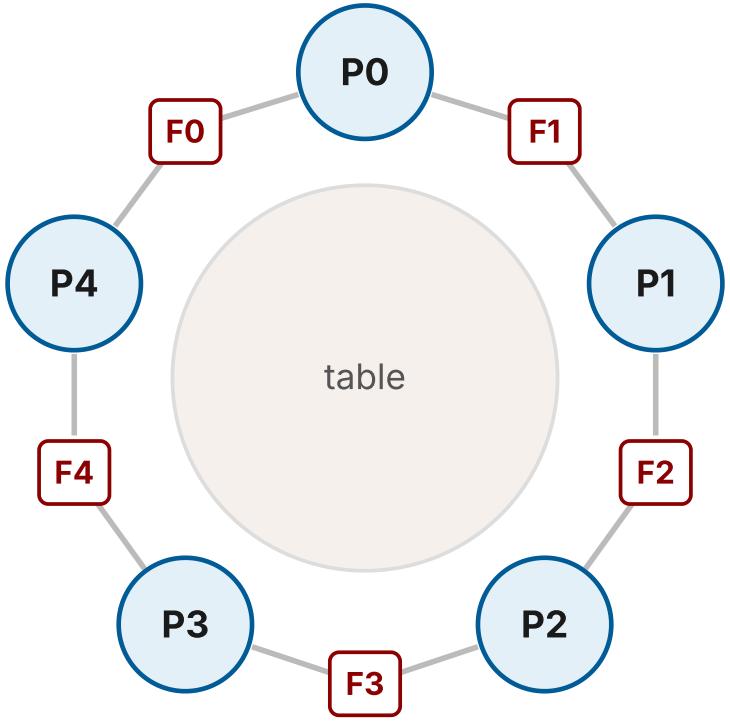
1. What are the three semaphores? Initial value of each, and why?
2. Why must `sem_wait(&empty)` come before `sem_wait(&mutex)` in the producer? What if reversed?
3. Trace the correct bounded buffer (MAX=2), one producer, one consumer, two produce-consume cycles. Give all semaphore values.
4. Two producers call `sem_wait(&empty)` with `empty = 1`. What happens?
5. Buffer full, producer in `sem_wait(&empty)` in the *correct* code: does it hold `mutex`? Why does this matter?

§ 7 — Dining Philosophers

Goal: see a classic problem where naive semaphore use deadlocks, and one fix.

📎 *Extended material → Appendix A.3 (reader-writer lock with semaphores, second fix, Tanenbaum)*

Dining Philosophers: Definition



> *Dijkstra, 1965: concurrency and deadlock in one story.*

- $N = 5$ philosophers, N forks.
- Each one **thinks**, then **eats**.
- Eating needs **both** neighbouring forks: P_i uses F_i (left) and $F_{(i+1)\%N}$ (right).
- **Goal:** no deadlock, no starvation.

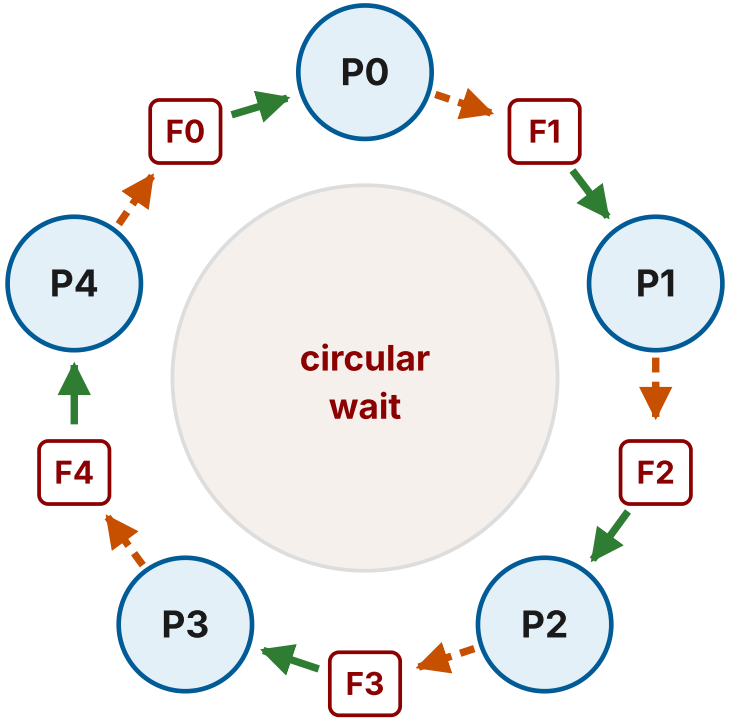
Dining Philosophers: First Attempt

```
#define N 5
sem_t forks[N];   // one semaphore per fork, each initialized to 1

void *philosopher(void *arg) {
    int i = *(int *)arg;
    while (1) {
        think();
        sem_wait(&forks[i]);            // pick up left fork
        sem_wait(&forks[(i+1) % N]);    // pick up right fork
        eat();
        sem_post(&forks[i]);            // put down left fork
        sem_post(&forks[(i+1) % N]);    // put down right fork
    }
}
```

Each fork is a **binary semaphore**; mutual exclusion on forks is correct. But is the whole program?

Preview of Lecture 3.5: The Naive Solution Deadlocks



- All five get hungry together; each grabs its **left** fork F_i (solid).
- Each now waits for its **right** fork $F_{(i+1)\%N}$ (dashed), held by the neighbour.
- Nobody ever puts a fork down: **circular wait = deadlock**.

> *The schedule is unlikely, but possible; a program that can deadlock is broken.*

Dining Philosophers: Breaking the Symmetry

```
void *philosopher(void *arg) {
    int i = *(int *)arg;
    while (1) {
        think();
        if (i == N - 1) {                   // last one: RIGHT fork first
            sem_wait(&forks[(i+1) % N]);    // = fork 0
            sem_wait(&forks[i]);
        } else {                            // others: LEFT fork first
            sem_wait(&forks[i]);
            sem_wait(&forks[(i+1) % N]);
        }
        eat();
        sem_post(&forks[i]);  sem_post(&forks[(i+1) % N]);
    }
}
```

- Everyone takes the **lower-numbered fork first** (a global order), so no cycle can form: **deadlock-free** ✅.
- Fairness is **not** guaranteed. A second fix (limit the table to $N - 1$): Appendix A.3. Ordering is the topic of Lecture 3.5.

✅ Checkpoint 5 — Dining Philosophers

Test your understanding:

1. Naive philosophers, $N = 3$: give a sequence of steps that deadlocks.
2. Each fork is protected correctly by a binary semaphore. Why does the program still deadlock?
3. The schedule "everyone grabs the left fork" is unlikely. Why is the program still broken?
4. Asymmetric solution, $N = 3$: which fork does each philosopher take first? Why is there no cycle?
5. Does the asymmetric solution guarantee that nobody starves?

§ 8 — Semaphore Implementation: The Zemaphore

Goal: show that a semaphore needs nothing beyond a mutex and a CV.

📎 *Extended material → Appendix A.4*

Zemaphore: A Semaphore from Mutex + CV

Idea: `value` *is the predicate, a mutex guards it, a CV sleeps on it. (*`Zem_init`*: set* `value`*, init lock and cond.)*

```c
typedef struct { int value; pthread_mutex_t lock; pthread_cond_t cond; } Zem_t;
void Zem_wait(Zem_t *s) {
    pthread_mutex_lock(&s->lock);
    while (s->value <= 0)                          // no ticket available
        pthread_cond_wait(&s->cond, &s->lock);     // sleep, lock released
    s->value--;                                    // take one ticket
    pthread_mutex_unlock(&s->lock);
}
void Zem_post(Zem_t *s) {
    pthread_mutex_lock(&s->lock);
    s->value++;                          // always: the post is remembered
    pthread_cond_signal(&s->cond);       // wake one sleeper, if any
    pthread_mutex_unlock(&s->lock);
}
```

- `while`, not `if`: Mesa semantics. Decrement **after** the check: `value` never goes negative.
- **Equal expressive power** to mutex + CV. Step-by-step trace and discussion: Appendix A.4.

✅ Checkpoint 6 — The Zemaphore

Test your understanding:

1. Write `Zem_wait()` from memory. Why `while` and not `if`?
2. Trace a Zemaphore with initial value 0: A calls `Zem_wait()`, then B calls `Zem_post()`. Show all value changes.
3. Zemaphore initialized to 3, five threads call `Zem_wait()`. Trace the value and who blocks.
4. Why does `Zem_post()` increment even when nobody is sleeping?
5. Can every semaphore solution be rewritten with locks and CVs, and vice versa?

Summary

> ***A semaphore is an integer $S \geq 0$ with a wait queue and two atomic operations:*** `sem_wait` *(decrement, sleeping while $S = 0$) and* `sem_post` *(increment, wake one).*

Init	Role	Pattern
1	mutual exclusion	`wait; critical section; post`
0	ordering	waiter: `wait` ; signaler: `post` (remembered)
N	resource pool	`wait; use one unit; post`

- **Bounded buffer:** `empty` (N), `full` (0), `mutex` (1); always `wait(resource)` **before** `wait(mutex)` .
- **Dining philosophers:** the naive solution deadlocks; a global fork order fixes it.
- **Zemaphore** = mutex + CV + counter: same expressive power as locks + CVs.

Lecture 3.4 Complete

Recommended Reading

- **OSTEP Ch. 31:** *Semaphores*
- Available free at: https://ostep.org

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it
- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!

§ Appendix

Appendix Map

Section	Extends	What you will find	Slides
A.1	§ 2	Named POSIX semaphores for inter-process synchronization (`sem_open`)	1
A.2	§ 2	Dijkstra's negative-value model, with the binary-semaphore trace redone	1
A.3	§ 7	Reader-writer lock; second Dining Philosophers fix; reader-writer trace; Tanenbaum's solution	4
A.4	§ 8	Zemaphore step-by-step trace, comparison with Dijkstra, binary trace, expressiveness	4
A.5	§ 6, § 7	Exercise 1 (bounded buffer, consumer side), Exercise 2 (Dining Philosophers, $N = 3$)	2

Supplementary material for deeper study. Not required during the main lecture.

📎 A.1 — Inter-Process Named Semaphores

A.1.1 · Named vs. Unnamed POSIX Semaphores

Unnamed Semaphores (`sem_init`)

```c
sem_t s;
sem_init(&s, 0, 1);   // thread-shared, initial value 1
// ... use s ...
sem_destroy(&s);
```

Shared between threads of the same process. Live in memory (global, heap, or stack).

Named Semaphores (`sem_open`)

```c
sem_t *s = sem_open("/my_semaphore", O_CREAT, 0644, 1);   // needs <fcntl.h>
// ... use s ...
sem_close(s);
sem_unlink("/my_semaphore");
```

Identified by a name in the filesystem. Can be shared between **separate processes**.

📎 A.2 — Dijkstra's Original Negative-Value Semantics

A.2.1 · The Negative Value Interpretation (Dijkstra's Model)

Dijkstra decrements **first**; negative values count the sleepers:

Value	Interpretation
$S > 0$	S resources available; no thread sleeping
$S = 0$	no resource available; no thread sleeping
$S < 0$	no resource available; $-S$ threads sleeping

- `sem_wait`: $S \leftarrow S - 1$; if $S < 0$, sleep. `sem_post`: $S \leftarrow S + 1$; if $S \leq 0$, wake one.
- **Binary trace (init 1):** A `wait` $1 \to 0$ (enters); B `wait` $0 \to -1$ (sleeps); A `post` $-1 \to 0$ (B woken, **already holds** the ticket); B `post` $0 \to 1$.
- **POSIX vs. Dijkstra:** same external behavior; POSIX keeps $S \geq 0$ (the main-body convention), so on Linux `sem_getvalue()` never reports a negative number (POSIX also allows reporting a negative count of waiters).

📎 A.3 — Reader-Writer and Dining Philosophers: Extended

A.3.1 · Reader-Writer Lock with Semaphores

Readers share, a writer excludes all. Both semaphores init 1: `writelock` *(writer, or the reader group),* `mutex` *(protects* `readers` *).*

```
sem_t mutex, writelock;   int readers = 0;
void read_lock(void) {
    sem_wait(&mutex);  readers++;
    if (readers == 1) sem_wait(&writelock);   // first reader blocks writers
    sem_post(&mutex);
}
void read_unlock(void) {
    sem_wait(&mutex);  readers--;
    if (readers == 0) sem_post(&writelock);   // last reader admits writers
    sem_post(&mutex);
}
void write_lock(void)   { sem_wait(&writelock); }
void write_unlock(void) { sem_post(&writelock); }
```

- The poster of `writelock` may differ from the waiter: legal for a semaphore, not a mutex.
- **Writer starvation:** continuous readers keep `readers > 0`.

A.3.2 · Reader-Writer Detailed State Trace

```
Initial: mutex=1, writelock=1, readers=0.

R1 read_lock():  wait(mutex) [0], readers=1 -> wait(writelock) [0], post(mutex) [1].
                 R1 READING. ✅
R2 read_lock():  wait(mutex) [0], readers=2 -> no wait on writelock, post(mutex) [1].
                 R2 READING together with R1. ✅
W1 write_lock(): wait(writelock): value 0 -> W1 SLEEPS (readers active). ❌
R1 read_unlock(): wait(mutex), readers=1 -> no post(writelock), post(mutex).
R2 read_unlock(): wait(mutex), readers=0 -> post(writelock) [1], post(mutex).
                 W1 wakes, retakes writelock [0]. W1 WRITING. ✅
```

A.3.3 · Second Fix: Limit the Table (`room` Semaphore)

```
sem_t room;                        // sem_init(&room, 0, N-1): at most N-1 may try to eat
sem_wait(&room);                   // enter the dining room
sem_wait(&forks[i]);               // left fork first, for everyone
sem_wait(&forks[(i+1) % N]);
eat();
sem_post(&forks[i]);  sem_post(&forks[(i+1) % N]);
sem_post(&room);
```

- With $N-1$ seated and N forks, one fork is always free: if the missing philosopher is P_j, then F_j is free and P_{j-1} can take it as its right fork. Deadlock-free.
- Fairness is not guaranteed.

A.3.4 · Tanenbaum's State-Array Solution to Dining Philosophers

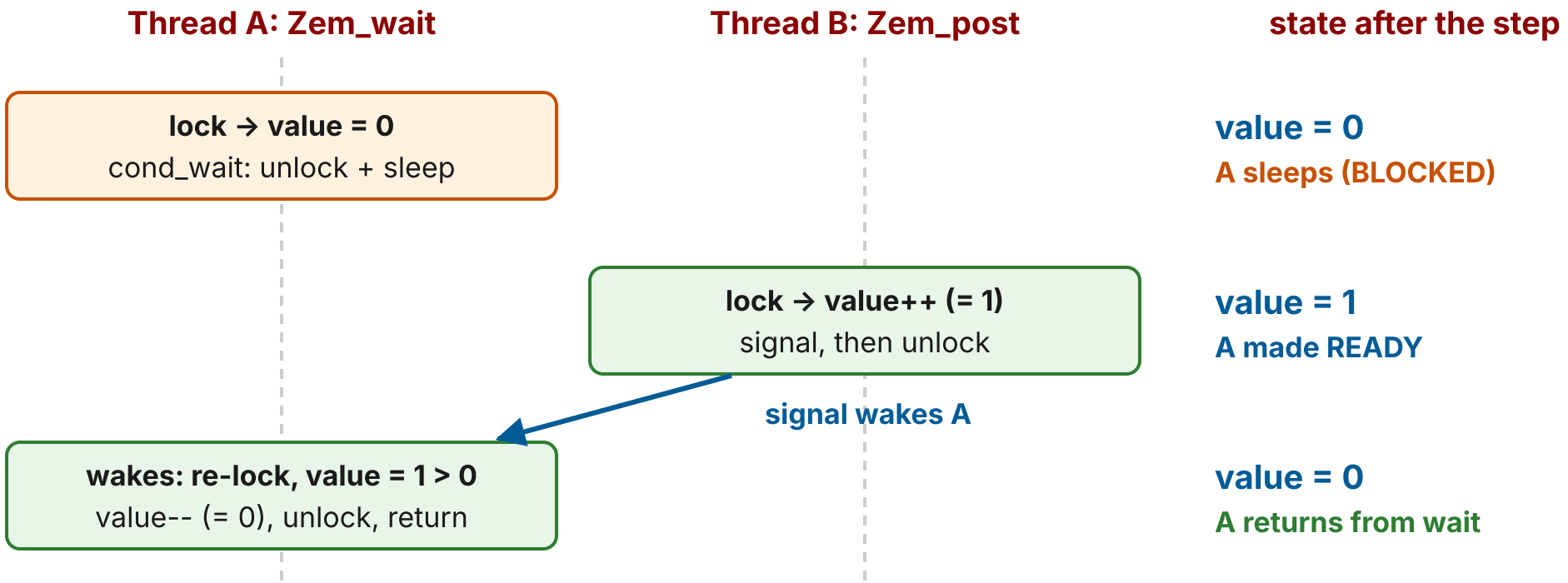
```
typedef enum { THINKING, HUNGRY, EATING } state_t;
state_t state[N];
sem_t   mutex;   // init=1: protects state array
sem_t   s[N];    // one per philosopher, init=0

void test(int i) {     // called with mutex held
    if (state[i] == HUNGRY && state[(i-1+N)%N] != EATING
                           && state[(i+1)%N]   != EATING) {
        state[i] = EATING;
        sem_post(&s[i]);   // philosopher i can eat
    }
}
void take_forks(int i) {
    sem_wait(&mutex);  state[i] = HUNGRY;  test(i);  sem_post(&mutex);
    sem_wait(&s[i]);       // sleep if cannot eat
}
void put_forks(int i) {
    sem_wait(&mutex);  state[i] = THINKING;
    test((i-1+N)%N);  test((i+1)%N);   // maybe a neighbour can eat now
    sem_post(&mutex);
}
```

Deadlock-free ✅; starvation is still possible in principle (neighbours can alternate and starve you).

📎 A.4 — Zemaphore Implementation Analysis

A.4.1 · Zemaphore: Wait and Post, Step by Step



Initial value 0 (an ordering semaphore): A waits first, B posts later.

A.4.2 · Zemaphore vs. Dijkstra's Original Model

Feature	Zemaphore (POSIX-style)	Dijkstra's original
Value	always ≥ 0	can be negative
Wait	`while (value <= 0) cond_wait()` then `value--`	`value--; if (value < 0) block()`
Sleepers	hidden in the CV's queue	$-\text{value}$ = number of sleepers
Wakeup	`cond_signal()`; woken thread re-checks	wakes a thread from the wait queue

A.4.3 · Zemaphore Binary Trace Verification

```
Zemaphore s: value=1 (binary semaphore).

T1 Zem_wait(&s): lock -> value=1 > 0 -> value-- = 0 -> unlock -> proceeds. ✅
T2 Zem_wait(&s): lock -> value=0 -> cond_wait (releases lock, sleeps). ❌
T1 Zem_post(&s): lock -> value++ = 1 -> signal(cond) [wakes T2] -> unlock.
T2 wakes from cond_wait, re-acquires lock:
                  while (value <= 0)? value=1 -> false -> value-- = 0 -> unlock. ✅

T1 and T2 were never in the critical section simultaneously. Correct. ✅
```

A.4.4 · Semaphores vs. Locks+CVs: Expressiveness

- **Equal power:** the Zemaphore builds semaphores from mutex + CV; the converse construction exists but is subtle.
- **Prefer semaphores when:**
    - the problem counts resources;
    - you need remembered signals (no lost wakeups);
    - you implement classic patterns (producer-consumer, reader-writer).
- **Prefer mutex + CV when:**
    - waiting on complex predicates (`x > 5 && y == 0`);
    - you need explicit Mesa semantics and lock ownership.

📎 A.5 — Additional Exam Practice

A.5.1 · Exercise 1: The Consumer Side of the Wrong Order

```
void *bad_consumer(void *arg) {
    sem_wait(&mutex);       // lock first (WRONG)
    sem_wait(&full);        // buffer empty? sleeps holding mutex
    int v = get();  sem_post(&mutex);  sem_post(&empty);
}
```

Task: the buffer is **empty** (`full = 0`) and the producer is correct. Find the deadlock.

- **Consumer:** `wait(mutex)` ⇒ `mutex = 0`; `wait(full)` with `full = 0` ⇒ **sleeps holding `mutex`**.
- **Producer:** `wait(empty)` ⇒ proceeds; `wait(mutex)` with `mutex = 0` ⇒ **sleeps**.
- Only the producer's `post(&full)` could wake the consumer, but it is stuck on `mutex`. **Deadlock.**
- **Fix:** `wait(full)` *before* `wait(mutex)`. The order of the two `post` calls does not matter.

A.5.2 · Exercise 2: Dining Philosophers Deadlock & Asymmetric Solution

- **Setup:** $N = 3$. Forks 0, 1, 2. P0 uses forks (0,1), P1 uses (1,2), P2 uses (2,0).
- **Naive solution** (all three hungry at once):
    - P0 takes fork 0, P1 takes fork 1, P2 takes fork 2.
    - P0 wants fork 1 $\rightarrow$ sleeps (P1 holds it); P1 wants fork 2 $\rightarrow$ sleeps; P2 wants fork 0 $\rightarrow$ sleeps.
    - **Circular wait $\rightarrow$ DEADLOCK.**
- **Asymmetric fix** (P2 takes its right fork first):
    - P0: fork 0 then 1. P1: fork 1 then 2. P2: fork 0 **first**, then fork 2.
    - Fork 0 is contested by P0 and P2; the loser sleeps *holding nothing*, so the winner can finish.
    - Everyone takes the lower-numbered fork first: no cycle, **progress guaranteed**.