## Lecture 33: Concurrency

- Moore's law ("Transistors per chip doubles every $N$ years"), where $N$ is roughly 2 (about $1,000,000\times$ increase since 1971).
- Has also applied to processor speeds (with a different exponent).
- But predicted to flatten: further increases to be obtained through *parallel processing* (witness: multicore/manycode processors).
- With distributed processing, issues involve interfaces, reliability, communication issues.
- With other parallel computing, where the aim is performance, issues involve synchronization, balancing loads among processors, and, yes, "data choreography" and communication costs.

## Example of Parallelism: Sorting

- Sorting a list presents obvious opportunities for parallelization.
- Can illustrate various methods diagrammatically using *comparators* as an elementary unit:
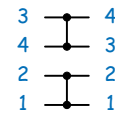


- Each vertical bar represents a *comparator*—a comparison operation or hardware to carry it out—and each horizontal line carries a data item from the list.
- A comparator compares two data items coming from the left, swapping them if the lower one is larger than the upper one.
- Comparators can be grouped into operations that may happen simultaneously; they are always grouped if stacked vertically as in the diagram.

## Sequential sorting

- Here's what a sequential sort (selection sort) might look like:
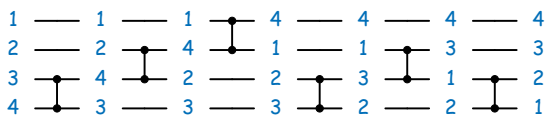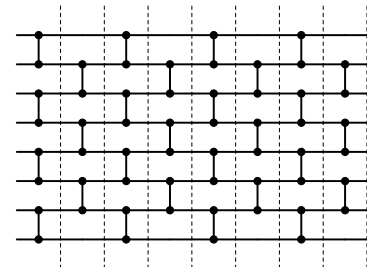


- Each comparator is a separate operation in time.
- In general, there will be $\Theta(N^2)$ steps.
- But since some comparators operate on distinct data, we ought to be able to overlap operations.

## Odd-Even Transposition Sorter



—— Data    ┋ Comparator    ┊ Separates parallel groups

## Odd-Even Sort Example
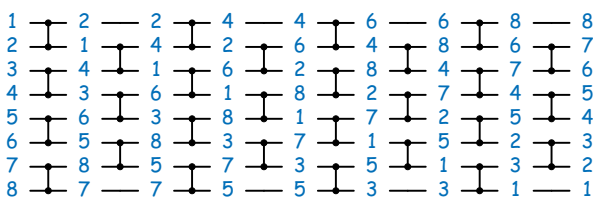
## Example: Bitonic Sorter



—— Data    ┋ Comparator    ┊ Separates parallel groups

| 77 | 77 | 77 | 77 | 77 | 77 | 92 |
|----|----|----|----|----|----|----|
| 16 | 16 | 47 | 47 | 47 | 92 | 77 |
| 8 | 47 | 16 | 16 | 52 | 52 | 52 |
| 47 | 8 | 8 | 8 | 92 | 47 | 47 |
| 1 | 52 | 52 | 92 | 8 | 8 | 16 |
| 52 | 1 | 92 | 52 | 16 | 16 | 8 |
| 6 | 92 | 1 | 6 | 6 | 6 | 6 |
| 92 | 6 | 6 | 1 | 1 | 1 | 1 |
| 24 | 24 | 24 | 99 | 99 | 99 | 99 |
| 7 | 7 | 99 | 24 | 35 | 56 | 56 |
| 99 | 99 | 7 | 15 | 48 | 48 | 48 |
| 15 | 15 | 15 | 7 | 56 | 35 | 35 |
| 13 | 35 | 48 | 56 | 7 | 24 | 24 |
| 35 | 13 | 56 | 48 | 15 | 15 | 15 |
| 56 | 56 | 13 | 35 | 24 | 7 | 13 |
| 48 | 48 | 35 | 13 | 13 | 13 | 7 |

| 92 | 92 | 92 | 92 | 99 |
|----|----|----|----|----|
| 77 | 77 | 77 | 99 | 92 |
| 52 | 52 | 56 | 56 | 77 |
| 47 | 47 | 99 | 77 | 56 |
| 16 | 35 | 35 | 52 | 52 |
| 8 | 48 | 48 | 48 | 48 |
| 6 | 56 | 52 | 35 | 47 |
| 1 | 99 | 47 | 47 | 35 |
| 99 | 1 | 24 | 24 | 24 |
| 56 | 6 | 15 | 16 | 16 |
| 48 | 8 | 13 | 13 | 15 |
| 35 | 16 | 16 | 15 | 13 |
| 24 | 24 | 1 | 8 | 8 |
| 15 | 15 | 6 | 7 | 7 |
| 13 | 13 | 8 | 1 | 6 |
| 7 | 7 | 7 | 6 | 1 |
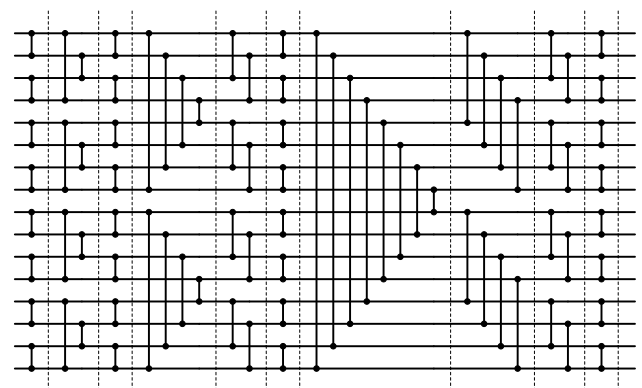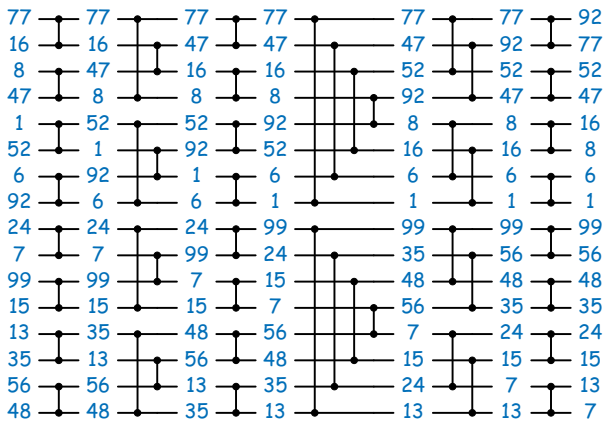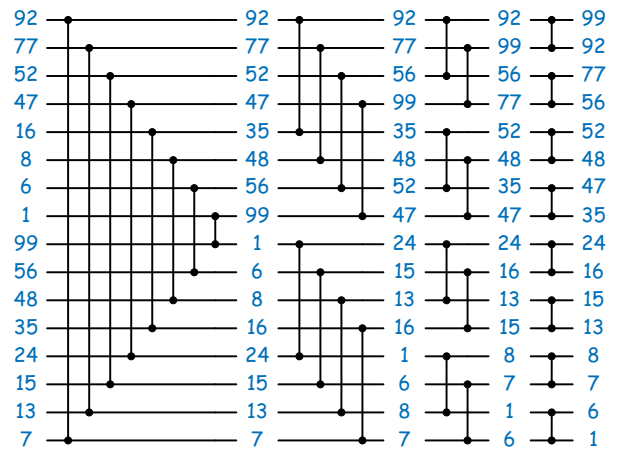
## Mapping and Reducing in Parallel

- The `map` function in Python conceptually provides many opportunities for parallel computation, if the computations of invididual items is *independent*.
- Less obviously, so does `reduce`, if the operation is *associative*. If list `L == L1 + L2`, and `op` is an associative operation, then

  `reduce(op, L) == op(reduce(op, L1), reduce(op, L2))`

  and the two smaller reductions can happen in parallel.

## Map-Reduce

- Google[tm] patented an embodiment of this approach (the validity of which is under dispute). Here's a very simplified version.
- User specifies a mapping operation and a reduction operation.
- In the mapping phase, the map operation is applied to each item of data, yielding a *list of key-value pairs* for each item.
- The reduce operation is then applied on all the values for each distinct key.
- The final result is a list of key-value pairs, with each value being the reduction of the values for that key as produced by the mapping phase.
- Standard simple example:
  - Each input item is a page of text.
  - The map operation takes a page of text ("The cow jumped over the moon…") and produces a list with the words as keys and the value 1 (`("the", 1), ("cow", 1), ("jumped", 1), ...`).
  - The reduce phase now sums the values for each key.
  - Result: for each key (word), get the total count.

## Implementing Parallel Programs

- The sorting diagrams were abstractions.
- Comparators could be processors, or they could be operations divided up among one or more processors.
- Coordinating all of this is the issue.
- One approach is to use *shared memory,* where multiple processors (logical or physical) share one memory.
- This introduces conflicts in the form of *race conditions:* processors racing to access data.

## Memory Conflicts: Abstracting the Essentials

- When considering problems relating to shared-memory conflicts, it is useful to look at the primitive read-to-memory and write-to-memory operations.
- E.g., the program statements on the left cause the actions on the right.

```
x = 5                    WRITE 5 -> x
x = square(x)            READ x -> 5
                         (calculate 5*5 -> 25)
                         WRITE 25 -> x
y = 6                    WRITE 6 -> y
y += 1                   READ y -> 6
                         (calculate 6+1 -> 7)
                         WRITE 7 -> y
```

## Conflict-Free Computation

- Suppose we divide this program into two separate processes, P1 and P2:

```
x = 5                    y = 6
x = square(x)            y += 1
```

| P1 | P2 |
|---|---|
| WRITE 5 -> x | WRITE 6 -> y |
| READ x -> 5 | READ y -> 6 |
| (calculate 5*5 -> 25) | (calculate 6+1 -> 7) |
| WRITE 25 -> x | WRITE 7 -> y |

```
                  x = 25
                  y = 7
```

- The result will be the same regardless of which process's READs and WRITEs happen first, because they reference different variables.

---

## Read-Write Conflicts

- Suppose that both processes read from x after it is initialized.

```
                  x = 5
```

| x = square(x) | y = x + 1 |
|---|---|

| P1 | P2 |
|---|---|
| READ x -> 5 | \| |
| (calculate 5*5 -> 25) | READ x -> 5 |
| WRITE 25 -> x | (calculate 5+1 -> 6) |
| \| | WRITE 6 -> y |

```
                  x = 25
                  y = 6
```

- The statements in P2 must appear in the given order, but they need not line up like this with statements in P1, because the execution of P1 and P2 is independent.

---

## Read-Write Conflicts (II)

- Here's another possible sequence of events

```
                  x = 5
```

| x = square(x) | y = x + 1 |
|---|---|

| P1 | P2 |
|---|---|
| READ x -> 5 | \| |
| (calculate 5*5 -> 25) | \| |
| WRITE 25 -> x | \| |
| \| | READ x -> 25 |
| \| | (calculate 25+1 -> 26) |
| \| | WRITE 26 -> y |

```
                  x = 25
                  y = 26
```

---

## Read-Write Conflicts (III)

- The problem here is that nothing forces P1 to wait for P2 to read x before setting it.
- Observation: The "calculate" lines have no effect on the outcome. They represent actions that are entirely local to one processor.
- The effect of "computation" is simply to delay one processor.
- But processors are assumed to be delayable by many factors, such as time-slicing (handing a processor over to another user's task), or processor speed.
- So the effect of computation adds nothing new to our simple model of shared-memory contention that isn't already covered by allowing any statement in one process to get delayed by any amount.
- So we'll just look at READ and WRITE in the future.

---

## Write-Write Conflicts

- Suppose both processes write to x:

```
                  x = 5
```

| x = square(x) | x = x + 1 |
|---|---|

| P1 | P2 |
|---|---|
| \| | READ x -> 5 |
| READ x -> 5 | \| |
| \| | WRITE 6 -> x |
| \| | \| |
| WRITE 25 -> x | |

```
                  x = 25
```

- This is a *write-write conflict:* two processes race to be the one that "gets the last word" on the value of x.

---

## Write-Write Conflicts (II)

```
                  x = 5
```

| x = square(x) | x = x + 1 |
|---|---|

| P1 | P2 |
|---|---|
| \| | READ x -> 5 |
| READ x -> 5 | \| |
| WRITE 25 -> x | \| |
| \| | WRITE 6 -> x |

```
                  x = 6
```

- This ordering is also possible; P2 gets the last word.
- There are also read-write conflicts here. What is the total number of possible final values for x? Four: 25, 5, 26, 36

## Coordinating Parallel Computation

Let's go back to bank accounts:

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance
    @property
    def balance(self): return self._balance
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
            return self._balance
```

```
                  acct = BankAccount(10)

acct.withdraw(8)                   acct.withdraw(7)
```
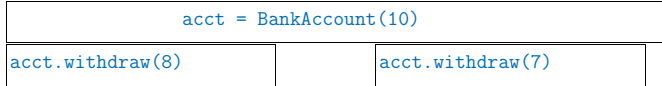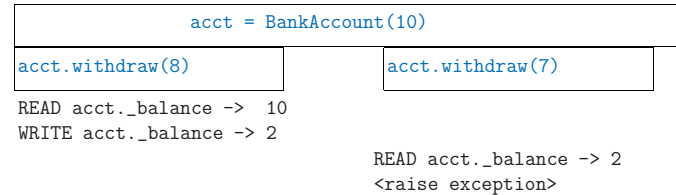
- At this point, we'd *like* to have the system raise an exception for one of the two withdrawals, and to set `acct.balance` to either 2 or 3, depending on with withdrawer gets to the bank first, like this...

---

## Desired Outcome

```python
class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
            return self._balance
```

```
                  acct = BankAccount(10)

acct.withdraw(8)                   acct.withdraw(7)

READ acct._balance ->  10
WRITE acct._balance -> 2
                             READ acct._balance -> 2
                             <raise exception>
```

But instead, we might get...

---

## Undesireable Outcome

```python
class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
            return self._balance
```
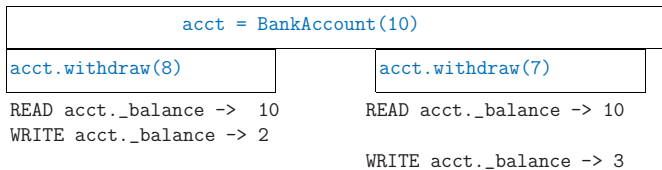
```
                  acct = BankAccount(10)

acct.withdraw(8)                   acct.withdraw(7)

READ acct._balance ->  10     READ acct._balance -> 10
WRITE acct._balance -> 2

                             WRITE acct._balance -> 3
```

Oops!

---

## Serializability

- We define the desired outcomes as those that would happen if withdrawals happened sequentially, in *some* order.
- The *nondeterminism* as to which order we get is acceptable, but results that are inconsistent with both orderings are not.
- These latter happen when operations overlap, so that the two processes see *inconsistent* views of the account.
- We want the withdrawal operation to act as if it is *atomic*—as if, once started, the operation proceeds without interruption and without any overlapping effects from other operations.

---

## One Solution: Critical Sections

- Some programming languages (e.g., Java) have special syntax for this. In Python, we can arrange something like this:

```python
def withdraw(amount):
    with CriticalSectionManager:
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
            return self._balance
```

- The `with` construct:
  1. Calls the `__enter__()` method of its "context manager" argument (here, some object we'll call `CriticalSectionManager`);
  2. Executes the body (indented portion);
  3. Finally, it calls the `__exit__()` method on the context manager. It guarantees that it will *always* do so, no matter how you exit from the body (via `return`, exception, etc.).
- The idea is that our *CriticalSectionManager* object should let just one process through at a time. How?

---

## Locks

- To implement our critical sections, we'll need some help from the operating system or underlying hardware.
- A common low-level construct is the *lock* or *mutex* (for "mutual exclusion"): an object that at any given time is "owned" by one process.
- If `L` is a lock, then
  - `L.acquire()` attempts to own `L` on behalf of the calling process. If someone else owns it, the caller *waits* for it to be release.
  - `L.release()` relinquishes ownership of `L` (if the calling process owns it).

## Implementing Critical Regions

- Using locks, it's easy to create the desired context manager:

```python
from threading import Lock

class CriticalSection:
    def __init__(self):
        self.__lock = Lock()

    def __enter__(self):
        self.__lock.acquire()

    def __exit__(self, exception_type, exception_val, traceback):
        self.__lock.release()
CriticalSectionManager = CriticalSection()
```

- The extra arguments to `__exit__` provide information about the exception, if any, that caused the **with** body to be exited.

- (In fact, the bare `Lock` type itself already has `__enter__` and `__exit__` procedures, so you don't really have to define an extra type).

## Granularity

- We've envisioned critical sections as being atomic with respect to *all* other critical sections.

- Has the advantage of simplicity and safety, but causes unnecessary waits.

- In fact, different accounts need not coordinate with each other. We can have a separate critical section manager (or lock) for each account object:

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance
        self._critical = CriticalSection()
    def withdraw(self, amount):
        with self._critical:
            ...
```

- That is, can produce a solution with finer *granularity* of locks.

## Synchronization

- Another kind of problem arises when different processes must communicate. In that case, one may have to wait for the other to send something.

- This, for example, doesn't work too well:

```python
class Mailbox:
    def __init__(self):
        self._queue = []
    def deposit(self, msg):
        self._queue.append(msg)
    def pickup(self):
        while not self._queue:
            pass
        return self._queue.pop()
```

- Idea is that one process deposits a message for another to pick up later.

- What goes wrong?

## Problems with the Naive Mailbox

```python
class Mailbox:
    def __init__(self):
        self._queue = []
    def deposit(self, msg):
        self._queue.append(msg)
    def pickup(self):
        while not self._queue:
            pass
        return self._queue.pop()
```

- *Inconsistency:* Two processes picking up mail can find the queue occupied simultaneously, but only one will succeed in picking up mail, and the other will get exception.

- *Busy-waiting:* The loop that waits for a message uses up processor time.

- *Deadlock:* If one is running two logical processes on one processor, busy-waiting can lead to nobody making any progress.

- *Starvation:* Even without busy-waiting one process can be shut out from ever getting mail.

## Conditions

- One way to deal with this is to augment locks with *conditions:*

```python
from threading import Condition
class Mailbox:
    def __init__(self):
        self._queue = []
        self._condition = Condition()
    def deposit(self, msg):
        with self._condition:
            self._queue.append(msg)
            self._condition.notify()
    def pickup(self):
        with self._condition:
            while not self._queue:
                self._condition.wait()
            return self._queue.pop()
```

- Conditions act like locks with methods `wait`, `notify` (and others).

- `wait` releases the lock, waits for someone to call `notify`, and then reacquires the lock.

## Another Approach: Messages

- Turn the problem inside out: instead of client processes deciding how to coordinate their operations on data, let the *data* coordinate its actions.

- From the Mailbox's perspective, things look like this:

```python
    self._queue = []
    while True:
        wait for a request, R, to deposit or pickup
        if R is a deposit of msg:
            self.__queue.append(msg)
            send back acknowledgement
        elif self.__queue and R is a pickup:
            msg = self.__queue.pop()
            send back msg
```

## Rendezvous

- Following ideas from C.A.R Hoare, the Ada language used the notion of a *rendezvous* for this purpose:

```
task type Mailbox is
    entry deposit(Msg: String);
    entry pickup(Msg: out String);
end Mailbox;

task body Mailbox is
    Queue: ...
begin
    loop
        select
          accept deposit(Msg: String) do Queue.append(Msg); end;
        or when not Queue.empty =>
          accept pickup(Msg: out String) do Queue.pop(Msg); end;
        end select;
    end loop;
end;
```