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 3.3: Concurrency III

Condition Variables and Producer-Consumer

Opening Thought

> *"… there are many cases where a thread wishes to check whether a condition is true before continuing its execution. For example, a parent thread might wish to check whether a child thread has completed before continuing (this is often called a join()); how should such a wait be implemented?"*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 30*

Locks give us mutual exclusion — the ability to protect shared state from concurrent access. But many concurrent programs need a richer form of coordination: a thread must **wait** until something happens in another thread before it can proceed. This is the problem of **ordering**, and condition variables are the canonical solution.

Lecture Roadmap

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

1. **Recap & Motivation: Beyond Mutual Exclusion** — locks do not express "wait for"
2. **The Condition Variable Abstraction** — `wait()`, `signal()`, atomic release, POSIX API
3. **The Parent-Child Synchronization Pattern** — the `done` state variable, two traces
4. **The Producer-Consumer Problem** — broken attempts: `if`, one CV, `while`
5. **Fixing Producer-Consumer: Two Condition Variables** — `cv_empty` / `cv_full`, bounded buffer
6. **Mesa vs. Hoare Semantics and `broadcast()`** — signal is a hint, why `while` is mandatory, covering condition
7. **Advanced Pattern: The Thread Pool** — producer-consumer in disguise
8. **Worked Example & Summary** — spot the bug, key takeaways

> *Extended material (Mesa-vs-Hoare table, covering-condition definition, barrier, reader-writer lock, Java monitors, exam problems 2–3) is in the **Appendix** (A.1–A.5): relevant section dividers point to their appendix counterpart.*

Prerequisite Check

To follow this lecture comfortably, ensure you recall:

- **Locks and mutual exclusion** (Lecture 3.2): `lock()`, `unlock()`, critical section.
- **Spinning vs. sleeping** (Lecture 3.2): busy-waiting burns CPU; a sleeping lock blocks the thread in the OS instead.
- **Thread states** (Lecture 3.1): RUNNING, READY, BLOCKED — and transitions between them.
- **`pthread` API basics** (Lecture 3.1): `pthread_create()`, `pthread_join()`, `pthread_mutex_t`.
- **Race conditions** (Lecture 3.1): why unprotected shared accesses are dangerous.

New concepts introduced today: condition variable, `wait()`, `signal()`, `broadcast()`, state variable, producer-consumer, bounded buffer, Mesa vs. Hoare semantics, spurious wakeup, covering condition, thread pool.

§ 1 — Recap & Motivation: Beyond Mutual Exclusion

Goal: see why a lock alone cannot make a thread *wait for an event*.

Motivating Example: Parent Waiting for Child

> ***Recap:*** *locks give mutual exclusion — taking turns on shared data. They say nothing about* ***waiting for something to happen****.* ***Problem:*** *the parent must not print "child done" before the child has finished.*

```c
void *child(void *arg) {
    printf("child: running\n");
    // ... do some work ...
    printf("child: done\n");
    return NULL;
}

int main(void) {
    pthread_t tid;
    printf("parent: creating child\n");
    pthread_create(&tid, NULL, child, NULL);
    // How does the parent WAIT here until the child is done?
    printf("parent: child done\n");
    return 0;
}
```

`pthread_join()` solves this specific case, but we need a general mechanism for arbitrary conditions — not just thread completion.

Naive Solution: Spin-Waiting on a Shared Flag

> ***First idea:*** *the child sets a flag when done; the parent spins until it sees the flag.*

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child: running\n");
    done = 1;           // signal completion
    return NULL;
}

int main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, child, NULL);
    while (done == 0) { /* spin */ }   // busy-wait for child
    printf("parent: child done\n");
    return 0;
}
```

It usually works, but it is **wasteful**: the parent burns a whole CPU core (or time slice) doing nothing useful, and under a priority scheduler on one core the spinner can even starve the child. Also, `volatile` only stops the compiler from caching `done`: it does not make accesses atomic or ordered, so this is still a data race.

The Solution: Condition Variable

> ***Idea:*** *instead of spinning,* ***sleep*** *(be BLOCKED in the OS) and let the other thread* ***wake you*** *when the condition becomes true.*

> ***A condition variable (CV) is a synchronization primitive that allows a thread to sleep until a particular condition is true, and allows another thread to signal when that condition has become true.***

Key properties (all made precise in § 2):

- A CV is always used together with a **mutex** that protects the shared state encoding the condition.
- A sleeping thread **releases the mutex and sleeps in one atomic step** — this is what prevents missed signals.
- A woken thread **re-acquires the mutex** before returning from the sleep call.
- A CV has no memory of its own: the condition is expressed by shared variables.

✅ Checkpoint 1 — Recap & Motivation

Test your understanding:

1. What does a lock (mutex) guarantee? What problem does it **not** solve?
2. Give two real-world concurrent programming scenarios where a thread must **wait for a condition** rather than compete for exclusive access.
3. Why is spin-waiting on a `volatile` flag a poor solution for inter-thread notification? Name three specific problems.
4. What two behaviors must a condition variable provide that spin-waiting does not?

§ 2 — The Condition Variable Abstraction

Goal: know exactly what `wait()` and `signal()` do, and why `wait()` must release the lock atomically.

Condition Variable: Definition and Intuition

> ***So far:*** *we want to sleep until "something is true".* ***Question:*** *what is "the something", and where do we sleep?*

- **Condition:** a predicate over shared **state variables** (e.g. `done == 1`, `count > 0`). It lives in ordinary variables, protected by a **mutex**.
- **Condition variable:** just a **queue of sleeping threads** — no counter, no memory of past signals.
- The CV provides the sleeping/waking; the mutex + state variables provide the truth.

Analogy — a waiting room:

- A thread enters the waiting room (`wait()`) when the condition is false.
- It sleeps until someone calls `signal()` (wake at least one) or `broadcast()` (wake all).
- Once awake, it leaves the room and **re-checks** the condition.

The Two Fundamental Operations: `wait()` and `signal()`

`wait(cv, m)` — caller must hold `m`

1. **Atomically** release `m` AND put the caller to sleep on `cv`'s queue (BLOCKED).
2. The caller sleeps until another thread signals `cv`.
3. When woken (BLOCKED → READY) and scheduled, it **re-acquires** `m`; only then does `wait()` return.

`signal(cv)`

1. Wakes **one** thread sleeping on `cv` (if any): BLOCKED → READY.
2. The signaler **keeps running and keeps** `m`; the woken thread cannot proceed before the signaler unlocks.
3. If nobody sleeps on `cv`: the signal is **lost** (nothing is remembered).

> ***Waking up is not running:*** *by then the state may have changed, so the woken thread must* ***re-check*** *the condition (§ 4 shows why; § 6 names it Mesa semantics).*

Why `wait()` Must Release the Lock *Atomically*

> ***Problem:*** *the waiter holds* `m` *while it checks the condition. It must release* `m` *so that someone else can change the state — but when?*

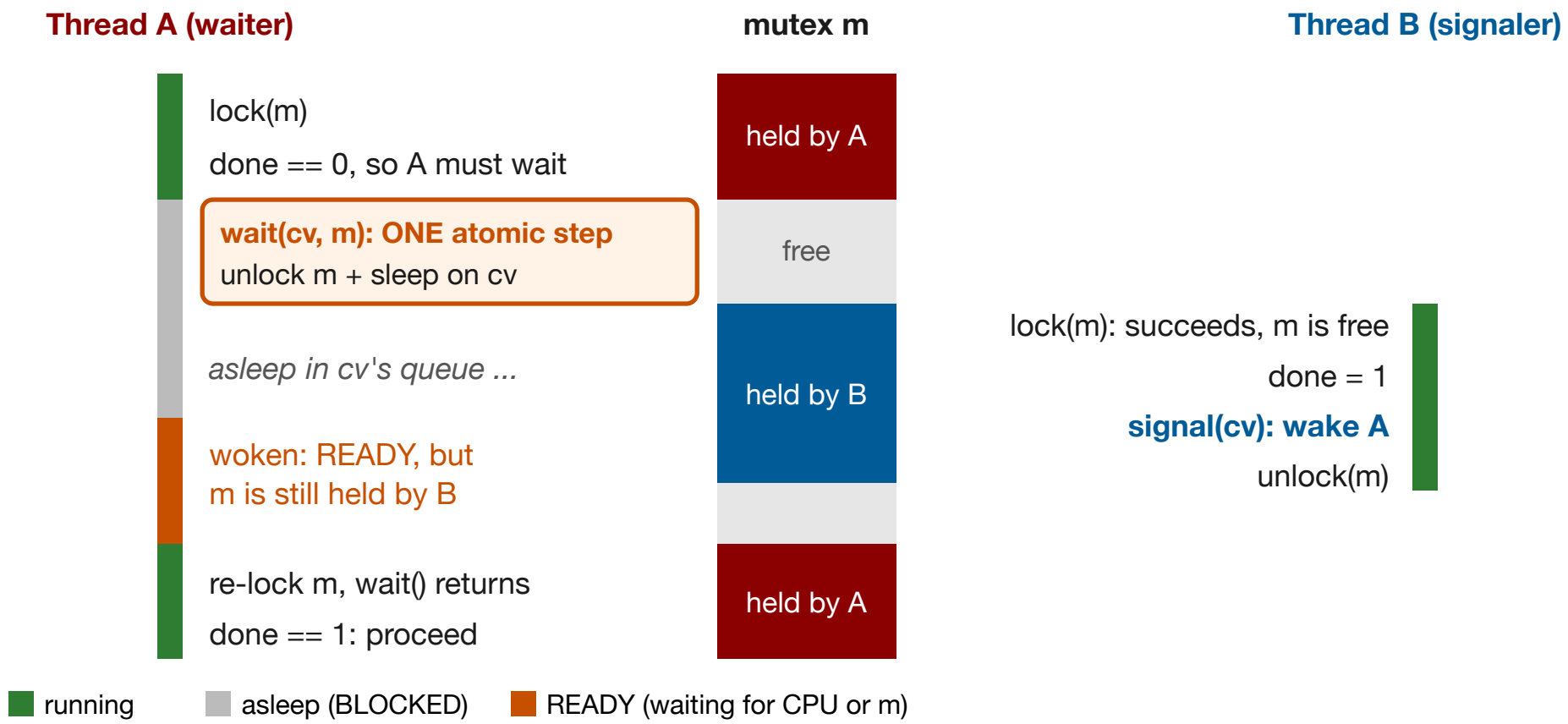
Suppose `wait()` were two separate steps: (1) `unlock(m)`, then (2) sleep.

```
A:  lock(m);   while (done == 0) {   // done == 0 -> A must wait
A:      unlock(m);                   // naive wait: unlocking without sleeping atomically (step 1)
        //  ---- A is preempted here; B runs ----
B:  lock(m); done = 1; signal(cv); unlock(m);  // nobody is asleep: signal LOST
A:      sleep on cv;                 // unlocking without sleeping atomically (step 2): A sleeps FOREVER
```

This is a **lost wakeup**: the condition became true *between* A's check and A's sleep.

> `pthread_cond_wait()` *does "release* `m` *+ enqueue + sleep" as* ***one indivisible step*** *— there is no gap for B to slip into.*

Sequence: `wait()` Releasing the Lock Atomically



*B could take `m` only because A's `wait()` released it **as part of going to sleep**. A re-checks the condition after re-locking.*

The Usage Pattern: Check–Sleep–Act / Change–Signal

```
// WAITER
pthread_mutex_lock(&m);
while (!condition) {                // (1) check
    pthread_cond_wait(&cv, &m);     // (2) sleep
}
perform_action();                   // (3) act
pthread_mutex_unlock(&m);
```

```
// SIGNALER
pthread_mutex_lock(&m);
make_condition_true();              // (4) change
pthread_cond_signal(&cv);           // (5) signal
pthread_mutex_unlock(&m);
```

Rules (memorize this template — every correct CV usage is a variation of it):

- Hold `m` when calling `wait()` (required). Always change the shared state **under** `m`; signaling while holding `m` is the safe default (POSIX allows signaling after unlocking, but only if the state change was made under `m`).
- Check the condition under `m`, **in a** `while` **loop** (why: § 4 and § 6). Always use the **same** mutex for the same condition.
- Change the state first, then signal.

POSIX API: `pthread_cond_t`

```c
#include <pthread.h>

pthread_cond_t cv = PTHREAD_COND_INITIALIZER;   // static initialization
pthread_cond_init(&cv, NULL);                   // or dynamic (NULL = default attributes)

// Atomically release mutex + sleep; re-acquire mutex before returning.
// The caller must hold `mutex` when calling.
int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);

// Wake one waiting thread (no effect if none is waiting)
int pthread_cond_signal(pthread_cond_t *cv);

// Wake ALL waiting threads
int pthread_cond_broadcast(pthread_cond_t *cv);

pthread_cond_destroy(&cv);                      // cleanup

// All return 0 on success; an error code on failure.
```

✅ Checkpoint 2 — The Condition Variable Abstraction

Test your understanding:

1. What does `pthread_cond_wait(&cv, &mutex)` do? List all three actions it performs, in order.
2. What happens if `pthread_cond_signal(&cv)` is called when no thread is waiting? What can go wrong if the state is changed *outside* the mutex?
3. Explain why "release mutex AND sleep" must be **atomic**. Which bug arises if they are two separate operations?
4. Why does `wait()` take the mutex as an argument? What does it do with it on return?
5. State the usage pattern. What must be true at the moment of `wait()` and at the moment `wait()` returns?

§ 3 — The Parent-Child Synchronization Pattern

Goal: apply the pattern to the simplest ordering problem and see why the state variable matters.

The Correct Solution: CV with Mutex

Condition: `done == 1`. **Ingredients:** state variable `done`, mutex `m`, CV `cv`. It must work whether the child finishes before or after the parent reaches the wait point, and the parent must not spin.

```
int done = 0;                                   // shared state variable
pthread_mutex_t m  = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;

void thr_exit(void) {                   // called by the child
    pthread_mutex_lock(&m);
    done = 1;                           // (4) change the state
    pthread_cond_signal(&cv);           // (5) wake the parent (if it sleeps)
    pthread_mutex_unlock(&m);
}

void thr_join(void) {                   // called by the parent
    pthread_mutex_lock(&m);
    while (done == 0)                   // (1) check
        pthread_cond_wait(&cv, &m);     // (2) sleep: release m, re-acquire on wakeup
    pthread_mutex_unlock(&m);           // (3) nothing else to do: just proceed
}
```

Full Implementation: Parent and Child

```
// done, m, cv, thr_exit(), thr_join(): as on the previous slide
void *child(void *arg) {
    printf("child: running\n");
    thr_exit();         // signal completion
    return NULL;
}

int main(void) {
    pthread_t tid;
    printf("parent: creating child\n");
    pthread_create(&tid, NULL, child, NULL);
    thr_join();         // wait for child to finish
    printf("parent: child done\n");
    return 0;
}
// Always prints:  parent: creating child
//                 child: running
//                 parent: child done
```

Trace: Both Orders Work

Order A — child finishes before the parent reaches `thr_join()`:

```
Child:   lock(m); done = 1; signal(cv) [nobody asleep: signal LOST]; unlock(m)
Parent:  lock(m); while (done == 0)? done is 1 -> false: never calls wait(); unlock(m)   OK
```

Order B — parent reaches `thr_join()` first:

```
Parent:  lock(m); while (done == 0)? true -> wait(cv, m): release m, sleep
Child:   lock(m); done = 1; signal(cv) [parent -> READY]; unlock(m)
Parent:  re-lock m; wait() returns; while (done == 0)? false; unlock(m)                OK
```

In both orders the parent proceeds exactly when `done == 1`.

Why the State Variable and the Mutex Are Essential

The CV **remembers nothing**: a signal sent while nobody sleeps is gone. The state variable `done` holds the **persistent truth** of the condition.

- **Without `done`** (parent always calls `wait()`): in Order A the child's signal is lost, the parent sleeps **forever**.
- **With `done`**: in Order A the parent sees `done == 1` and never sleeps — the state compensates for the lost signal.
- **Without the mutex:** the parent checks `done == 0`; the child then sets `done = 1` and signals; only then does the parent sleep — the same **lost wakeup** as in § 2 (a gap between check and sleep).

> ***Rule:*** *a CV must ALWAYS be paired with a state variable that encodes the condition, and both are protected by the mutex.*

✅ Checkpoint 3 — The Parent-Child Synchronization Pattern

Test your understanding:

1. Trace `thr_join()` and `thr_exit()` when the child finishes before the parent reaches `thr_join()`. Does the parent ever call `pthread_cond_wait()`? Why not?
2. Why is the `done` variable necessary alongside the condition variable? What goes wrong if we remove it and rely solely on the CV signal?
3. What race condition arises if we remove the mutex from `thr_join()` / `thr_exit()`?
4. Write the usage pattern from memory. Label each part: check, sleep, act (waiter); change, signal (signaler).
5. In `thr_join()` we use `while (done == 0)`. Would `if` also work in this two-thread example? Do you expect it to keep working in general?

§ 4 — The Producer-Consumer Problem

Goal: build a correct producer-consumer step by step — each broken attempt teaches one rule.

The Producer-Consumer Problem and the Single-Slot Buffer

> ***Producers*** *generate items and put them in a fixed-size shared buffer;* ***consumers*** *take them out (the* ***bounded-buffer problem****). Examples: a Unix pipe, a web server's accept thread feeding workers.*

Invariants: a producer waits if the buffer is **full**; a consumer waits if it is **empty**; one thread at a time touches the buffer. We start with **one slot**; `count` is the **state variable** (`loops` = items per thread):

```
int buffer;           // the single shared slot
int count = 0;        // 0 = empty, 1 = full

void put(int value) { assert(count == 0); count = 1; buffer = value; }
int  get(void)      { assert(count == 1); count = 0; return buffer; }
```

`put()` / `get()` assume the right state — enforcing it is the job of the conditions we add next.

Attempt 1: One Condition Variable, with `if`

A natural first implementation — one CV for both kinds of waiter:

```c
pthread_mutex_t m  = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    pthread_mutex_lock(&m);
    if (count == 1)         // full?
      pthread_cond_wait(&cv, &m);
    put(i);
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&m);
  }
  return NULL;
}
```

```c
void *consumer(void *arg) {
  for (int i = 0; i < loops; i++) {
    pthread_mutex_lock(&m);
    if (count == 0)         // empty?
      pthread_cond_wait(&cv, &m);
    int val = get();
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&m);
  }
  return NULL;
}
```

With **one producer and one consumer** this works (ignoring spurious wakeups, § 6): they strictly alternate put/get.

Next: what if there are two consumers?

Break #1: Two Consumers — Woken Does Not Mean Running

Setup: consumers C1, C2, producer P; same code as Attempt 1 (`if`). C1 is already asleep on `cv` (`count == 0`).

#	Thread	Action	`count`
1	C1	`lock`; `if (count == 0)` true → `wait(cv)`: releases `m`, sleeps	0
2	P	`lock`; `put()`; `signal(cv)` → C1 becomes READY (not running yet)	1
3	P	`unlock(m)` — `m` is free, C1 has not run	1
4	C2	`lock`; `if (count == 0)` false → `get()`; `unlock` — **steals the item**	0
5	C1	scheduled: re-locks `m`, `wait()` returns — **no re-check** after `if`	0
6	C1	`get()` → `assert(count == 1)` **fails: CRASH**	0

> *Between the signal (step 2) and C1's re-lock (step 5), another thread changed the state. A woken thread must **re-check** the condition: `if` → `while`.*

Attempt 2 and Break #2: `while`, but Still One CV — Deadlock

Attempt 2: `if` → `while` in both threads (`while (count == 1)` / `while (count == 0)`). This fixes Break #1: at step 5, C1 re-tests `count == 0` and waits again ✅. But with one `cv`, C1 and C2 start first (`count == 0`) and P produces at least two items:

#	Thread	Action	`count`	asleep on `cv`
1	C1	`count == 0` → `wait(cv)`	0	C1
2	C2	`count == 0` → `wait(cv)`	0	C1, C2
3	P	`put()`; `signal(cv)` → wakes C1 (READY)	1	C2
4	P	next item: `count == 1` → `wait(cv)`	1	C2, P
5	C1	re-locks; `get()`; `signal(cv)` → wakes **C2** (not P!)	0	P
6	C1	next item: `count == 0` → `wait(cv)`	0	P, C1
7	C2	re-locks; `while (count == 0)` → `wait(cv)` again	0	P, C1, C2

Everyone sleeps, nobody can signal: the consumer's signal woke a *consumer*, but was meant for the *producer*.

✅ Checkpoint 4 — The Producer-Consumer Problem

Test your understanding:

1. State the producer-consumer problem. What are the invariants that must be maintained?
2. With one producer and one consumer, does Attempt 1 (one CV, `if`) work? Trace one full put/get cycle.
3. In Break #1, at which step does `if` cause the failure? What happened between the signal and the wakeup?
4. Why does `while` fix Break #1? What does it do that `if` does not?
5. Why does Break #2 still deadlock with `while`? What goes wrong with a single CV when producers and consumers both sleep on it?

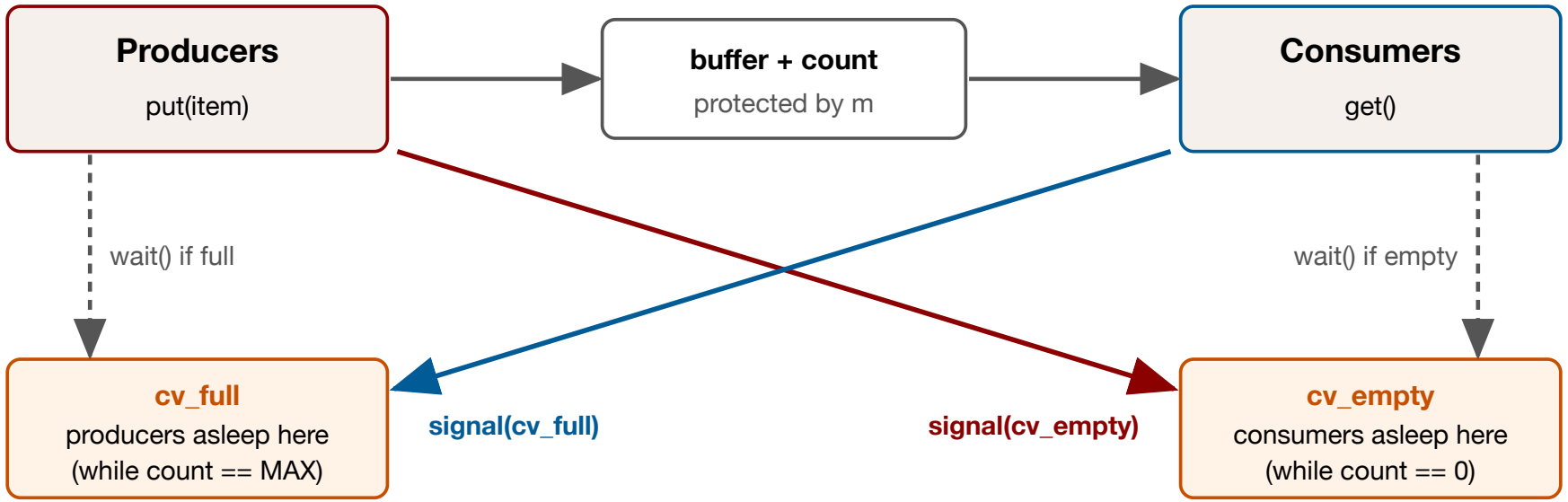
§ 5 — Fixing Producer-Consumer: Two Condition Variables

Goal: one CV per condition, then scale the buffer from 1 slot to N.

The Root Cause: One CV, Two Types of Waiter

> ***From Break #2:*** *a signal cannot choose who it wakes.* ***Idea:*** *give each condition its own CV, so every sleeper on a CV waits for the same thing.*

- `cv_empty`: **consumers** sleep here while the buffer is **empty**; `cv_full`: **producers** sleep here while it is **full**.
- After `put`, a producer signals `cv_empty`; after `get`, a consumer signals `cv_full`.



The Fix: Two Separate Condition Variables

```c
pthread_mutex_t m        = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t  cv_full  = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    pthread_mutex_lock(&m);
    while (count == 1)          // full: wait
      pthread_cond_wait(&cv_full, &m);
    put(i);
    pthread_cond_signal(&cv_empty); // wake a consumer
    pthread_mutex_unlock(&m);
  }
  return NULL;
}
```

```c
void *consumer(void *arg) {
  for (int i = 0; i < loops; i++) {
    pthread_mutex_lock(&m);
    while (count == 0)          // empty: wait
      pthread_cond_wait(&cv_empty, &m);
    int val = get();
    pthread_cond_signal(&cv_full);  // wake a producer
    pthread_mutex_unlock(&m);
  }
  return NULL;
}
```

Only three things changed from Attempt 2: two CVs, `while`, and each side signals the **other** side's CV.

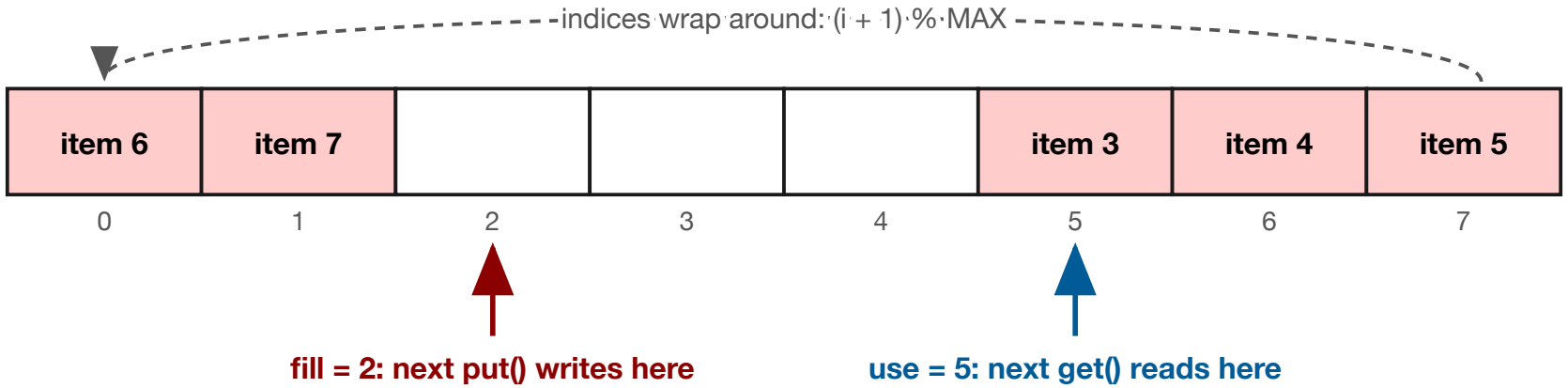
Replay of Break #2 with Two CVs

Same scenario (C1, C2 start first; P produces two items):

#	Thread	Action	`count`	asleep on `cv_empty` / `cv_full`
1–2	C1, C2	`count == 0` → `wait(cv_empty)`	0	C1, C2 / —
3	P	`put()` ; `signal(cv_empty)` → wakes C1	1	C2 / —
4	P	next item: `count == 1` → `wait(cv_full)`	1	C2 / P
5	C1	`get()` ; `signal(cv_full)` → wakes **P** (only sleeper there)	0	C2 / —
6	C1	next item: `count == 0` → `wait(cv_empty)`	0	C2, C1 / —
7	P	re-locks; `count == 1`? no → `put()` ; `signal(cv_empty)` → progress ✅	1	one consumer woken

A consumer's signal can now reach only a producer, and vice versa — no deadlock. The `while` remains necessary (Break #1 does not depend on how many CVs we use).

Generalizing: The Bounded Buffer (Size N > 1)



MAX = 8, `count` = 5 items (3 to 7), wrapping past the end. `fill == use` both when empty and when full — `count` tells which. The producer now waits while `count == MAX`; the rest is as in the single-slot code (full listing on the next slide).

```
#define MAX 8
int buffer[MAX], fill = 0, use = 0, count = 0;

void put(int val) { buffer[fill] = val; fill = (fill + 1) % MAX; count++; }
int  get(void)    { int val = buffer[use]; use = (use + 1) % MAX; count--; return val; }
```

Full Bounded Buffer Implementation

Same structure as the single-slot version: `put()` / `get()` as above, mutex and two CVs as before; the full test becomes `count == MAX`.

```c
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    pthread_mutex_lock(&m);
    while (count == MAX)      // full
      pthread_cond_wait(&cv_full, &m);
    put(i);
    pthread_cond_signal(&cv_empty);
    pthread_mutex_unlock(&m);
  }
  return NULL;
}
```

```c
void *consumer(void *arg) {
  for (int i = 0; i < loops; i++) {
    pthread_mutex_lock(&m);
    while (count == 0)        // empty
      pthread_cond_wait(&cv_empty, &m);
    int val = get();
    pthread_cond_signal(&cv_full);
    pthread_mutex_unlock(&m);
    printf("consumed: %d\n", val);
  }
  return NULL;
}
```

Correct for **any number** of producers and consumers. `printf` is outside the lock: no need to hold `m` while consuming.

✅ Checkpoint 5 — Fixing Producer-Consumer: Two Condition Variables

Test your understanding:

1. For each of `cv_empty` and `cv_full`: which condition does it stand for, who sleeps on it, and who signals it?
2. Why can Break #2 (deadlock) not happen with two CVs? Which thread does the consumer's signal reach?
3. In the bounded buffer, which condition does the producer test in its `while`? The consumer? Why can't we use `fill == use` to detect "full"?
4. Trace the bounded buffer (`MAX = 2`) with 2 producers and 2 consumers, 3 items each. Check that `count` stays in $[0, \mathrm{MAX}]$ and that no thread sleeps forever.
5. With two CVs, is `if` instead of `while` safe now? Use Break #1 to justify.

§ 6 — Mesa vs. Hoare Semantics and `broadcast()`

Goal: know what `signal()` promises (hence why `while` is mandatory) and when `signal()` is not enough.

📎 *Extended material → Appendix A.1 (Mesa vs. Hoare comparison), A.2 (covering condition)*

Two Flavors of Condition Variable Semantics

> ***From § 4:*** *the woken consumer found* `count == 0`*. Is that a bug in our code, or a property of* `signal()`*? It depends on what* `signal()` *promises — two designs exist.*

When thread S calls `signal()` and thread W sleeps on the CV:

Hoare Semantics (1974)

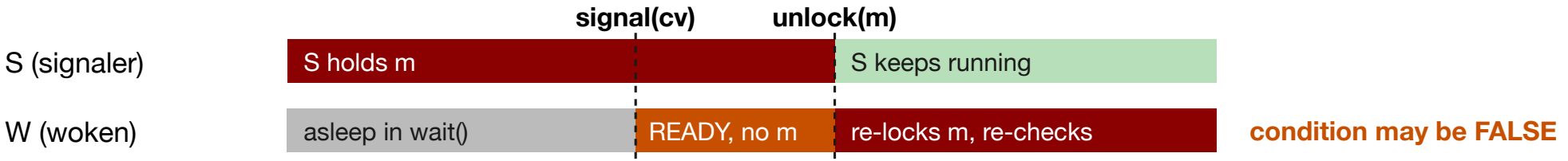
- S **immediately hands the mutex to W**; W runs at once and S is suspended.
- The condition is **guaranteed** true when W returns from `wait()`.
- S resumes when W releases the mutex (or waits again).

Mesa Semantics (1980, Xerox PARC)

- S **keeps running and keeps the mutex**; W moves to READY.
- W runs at some **later scheduling decision**, after re-acquiring the mutex — others may get it first.
- The condition **may or may not** be true when W runs.

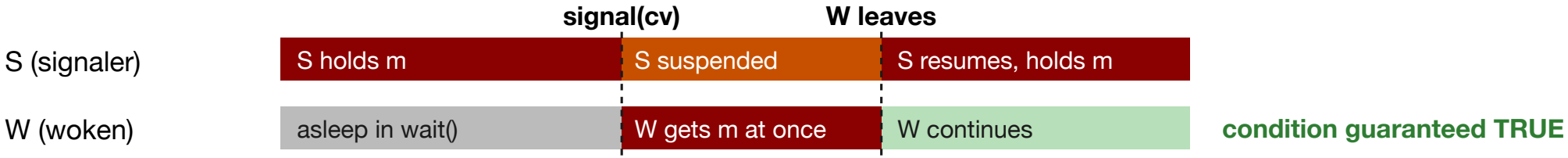
Timeline: Who Holds the Mutex?

Mesa: signal-and-continue



Between signal and W's re-lock, another thread may take m and change the state.

Hoare: signal-and-wait (hand-off)



No one can intervene, but every signal costs an immediate context switch (and a switch back).

holds the mutex | running without the mutex | READY / suspended | asleep (BLOCKED)

Mesa Semantics: Signal Is a Hint, So `while` Is Mandatory

> ***Real systems use Mesa semantics*** *(POSIX, Java, C#, Go): hand-off to the waiter would cost a context switch per signal. So* `signal()` *is only a* ***hint****: "something changed — wake up and CHECK." (Full comparison table: A.1.)*

Three ways a woken thread finds the condition **false**:

- **a. Sneak-in:** another thread takes `m` between the signal and W's re-lock (Break #1).
- **b. Spurious wakeup:** `wait()` returns although nobody signaled — POSIX explicitly allows it.
- **c. Shared CV, different conditions:** the signal wakes a waiter for a condition that did not change (Break #2; the allocator, next).

Hence `while (!condition) wait(cv, m);` — if false go back to sleep, if true proceed. Under POSIX, `if` is **always a bug**.

When `signal()` Is Not Enough: The Covering Condition

> ***From § 5:*** *one CV per condition worked because every sleeper on a CV waits for the same thing. What if sleepers on one CV need different things?*

Motivating example: a memory allocator (one CV, shared counter `bytes_free`)

- `bytes_free == 0`. Thread A is asleep waiting for 100 bytes; thread B is asleep waiting for 50 bytes.
- Another thread frees 60 bytes: `bytes_free = 60`. B can now proceed (60 ≥ 50); A cannot.
- The freeing thread calls `signal()` — it wakes **one** sleeper, and it cannot choose:
    - wakes A: `60 < 100` → A (correctly, thanks to `while`) sleeps again. **B stays asleep although its condition is true.**
    - wakes B: fine — but only by luck.

> *`while` keeps the program safe, but the wasted wakeup costs progress. We would need one CV per request size to target the right thread.*

`broadcast()` as the Solution

When we cannot tell which waiter(s) can proceed, wake them **all**:

```
void *allocate(int size) {
    pthread_mutex_lock(&m);
    while (bytes_free < size)           // wait until THIS request fits
        pthread_cond_wait(&cv, &m);
    bytes_free -= size;
    void *ptr = do_alloc(size);         // carve out a block (details omitted)
    pthread_mutex_unlock(&m);
    return ptr;
}

void free_mem(void *ptr, int size) {
    pthread_mutex_lock(&m);
    bytes_free += size;                 // memory returned
    pthread_cond_broadcast(&cv);        // wake EVERY waiter
    pthread_mutex_unlock(&m);
}
```

Each woken thread re-tests its **own** `bytes_free < size`: B proceeds, A goes back to sleep. The price is the **thundering herd**: all waiters wake and contend for `m`, and most just sleep again. This is a **covering condition** (it covers every waiter that might proceed): use `broadcast()` when waiters wait for *different* conditions, `signal()` when they all wait for the *same* one. `broadcast()` + `while` is never incorrect, only slower.

✅ Checkpoint 6 — Mesa vs. Hoare Semantics and `broadcast()`

Test your understanding:

1. Describe Hoare and Mesa semantics: what does the signaler do right after `signal()`, and what is the woken thread guaranteed?
2. Why is `if` safe under Hoare but a bug under Mesa? Point to the step in Break #1.
3. Give three situations in which a woken thread finds the condition false. What is a spurious wakeup, and why is it harmless with `while`?
4. A (100 bytes) and B (50) are asleep; 60 bytes are freed with `signal()`. Describe the bad outcome and how `broadcast()` fixes it.
5. When is `signal()` enough and when do you need `broadcast()`? What is the thundering herd?

§ 7 — Advanced Pattern: The Thread Pool

Goal: recognize check–sleep–act / change–signal in a real program structure.

📎 *Extended material → Appendix A.3 (barrier), A.4 (reader-writer locks, Java monitors)*

The Thread Pool Pattern

> ***A producer-consumer in disguise:*** `submit()` *is the producer, the workers are the consumers, the queue is the buffer.*

```c
typedef struct { void (*func)(void*); void *arg; } Task;
Task queue[MAX_TASKS];
int  head = 0, tail = 0, count = 0;
pthread_mutex_t m  = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;

void *worker(void *arg) {
  while (1) {
    pthread_mutex_lock(&m);
    while (count == 0)
      pthread_cond_wait(&cv, &m);
    Task t = queue[head];
    head = (head + 1) % MAX_TASKS;
    count--;
    pthread_mutex_unlock(&m);
    t.func(t.arg);     // outside the lock
  }
}
```

```c
void submit(void (*f)(void*), void *a) {
  pthread_mutex_lock(&m);
  // assumes count < MAX_TASKS
  // (else: a 2nd CV, as in § 5)
  queue[tail] = (Task){f, a};
  tail = (tail + 1) % MAX_TASKS;
  count++;
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&m);
}
```

`signal()` is enough: all workers wait for the same condition (`count > 0`) and one task feeds one worker. Running the task **outside the lock** lets tasks run in parallel.

✅ Checkpoint 7 — The Thread Pool Pattern

Test your understanding:

1. In the thread pool, why does the worker execute the task **outside** the mutex? What happens if it held the lock?
2. Which producer-consumer roles do `submit()` and `worker()` play? Why is `signal()` enough in `submit()`?
3. Map `worker()` and `submit()` onto check–sleep–act and change–signal.
4. What extra mechanism would `submit()` need if the queue could fill up?

§ 8 — Worked Example & Summary

Goal: apply the pattern to a bug-hunting exercise and consolidate the lecture.

📎 *Extended material → Appendix A.5 (exam problems 2–3)*

Problem 1: Spot the Bug — `if` Instead of `while`

```
// Consumer with a bug (two CVs; the producer is correct):
void *consumer(void *arg) {
    pthread_mutex_lock(&m);
    if (count == 0)                    // BUG: should be while
        pthread_cond_wait(&cv_empty, &m);
    int val = get();
    pthread_cond_signal(&cv_full);
    pthread_mutex_unlock(&m);
    return NULL;
}
```

Question: with 2 consumers (C1, C2) and 1 producer (P), trace a scenario where this code crashes. *(Hint: two CVs rule out cross-wakeups — what else can happen between the signal and the wakeup?)*

Problem 1: Solution

```
Initial: count = 0. C1 is asleep on cv_empty (C2 has not started yet).

P:   lock; put() -> count = 1; signal(cv_empty) -> C1 READY (needs m); unlock.
C2:  starts now: lock; if (count == 0)? count is 1 -> false -> get(); count = 0; unlock.
       (C2 sneaked in before C1 re-locked m and took the item.)
C1:  re-locks m; wait() returns; the if was already passed -> get() with count == 0
       -> assert(count == 1) FAILS -> CRASH. ✗

Fix: replace if with while.
C1:  re-locks m; while (count == 0)? true -> wait(cv_empty) again -> sleeps. ✅
```

Summary

- Locks give **mutual exclusion**; condition variables give **ordering**: sleep until a condition is true.
- A CV is only a **queue of sleepers**; the condition lives in **state variables** protected by the mutex (signals are not remembered).
- `wait()` = **atomically** release `m` + sleep; re-acquire `m` before returning. Atomicity prevents lost wakeups.
- **Mesa semantics** (all real systems; Hoare hands off at once, too costly): a signal is a hint, so waiters use `while` (sneak-in, spurious wakeups, shared CV).
- **Producer-consumer:** mutex + `count` + circular buffer, and **one CV per condition** (`cv_empty`, `cv_full`) with `signal()`.
- Waiters with **different** conditions on one CV → `broadcast()` (covering condition); cost: thundering herd.
- A **thread pool** is producer-consumer in disguise: same pattern, one CV, `signal()`.

Summary: The Template and the Rules

```
// WAITER
pthread_mutex_lock(&m);
while (!condition)              // (1) check
    pthread_cond_wait(&cv, &m); // (2) sleep
perform_action();               // (3) act
pthread_mutex_unlock(&m);
```

```
// SIGNALER
pthread_mutex_lock(&m);
make_condition_true();          // (4) change
pthread_cond_signal(&cv);       // (5) signal
pthread_mutex_unlock(&m);
```

1. **Two CVs, not one** — no cross-type wakeups.
2. `while`**, not** `if` — handles sneak-in and spurious wakeups.
3. **Change the state first, then signal.**
4. **Hold the mutex** whenever you touch the state or call `wait()`.

Lecture 3.3 Complete

Recommended Reading

- **OSTEP Ch. 30:** *Condition Variables*
- 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	§ 6	Mesa vs. Hoare comparison table; `if` vs. `while` code	2
A.2	§ 6	The covering condition: definition and when to use `broadcast()`; check questions	2
A.3	§ 7	The barrier pattern (generation counter); check questions	2
A.4	§ 7	Reader-Writer lock with CVs, writer starvation, Java monitors, check questions	6
A.5	§ 8	Exam problems: bounded buffer from scratch (Problem 2), RW lock trace (Problem 3)	2

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

📎 A.1 — Mesa vs. Hoare: Comparison and Code

A.1.1 · Why Mesa Semantics Is Used in Practice

	Hoare	Mesa (POSIX, Java, C#, Go, …)
Cost of `signal()`	immediate context switch (and back)	just moves W to READY
Implementation	direct lock hand-off, extra queue of suspended signalers	plain block / unblock
Guarantee	condition true on return → `if` would suffice	**none** — `signal()` is a hint
Code	simple for the waiter, hard for the OS	waiter must re-check: `while`

Virtually all real systems and languages use **Mesa** semantics: cheaper, simpler, and robust to extra or unexpected wakeups.

A.1.2 · `if` vs. `while`: The Only Code Difference

```
// Hoare semantics (not used in practice):
pthread_mutex_lock(&m);
if (!condition)                 // if is enough: condition guaranteed true on return
    pthread_cond_wait(&cv, &m);
do_work();
pthread_mutex_unlock(&m);

// Mesa semantics (POSIX pthreads — always write this):
pthread_mutex_lock(&m);
while (!condition)              // while is required: re-check after every wakeup
    pthread_cond_wait(&cv, &m);
do_work();                      // condition is true NOW: we just checked it, holding m
pthread_mutex_unlock(&m);
```

Under POSIX (Mesa, spurious wakeups allowed), `if` is **always a bug**. `while` is correct under both semantics.

📎 A.2 — The Covering Condition

A.2.1 · The Covering Condition

> ***A covering condition is a condition that "covers" every case in which some waiter might be able to proceed, when we cannot tell which one.***

Use `broadcast()` when…	Use `signal()` when…
waiters on one CV have **different** conditions (allocate 100 vs 50)	all waiters wait for the **same** condition
one state change can satisfy **several** waiters (300 bytes freed)	one state change enables **exactly one** waiter (one item → one consumer)
you are unsure which waiter to wake	you can argue which waiters may proceed

`broadcast()` + `while` is **never incorrect** — only possibly slower. Remark: it would even have rescued the one-CV producer-consumer of § 4 (at the price of waking everyone on every operation); two CVs with `signal()` is the precise solution.

A.2.2 · Check Yourself: `broadcast()` and the Covering Condition

Test your understanding:

1. In the allocator example, A (100 bytes) and B (50 bytes) are asleep and 60 bytes are freed with `signal()`. Describe the bad outcome.
2. What does each thread do after `broadcast()` here? Why is `broadcast()` with `while` always correct under Mesa semantics?
3. What is the thundering herd? Why is it acceptable in a covering condition?
4. Give one example from this lecture where `signal()` is enough and one where `broadcast()` is needed.
5. Define "covering condition" in your own words.

📎 A.3 — The Barrier Pattern

A.3.1 · The Barrier Pattern

A **barrier**: every thread waits until all `total` threads have arrived.

```
typedef struct {
    int total, arrived, generation;           // generation = current round
    pthread_mutex_t m;  pthread_cond_t cv;
} barrier_t;

void barrier_wait(barrier_t *b) {
    pthread_mutex_lock(&b->m);
    int my_gen = b->generation;
    if (++b->arrived == b->total) {         // last thread to arrive
        b->arrived = 0;                     // reset for the next round
        b->generation++;                    // this round is over
        pthread_cond_broadcast(&b->cv);     // wake ALL waiters
    } else {
        while (my_gen == b->generation)     // wait until THIS round ends
            pthread_cond_wait(&b->cv, &b->m);
    }
    pthread_mutex_unlock(&b->m);
}
```

Why `generation`? *Waiting on `arrived < total` breaks on reuse: a fast thread can re-enter after the reset and make a slow waiter sleep again. (POSIX: `pthread_barrier_t`.)*

A.3.2 · Check Yourself: The Barrier

1. Explain the barrier pattern. What does `pthread_cond_broadcast()` do in `barrier_wait()`? Why is `broadcast()` necessary here instead of `signal()`?
2. Why does the barrier use a `generation` counter instead of waiting on `arrived`?
3. Which thread plays "change–signal" and which play "check–sleep" in `barrier_wait()`?

📎 A.4 — Reader-Writer Locks & Language Monitors

A.4.1 · The Reader-Writer Lock Problem

Not all concurrent accesses require exclusive access. Reads can proceed concurrently:

Requirements for a reader-writer lock:

- Multiple readers can hold the lock simultaneously (read concurrently).
- A writer needs EXCLUSIVE access (no readers, no other writers).

Motivation:

- A database with frequent reads and rare writes.
- Allowing concurrent reads dramatically improves throughput.
- Writers must wait for all current readers to finish; readers must wait for any active writer to finish.

Two logical states:

- "Reader-held": $N \geq 1$ readers currently reading. New readers can join; writers must wait.
- "Writer-held": exactly 1 writer writing. Both readers and writers must wait.

A.4.2 · Implementing a Reader-Writer Lock with CVs

```
typedef struct {
    int       readers;    // active readers
    int       writers;    // 0 or 1 (active writer)
    pthread_mutex_t m;
    pthread_cond_t  cv_read;
    pthread_cond_t  cv_write;
} rwlock_t;

void read_lock(rwlock_t *rw) {
    pthread_mutex_lock(&rw->m);
    while (rw->writers > 0)           // wait while writer active
        pthread_cond_wait(&rw->cv_read, &rw->m);
    rw->readers++;
    pthread_mutex_unlock(&rw->m);
}

void read_unlock(rwlock_t *rw) {
    pthread_mutex_lock(&rw->m);
    rw->readers--;
    if (rw->readers == 0)
        pthread_cond_signal(&rw->cv_write);  // last reader: wake writer
    pthread_mutex_unlock(&rw->m);
}
```

A.4.3 · Reader-Writer Lock: Write Lock and Unlock

```
void write_lock(rwlock_t *rw) {
    pthread_mutex_lock(&rw->m);
    while (rw->writers > 0 || rw->readers > 0)
        pthread_cond_wait(&rw->cv_write, &rw->m);
    rw->writers = 1;
    pthread_mutex_unlock(&rw->m);
}

void write_unlock(rwlock_t *rw) {
    pthread_mutex_lock(&rw->m);
    rw->writers = 0;
    pthread_cond_broadcast(&rw->cv_read);   // wake ALL waiting readers
    pthread_cond_signal(&rw->cv_write);     // and one waiting writer
    pthread_mutex_unlock(&rw->m);
}

// POSIX provides pthread_rwlock_t for production use:
// pthread_rwlock_rdlock(), pthread_rwlock_wrlock(), pthread_rwlock_unlock()
```

Readers and the writer both re-test their `while` condition after waking; whoever finds it false sleeps again.

`broadcast` for readers: *all* of them may enter together.

A.4.4 · Writer Starvation and Fair RW Locks

A subtle fairness problem: a stream of readers R1, R2, R3, … arrives every 10 ms, and a writer W is waiting.

```
R1: read_lock()  -> writers == 0 -> readers = 1.
W:  write_lock() -> readers > 0 -> sleeps on cv_write.
R2: read_lock()  -> read_lock() tests ONLY writers (== 0) -> joins: readers = 2.
R1: read_unlock() -> readers = 1.   (R2 still reading)
R3: read_lock()  -> joins: readers = 2.   ... and so on.
```

W **never** gets the write lock as long as readers keep overlapping: this is **writer starvation**.

Mitigations:

1. **Pending-writer flag:** if a writer is waiting, new readers are not admitted.
2. **Priority-based:** writers have higher priority than readers.
3. POSIX leaves the policy implementation-defined.

A.4.5 · Condition Variables in Java (`wait()` / `notify()`)

Java's `synchronized` / `wait()` / `notify()` mechanism mirrors POSIX CVs (also Mesa semantics):

```
// Every Java object has a built-in monitor (mutex + ONE condition queue)
class BoundedBuffer {
    private int[] buf = new int[8];
    private int count = 0, fill = 0, use = 0;

    public synchronized void put(int val) throws InterruptedException {
        while (count == buf.length) wait();   // full: releases the lock and sleeps
        buf[fill] = val; fill = (fill + 1) % buf.length; count++;
        notifyAll();   // wake everyone (producers AND consumers)
    }

    public synchronized int get() throws InterruptedException {
        while (count == 0) wait();            // empty: wait
        int val = buf[use]; use = (use + 1) % buf.length; count--;
        notifyAll();
        return val;
    }
}
```

`wait()` / `notify()` / `notifyAll()` ≡ `cond_wait` / `signal` / `broadcast`; `synchronized` ≡ lock/unlock around the method. With only **one** condition queue, `notifyAll()` is needed (not `notify()`) — as in Break #2.

A.4.6 · Check Yourself: Reader-Writer Locks and Java

1. In `write_unlock()`, why use `broadcast()` to wake readers but `signal()` to wake a writer?
2. Describe writer starvation in a reader-writer lock. What design change prevents it?
3. What is Java's equivalent of `pthread_cond_wait()`, `pthread_cond_signal()`, and `pthread_cond_broadcast()`?
4. Why does the Java `BoundedBuffer` need `notifyAll()` rather than `notify()`?

📎 A.5 — Additional Exam Practice Problems

A.5.1 · Problem 2: Implement Bounded Buffer from Scratch

Task: implement a correct bounded buffer with capacity 5, supporting multiple producers and consumers. Use two condition variables and `while` loops.

```c
#define N 5
int buf[N], fill = 0, use = 0, count = 0;
pthread_mutex_t m        = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv_full  = PTHREAD_COND_INITIALIZER;   // producers wait here
pthread_cond_t  cv_empty = PTHREAD_COND_INITIALIZER;   // consumers wait here

void produce(int val) {
    pthread_mutex_lock(&m);
    while (count == N) pthread_cond_wait(&cv_full, &m);
    buf[fill] = val; fill = (fill + 1) % N; count++;
    pthread_cond_signal(&cv_empty);
    pthread_mutex_unlock(&m);
}

int consume(void) {
    pthread_mutex_lock(&m);
    while (count == 0) pthread_cond_wait(&cv_empty, &m);
    int v = buf[use]; use = (use + 1) % N; count--;
    pthread_cond_signal(&cv_full);
    pthread_mutex_unlock(&m);
    return v;
}
```

A.5.2 · Problem 3: Tracing Reader-Writer Lock State Transitions

Setup: RW lock initialized. R1, R2 are readers. W1 is a writer.

```
Step 1: R1 calls read_lock().
  writers == 0 -> no wait. readers = 1. R1 in CS.

Step 2: R2 calls read_lock().
  writers == 0 -> no wait. readers = 2. R1 and R2 both in CS concurrently. ✅

Step 3: W1 calls write_lock().
  readers == 2 > 0 -> W1 waits on cv_write.

Step 4: R1 finishes. read_unlock(): readers = 1. readers != 0 -> no signal.

Step 5: R2 finishes. read_unlock(): readers = 0. signal(cv_write).
  W1 wakes, re-locks, re-tests: writers == 0 && readers == 0 -> loop exits. W1 enters CS.

Step 6: W1 finishes. write_unlock(): writers = 0.
  broadcast(cv_read): wake all waiting readers (none here).
  signal(cv_write): wake one waiting writer (none here). Done.

Concurrent reads ✅. Writer waits for all readers ✅. Exclusive write ✅.
```