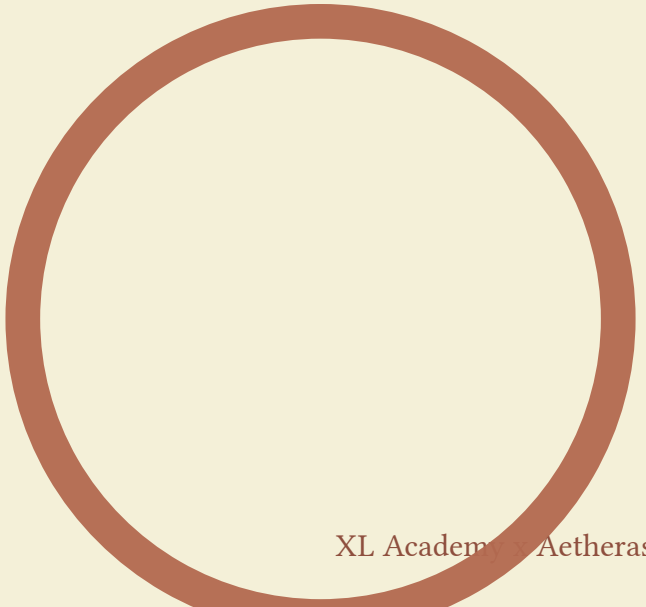
L17

AI Game Development & Computer Science Summer Program

Concurrency and Parallelism



Basic Definitions

Start by separating simultaneous progress from actually running at the same instant.

Concurrency means multiple computations can make progress

They may interact, and the order may matter.

Concurrent examples

- A cursor moves while a program computes.
- A timer keeps updating while you type.
- A browser waits for a network response while the page stays usable.
- A game handles input while animation continues.

Why it matters

- The program has more than one active task.
- Tasks can share state.
- The result may depend on timing.
- Responsiveness is a user-facing feature.

Parallelism is one way to implement concurrency

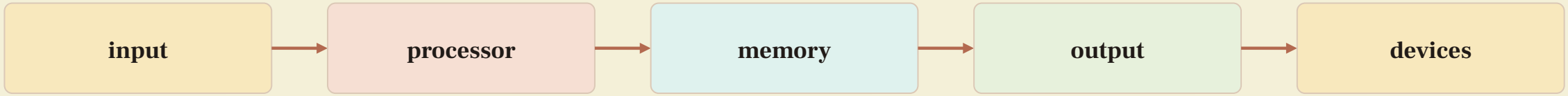
Concurrent work can be interleaved, parallel, or distributed.

Term	Meaning	Course example
concurrent	multiple tasks are in progress	input, timer, and render loop
parallel	tasks run at the same time	workers on different CPU cores
distributed	tasks run on different machines	server plus browser clients

A program can be concurrent without being parallel.

Computers already split responsibility

Concurrency fits into the bigger architecture of a machine.



Processor role

- Control decides what instruction comes next.
- Datapath does arithmetic, movement, and comparisons.
- Modern machines often have multiple cores that can help at once.

Processors are physical objects with limits

More transistors helped for decades, but heat and power changed the path forward.

Layer	What matters
silicon chip	tiny circuits switch signals
transistors	billions to trillions of switches
package	connects chip to board and moves heat away
power	too much heat limits clock-speed growth

Course takeaway

- Hardware limits are why software increasingly relies on multiple helpers instead of one ever-faster core.

Why Parallelism Became Important

More chips did not simply make every old program faster by magic.

Moore's Law was about transistor growth

The historical pattern was roughly more transistors per chip over time.

Old expectation	What changed
more transistors	more possible computation
faster clock speeds	harder because of heat and power
single fast processor	many cores and specialized processors

Modern result

- To use more hardware, software often needs work that can be split.

Different processors help different kinds of work

Parallel hardware is powerful only when the problem shape fits it.

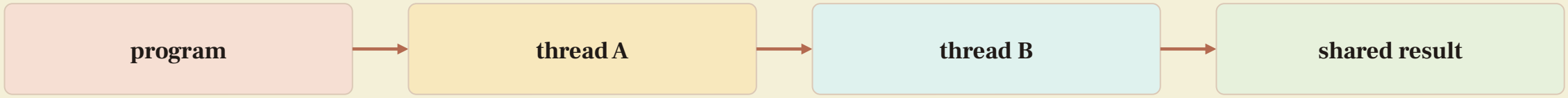
Hardware	Good fit	Limit to notice
CPU cores	separate tasks with branching logic	coordination and shared state
GPU cores	many similar calculations on pixels or arrays	poor fit for tangled dependencies
efficient chips	more work per watt	heat and battery still matter

Course takeaway

- Do not just ask whether a computer has many helpers.
- Ask whether the work can be split without changing the answer.

Threads describe separate streams of execution

A program can fork work into separate paths.



Thread idea

- Each thread has its own instruction stream.
- On one core, time sharing can switch between threads.
- On multiple cores, threads may run in parallel.

Time sharing can feel simultaneous

A single core can switch quickly between tasks.

Time slice	Task running	User-visible effect
1	input handler	keypress appears
2	animation	sprite moves
3	timer	clock changes
4	input handler	next key appears

Key point

- The user sees smooth progress even if only one task runs at each tiny moment.

Browsers add their own concurrency model

TypeScript code often runs on one main thread, but events arrive over time.

Browser piece	Concurrency role
event loop	chooses the next task to run
timers	schedule future work
fetch	waits outside your code and resumes later
Web Worker	runs code away from the main UI thread

Important nuance

- Most ordinary UI handlers do not run at the same instant, but their completion order can still surprise you.

Speedup Has Limits

Adding helpers creates potential, overhead, and new failure modes.

Amdahl's Law explains why speedup is capped

The serial part of a program limits the benefit of extra cores.

Program part	Can more helpers speed it up?
serial setup	no
parallel work	yes, if it splits cleanly
combine results	usually limited

Amdahl formula

- speedup = 1 / (serial + parallel / workers)
- If 20% must stay serial, even perfect parallel work cannot beat 5x overall.

Parallel work has overhead

Splitting and recombining work also costs time

Overhead	What can go wrong
divide work	the split is uneven
send work	communication costs time
shared resources	workers block each other
slowest worker	everyone waits
combine results	final merge may be serial

Some problems split better than others

Parallelism helps most when pieces are independent.

Splits well

- Resize 100 independent images.
- Check many candidate moves independently.
- Run tests that do not share state.

Splits poorly

- Update one shared score repeatedly.
- Wait for each step before the next one exists.
- Use one resource that only one task can hold.

Responsiveness is a speedup goal too

Concurrency is not only about finishing faster; it can keep the interface alive.

Heavy task	Bad UI	Better UI
path search	page freezes	show progress or use a worker
large validation	button appears broken	disable with status text
asset loading	empty screen	show loading state and retry path

AI assist: ask where work can be split, but verify setup cost, shared state, and user-facing feedback yourself.

Parallel Programming Is Hard

Once tasks interact, timing can change correctness.

Race conditions happen when timing changes the result

Two tasks read and write shared state in an unsafe order.

```
let balance = 100;

const withdraw75 = async (): Promise<boolean> => {
  const observed = balance;
  await Promise.resolve();
  if (observed < 75) return false;
  balance = observed - 75;
  return true;
};

await Promise.all([withdraw75(), withdraw75()]);
```

Trace risk

- Both calls read 100 before either resumes after await.
- Both return true, but the final balance is 25: one withdrawal was not accounted for.

Nondeterministic code can run in more than one valid order

The same code and input can produce different outcomes.

Interleaving	Read A	Read B	Write order	Result
safe	100	25	A then B rejects	25
race	100	100	A then B writes	25 or wrong audit
race	100	100	B then A writes	25 or wrong audit

If your proof only works for one order, the concurrent program is not proven correct.

Browser code can still have order bugs

Async completion order can overwrite newer state with older information.

```
let selectedLevel = "forest";

const loadLevel = async (level: string) => {
  const data = await fetchLevel(level);
  selectedLevel = level;
  render(data);
};

loadLevel("forest");
loadLevel("desert");
```

Bug to find

- What if forest finishes after desert?
- Which level should the player see?
- Where should the code check the latest request?

Avoid races by controlling ownership

Make one place responsible for updating shared state.

Hazard	Safer habit
many tasks write state	one reducer or update function
stale async result	check request id before rendering
shared mutable object	copy data before handing it out
unclear order	log event, old state, new state

TypeScript help

- Types can make state shape explicit, but types do not automatically prove timing is safe.

Deadlock means everyone waits forever

Each task holds something another task needs.

Task	Holds	Waiting for
A	pencil	ruler
B	ruler	pencil

Software version

- Two parts of a program each hold one resource.
- Neither can continue until the other releases.
- No task is technically crashing, but no progress happens.

Livelock means movement without progress

Tasks keep reacting to each other but never finish.

Deadlock

- No one moves.
- Everyone waits.
- The system is stuck silently.

Livelock

- Tasks keep changing.
- Each reaction blocks the other.
- The system looks busy but gets nowhere.

Use Concurrency Deliberately

A responsive program needs a timing story, not just more helpers.

Trace concurrent code as possible orders

Do not assume the order you want is the order you get.

Step	Task A	Task B	Shared state
1	read state		score = 4
2		read state	score = 4
3	write score 6		score = 6
4		write score 5	score = 5

The table shows a lost update: Task B overwrites Task A's newer score.

Exercise: classify the concurrency hazard

Name the problem before proposing a fix

Scenario	Hazard
two async loads render in the wrong order	_____
two workers update one shared score	_____
two tasks each hold one needed resource	_____
workers spend more time coordinating than working	_____
a serial setup step dominates total runtime	_____

Exercise: make the level loader safe

Ignore stale results before they can overwrite the current screen.

```
let latestRequest = 0;

const loadLevel = async (level: string) => {
  const requestId = latestRequest + 1;
  latestRequest = requestId;

  const data = await fetchLevel(level);
  // TODO: what should we check here?
  render(data);
};
```

Target behavior

- If a newer request has started, the older result should not render.
- The UI should show the level the player most recently chose.

AI assist: ask for possible interleavings and stale-result tests, then trace them yourself.

Summary: concurrency is power plus responsibility

The hard part is coordinating shared state and timing.

Keep

- Concurrency means multiple computations can make progress.
- Parallelism means work actually runs at the same time.
- More cores help only when work can split cleanly.
- Serial work, overhead, races, nondeterminism, deadlock, and livelock limit correctness and speed.
- Browser TypeScript still needs a timing story for events, async results, and workers.

Proof checkpoint

Keep the artifact that proves today's concept.

Proof to keep

- One concurrency definition.
- One concurrency vs parallelism comparison.
- One speedup limit or overhead example.
- One traced interleaving.
- One race, nondeterminism, deadlock, or livelock fix idea.