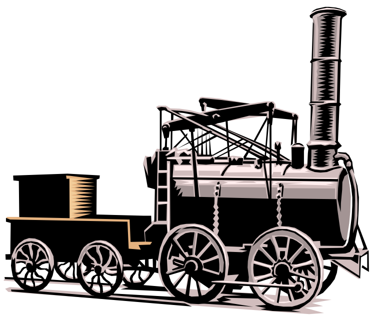
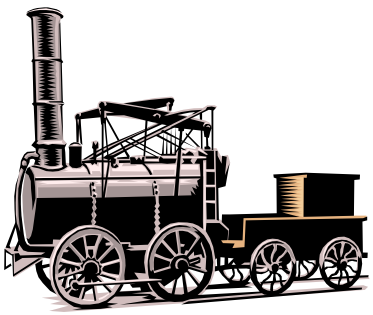
Shared-Memory Computability



Universal Object



Wait-free/Lock-free computable

=

Threads with methods that solve n-consensus

GetAndSet is not Universal

```
public class RMWRegister {
 private int value;
 public boolean getAndSet(int update)
 {
  int prior = this.value;
  this.value = update;
  return prior;
 }
}
```

GetAndSet is not Universal

```
public class RMWRegister {
 private int value;
 public boolean getAndSet(int update)
 {
  int prior = this.value;
  this.value = update;
  return prior;
 }
}
```

Consensus number 2

GetAndSet is not Universal

```
public class RMWRegister {
 private int value;
 public boolean getAndSet(int update)
 {
  int prior = this.value;
  this.value = update;
  return prior;
 }
}
```

Not universal for ≥ 3 threads

CompareAndSet is Universal

```
public class RMWRegister {
 private int value;
 public boolean
   compareAndSet(int expected,
                 int update) {
  int prior = this.value;
  if (this.value == expected) {
   this.value = update;
   return true;
  }
 return false;
 }}
```

CompareAndSet is Universal

```
public class RMWRegister {
 private int value;
 public boolean
   compareAndSet(int expected,
                 int update) {
  int prior = this.value;
  if (this.value == expected) {
   this.value = update;
   return true;
  }
 return false;
 }}
```

Consensus number ∞

CompareAndSet is Universal

```
public class RMWRegister {
 private int value;
 public boolean
   compareAndSet(int expected,
                 int update) {
  int prior = this.value;
  if (this.value == expected) {
   this.value = update;
   return true;
  }
 return false;
 }}
```

Universal for any number of threads

On Older Architectures

- IBM 360
  - testAndSet (getAndSet)
- NYU UltraComputer
  - getAndAdd
- Neither universal
  - Except for 2 threads

On Newer Architectures

- Intel x86, Itanium, SPARC
  - compareAndSet
- Alpha AXP, PowerPC
  - Load-locked/store-conditional
- All universal
  - For any number of threads
- Trend is clear …

Practical Implications

- Any architecture that does not provide a universal primitive has inherent limitations
- You cannot avoid locking for concurrent data structures …
- But why do we care?

Locking and Schedeuling

- What are the practical implications of locking?
- Locking affects the assumptions we need to make on the operating system in order to guarantee progress
- Lets understand how…

Schedeuling

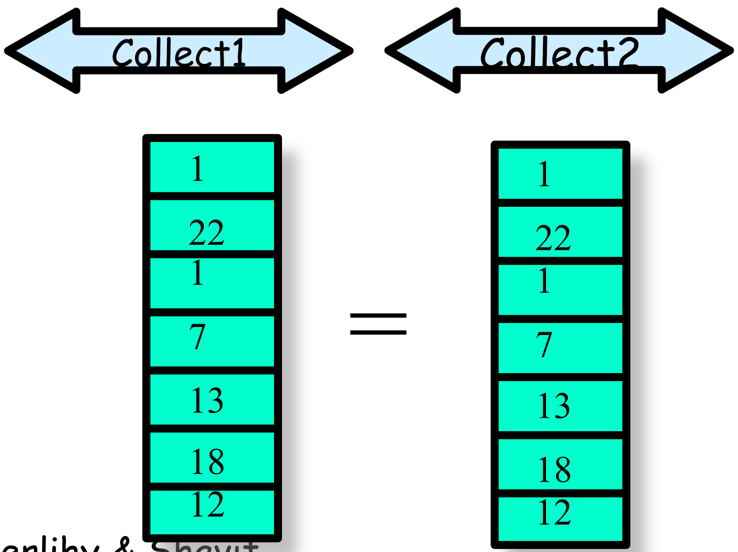
- The scheduler is a part of the OS that determines
  - Which thread gets to run on which processor
  - How long it runs for
- A given thread can thus be active, that is, executing instructions, or suspended

Review Progress Conditions

- Deadlock-free: <u>some</u> thread trying to acquire the locks eventually succeeds.
- Starvation-free: <u>every</u> thread trying to acquire the locks eventually succeeds.
- Lock-free: <u>some</u> thread calling the method eventually returns.
- Wait-free: <u>every</u> thread calling the method eventually returns.
- Obstruction-free: <u>every</u> thread calling the method returns if it executes in isolation for long enough.

The Simple Snapshot is Obstruction-Free

- Put increasing labels on each entry
- Collect twice
- If both agree,
  - We're done
- Otherwise,
  - Try again



Obstruction-freedom

- In the simple snapshot alg:
- The update method is wait-free
- But the scan is obstruction-free: will complete only if it executes for long enough without concurrent updates.

Progress of Methods

- Some of the above defs refer to locks (part of implementation) or method calls
- And they ignore the scheduler
- Lets refine our progress definitions so that they apply to methods, and
- Take scheduling into account

A "Periodic Table" of Progress Conditions

	Non-Blocking		Blocking
Everyone makes progress	Wait-free	Obstruction-free	Starvation-free
Someone makes progress	Lock-free		Deadlock-free

A bit more formally

- Standard notion of abstract object
- Progress conditions relate to method calls of an object
- Threads on a multiprocessor never fail:
  - A thread is active if it takes an infinite number of concrete (machine level) steps
  - And is suspended if not.

Maximal vs. Minimal

- For a given history H:
- Minimal progress: in every suffix of H, <u>some</u> method call eventually completes.
- Maximal progress: in every suffix of H, <u>every</u> method call eventually completes.

The "Periodic Table" of Progress Conditions

	Non-Blocking		Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free		Deadlock-free

The Scheduler's Role

On a multiprocessor progress properties:

- Are not about the guarantees a method's implementation provides.
- They are about the scheduling assumptions needed in order to provide minimal or maximal progress.

Fair Scheduling

- A history is fair if each thread takes an infinite number of steps

- A method implementation is deadlock-free if it guarantees minimal progress in every fair history, and maximal progress in some fair history.

Starvation Freedom

- A method implementation is starvation-free if it guarantees maximal progress in every fair history.

Dependent Progress

- A progress condition is dependent if it does not guarantee minimal progress in every history, and is independent if it does.
- The blocking progress conditions (deadlock-freedom, Starvation-freedom) are dependent

Non-blocking Independent Conditions

- A method implementation is lock-free if it guarantees minimal progress in every history, and maximal progress in some history.
- A method implementation is wait-free if it guarantees maximal progress in every history.

The "Periodic Table" of Progress Conditions

	Non-Blocking		Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free		Deadlock-free
	Independent		Dependent

Uniformly Isolating Schedules

- A history is uniformly isolating if, for every k > 0, any thread that takes an infinite number of steps has an interval where it takes at least k contiguous steps
- Modern systems provide ways of providing isolation…later we will learn about "backoff" and "yeild".

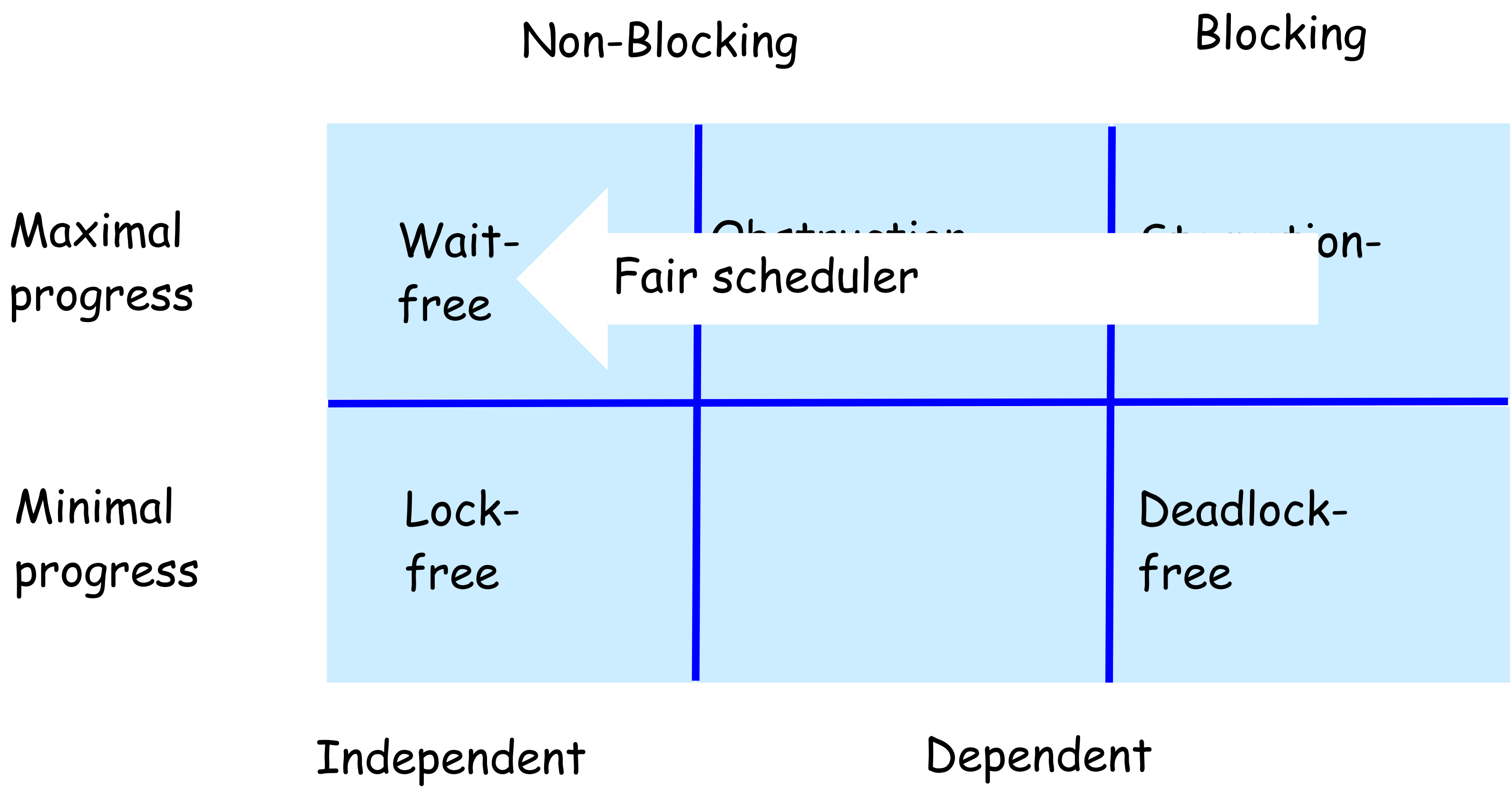
A Non-blocking Dependent Condition

- A method implementation is obstruction-free if it guarantees maximal progress in every uniformly isolating history.

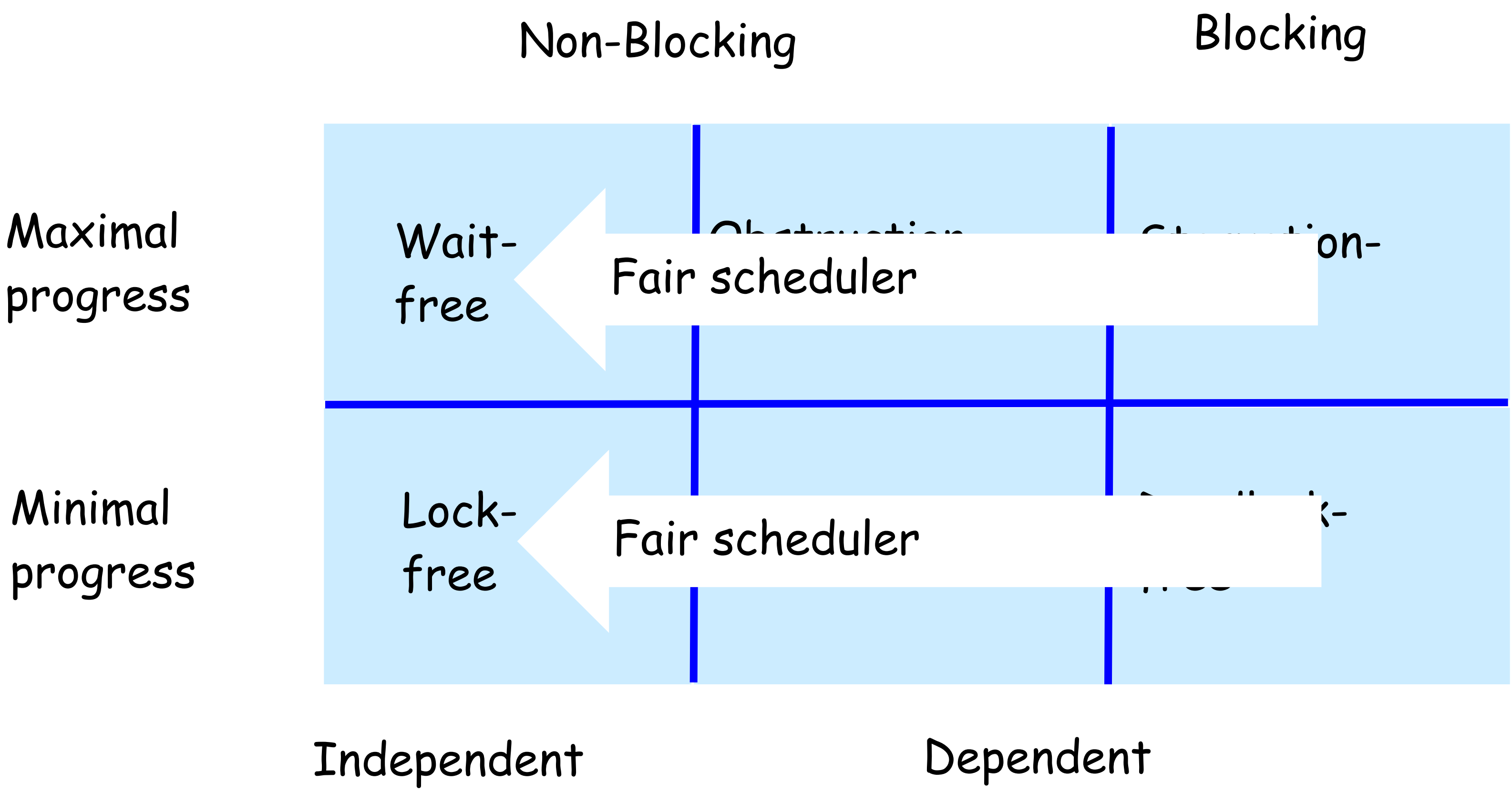
The "Periodic Table" of Progress Conditions

	Non-Blocking		Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free		Deadlock-free
	Independent	Dependent	

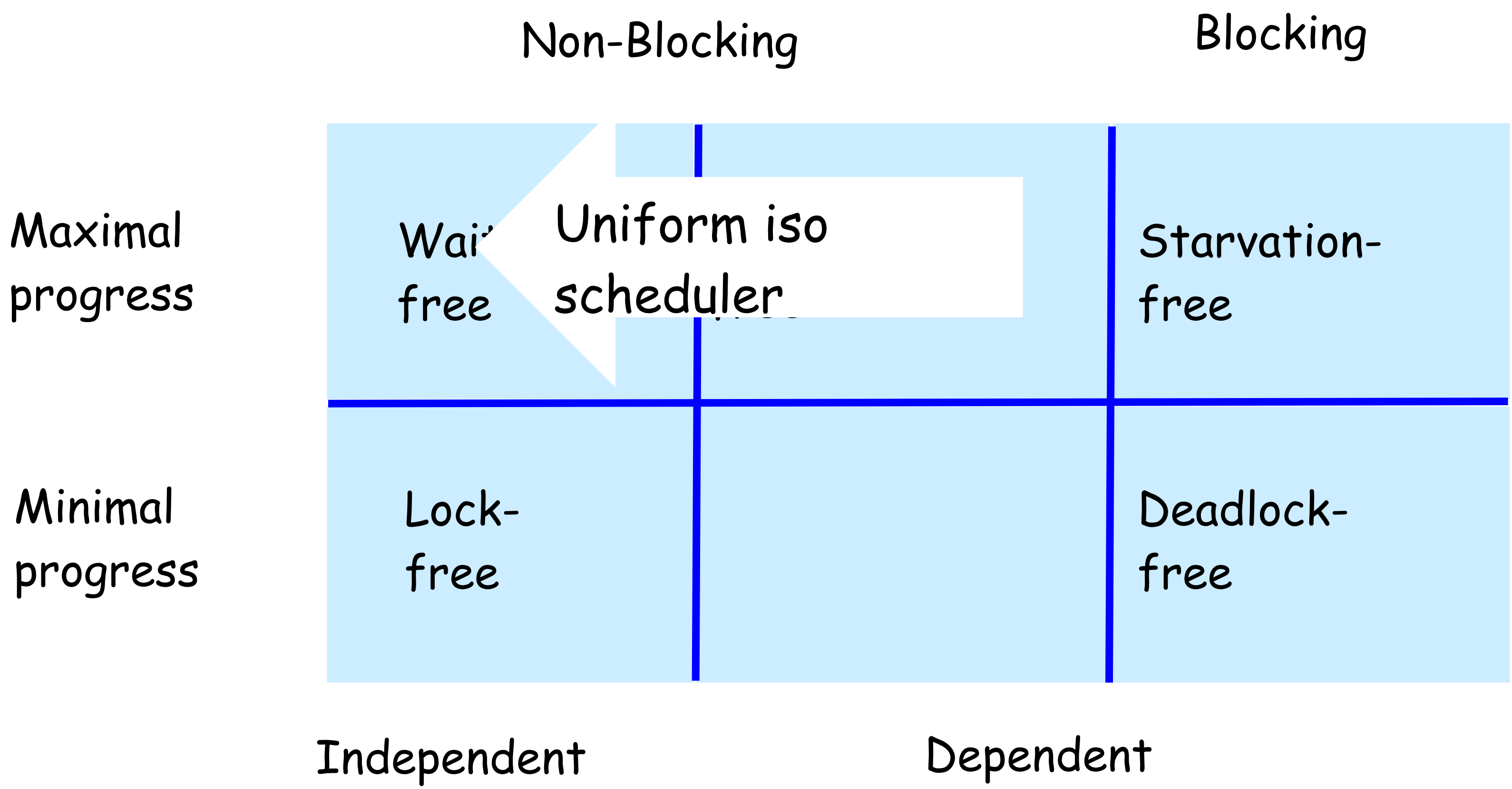
The "Periodic Table" of Progress Conditions



The “Periodic Table” of Progress Conditions



The “Periodic Table” of Progress Conditions



The “Periodic Table” of Progress Conditions

	Non-Blocking		Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free	?	Deadlock-free
	Independent	Dependent	

The “Periodic Table” of Progress Conditions

	Non-Blocking		Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free	Clash-free	Deadlock-free
	Independent	Dependent	

Clash-Freedom: the "Einsteinium" of Progress

Clash-Freedom: the "Einsteinium" of Progress

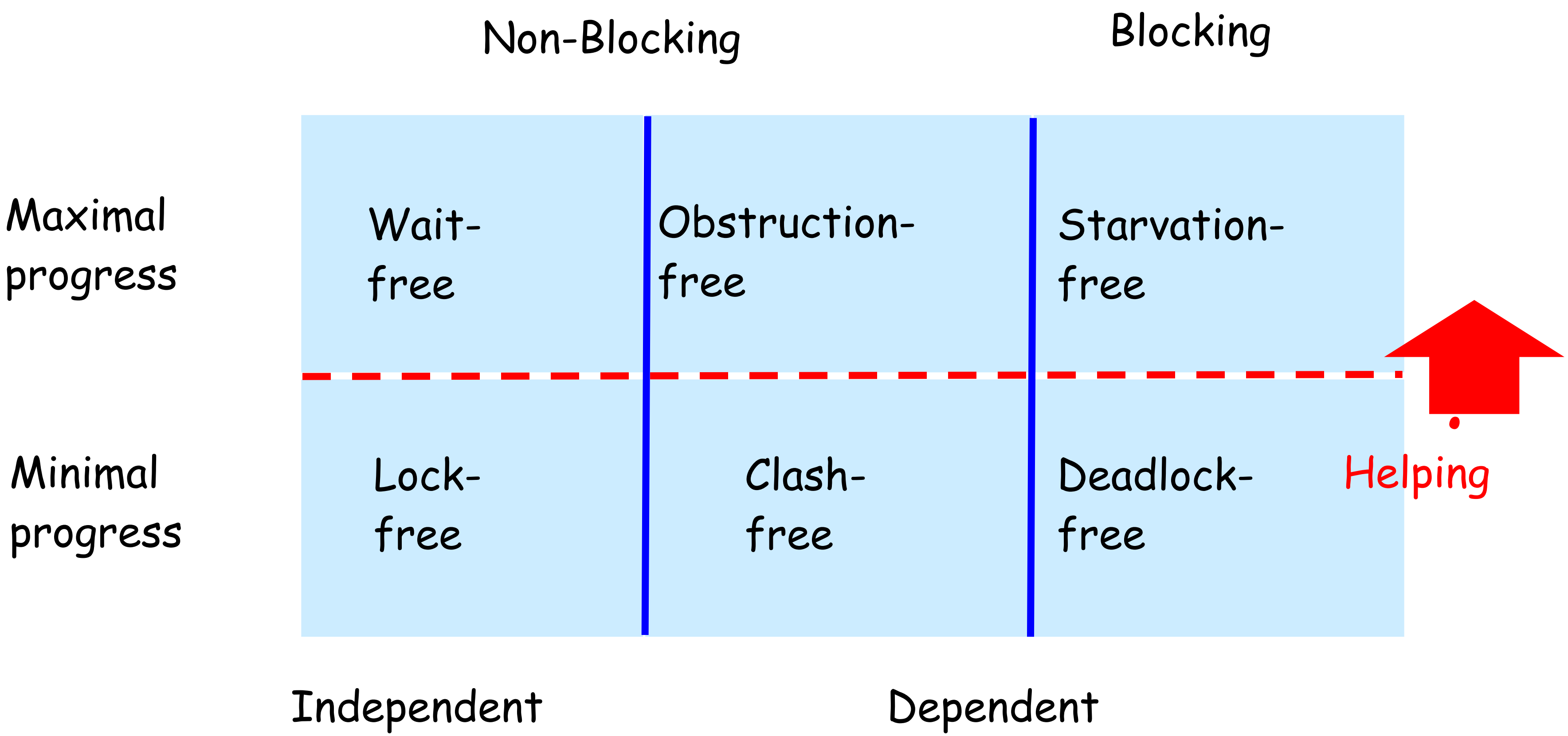
- A method implementation is clash-free if it guarantees <u>minimal</u> progress in every uniformly isolating history.
- Thm: clash-freedom strictly weaker than obstruction-freedom

Getting from Minimal to Maximal

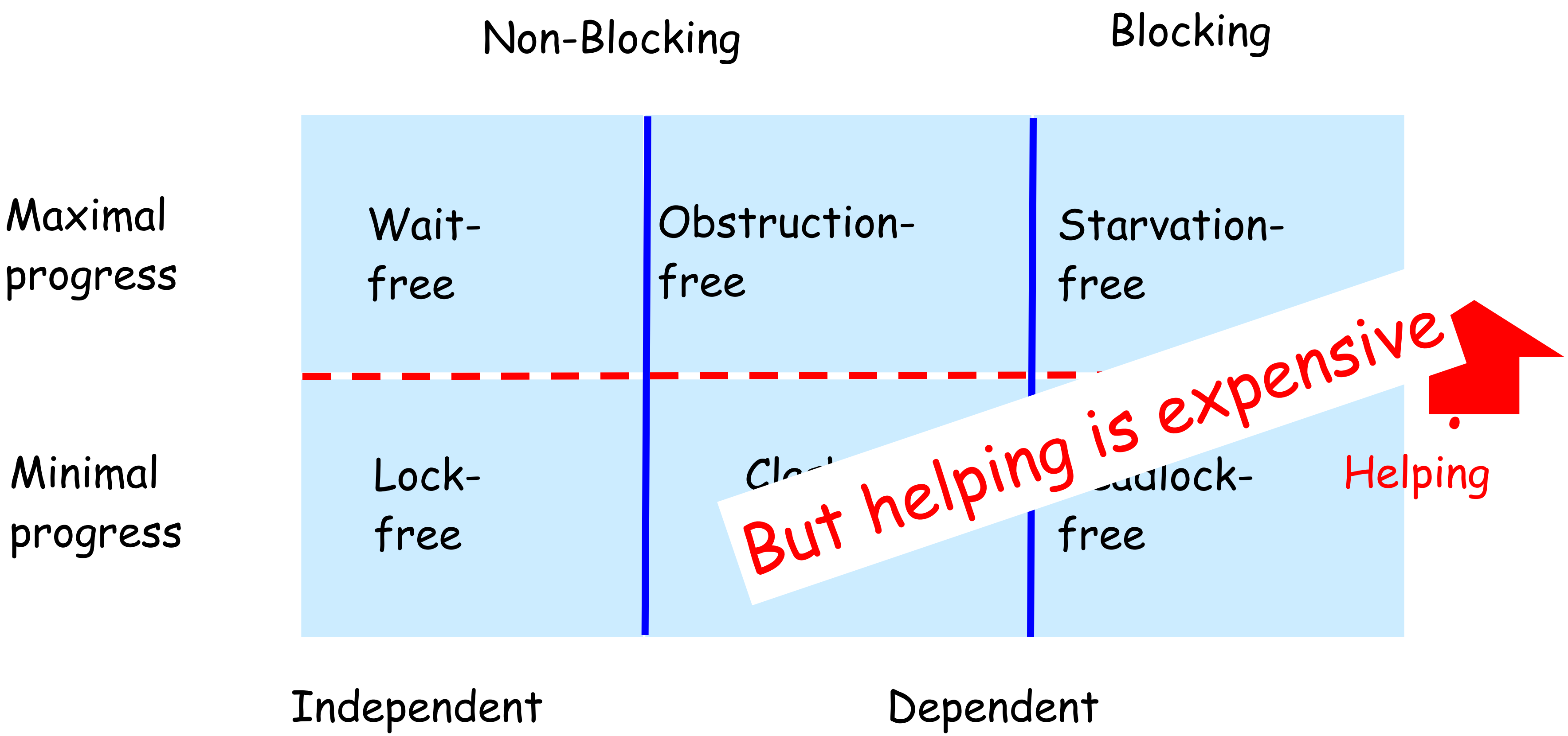
	Non-Blocking		Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free	Clash-free	Deadlock-free
	Independent	Dependent	

?

Getting from Minimal to Maximal



Getting from Minimal to Maximal



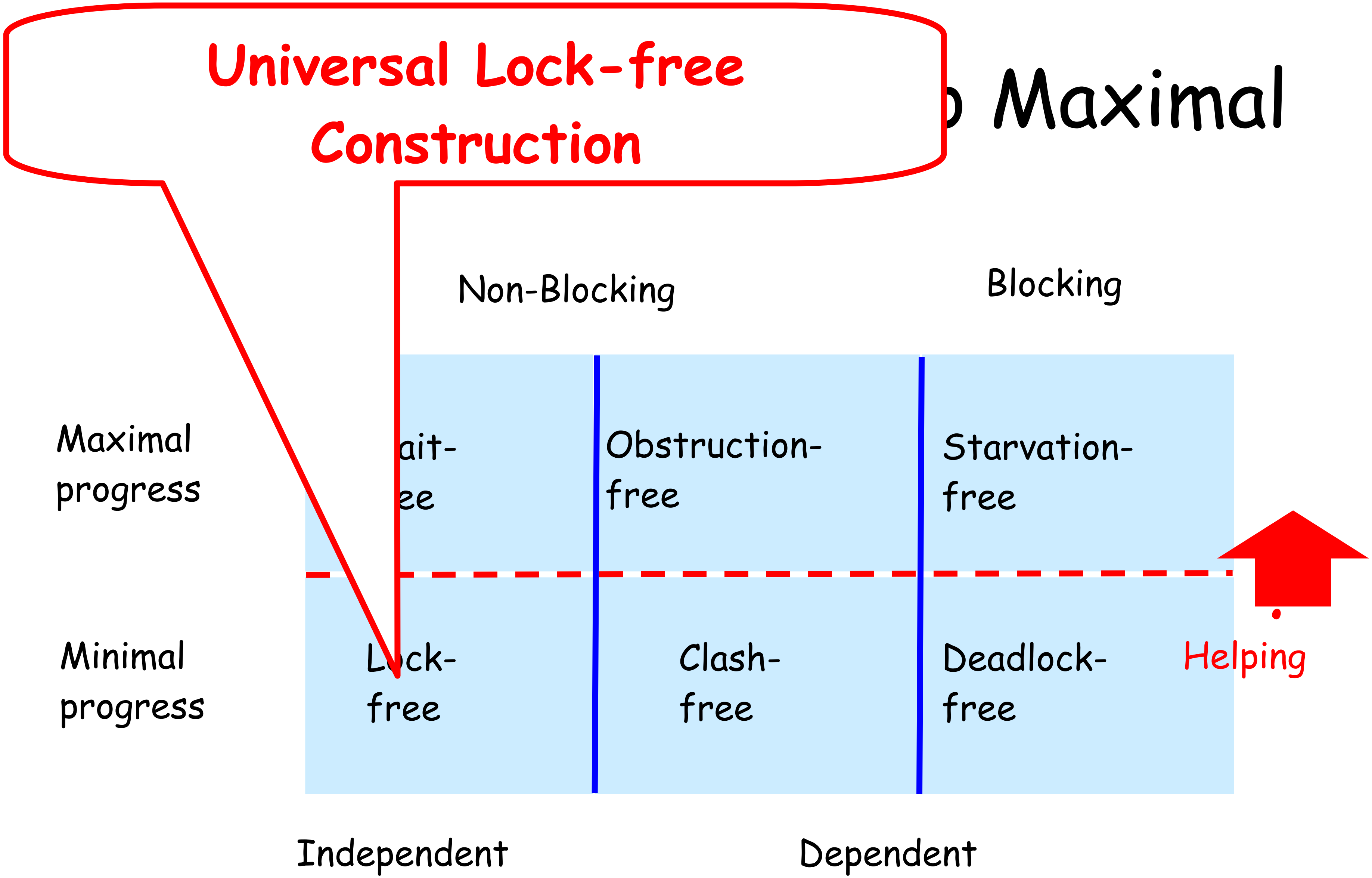
Universal Constructions

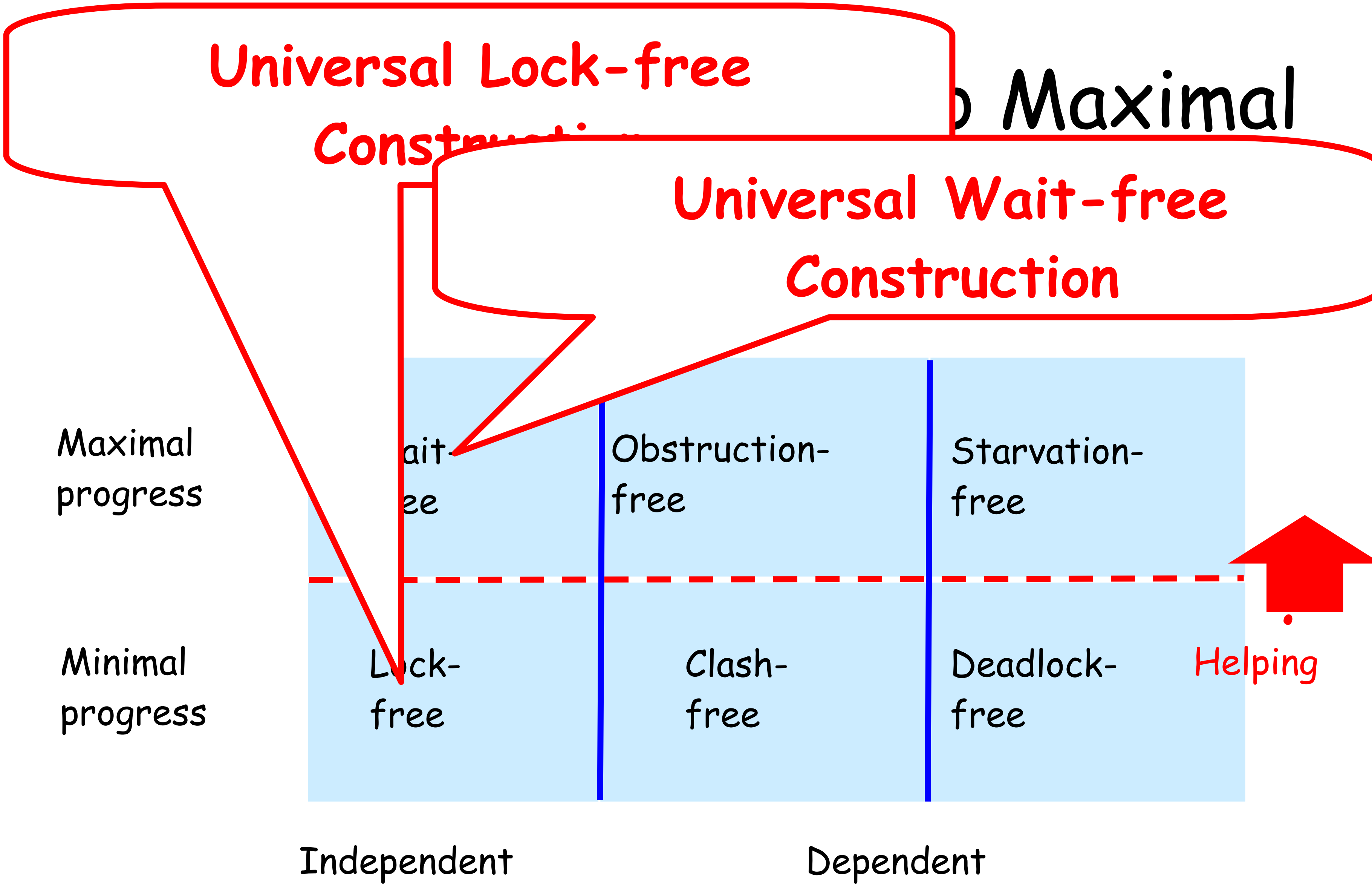
- Our lock-free universal construction provides minimal progress
- A scheduler is benevolent for that algorithm if it guarantees maximal progress in every allowable history.
- Many real-world operating system schedulers are benevolent
- They do not single out any indiviudual thread

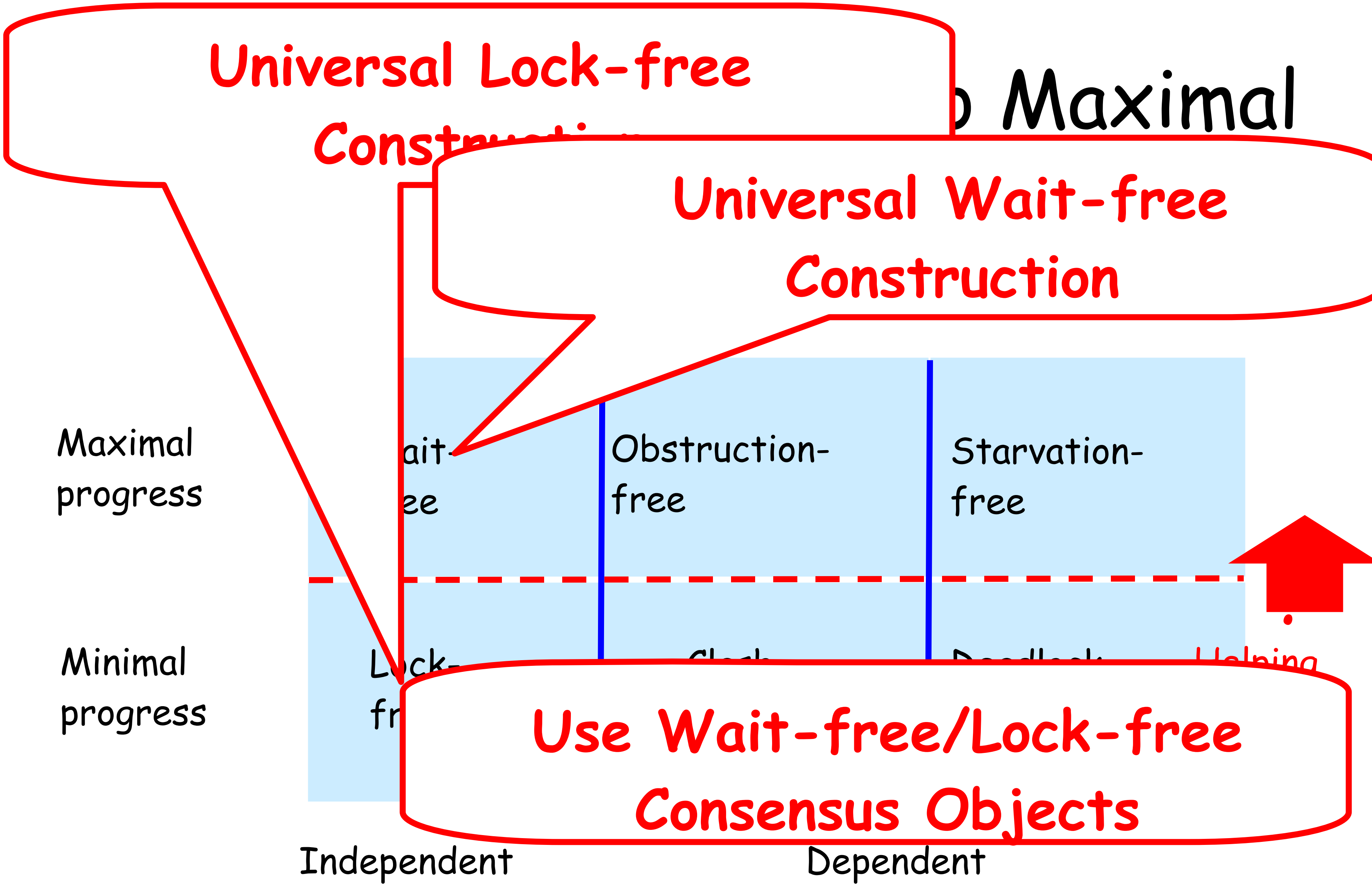
Getting from Minimal to Maximal

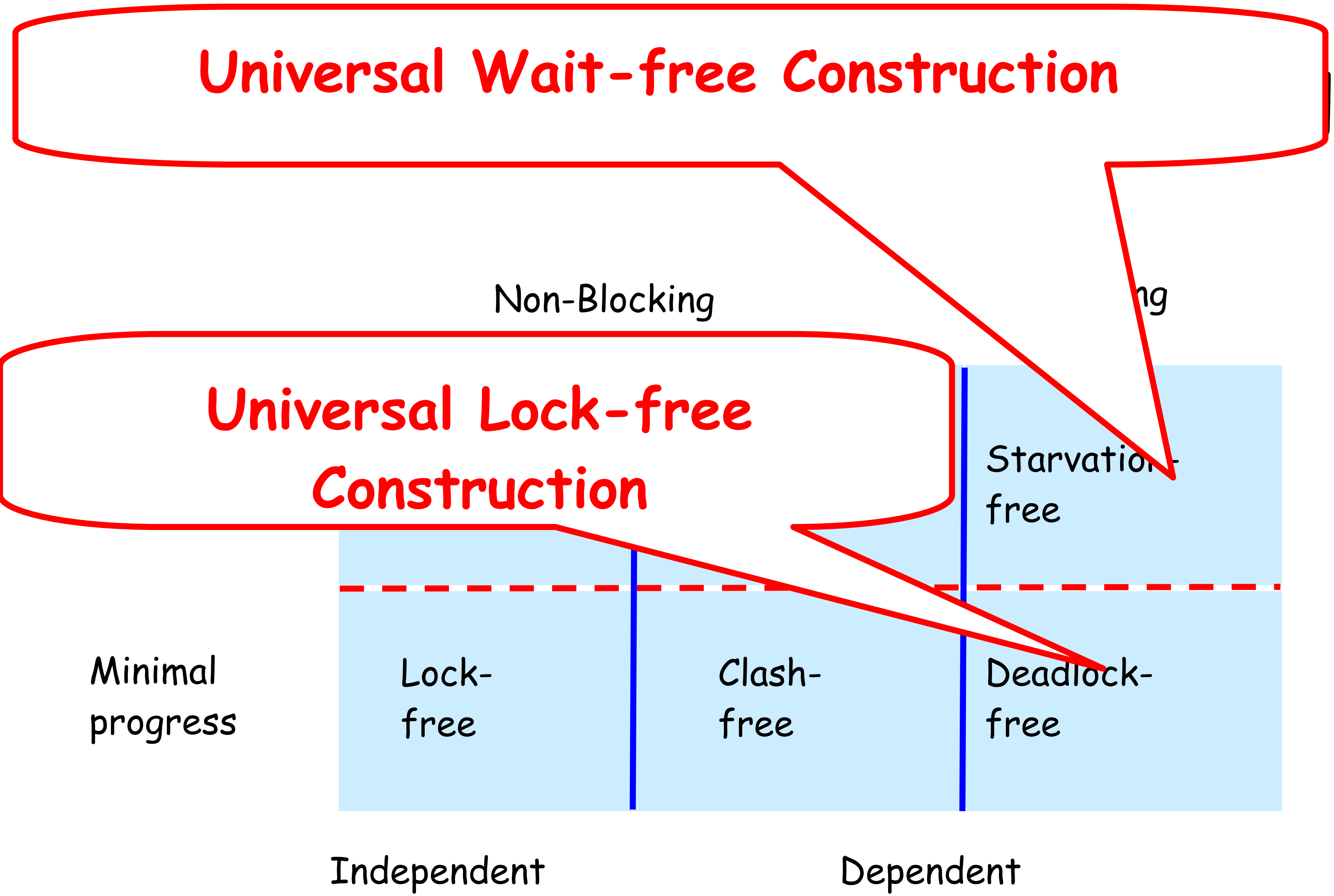
	Non-Blocking		Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free	Clash-free	Deadlock-free
	Independent	Dependent	

Helping


Universal Lock-free
Construction
o Maximal
Non-Blocking
Blocking
Maximal
progress
ait-
ee
Obstruction-
free
Starvation-
free
Minimal
progress
Lock-
free
Clash-
free
Deadlock-
free
Helping
Independent
Dependent


Maximal
Universal Lock-free
Universal Wait-free
Construction
Maximal progress
Minimal progress
Obstruction-free
Starvation-free
Lock-free
Clash-free
Deadlock-free
Helping
Independent
Dependent


Universal Lock-free Construction
Maximal
Universal Wait-free Construction
Maximal progress
Obstruction-free
Starvation-free
Minimal progress
Use Wait-free/Lock-free Consensus Objects
Independent
Dependent


Universal Wait-free Construction
Non-Blocking
Universal Lock-free Construction
Starvatio
free
Minimal progress
Lock-free
Clash-free
Deadlock-free
Independent
Dependent

Universal Wait-free Construction

Non-Blocking

Universal Lock-free Construction

Starvation-free

Minimal

Lock-

Clash-

Deadlock-

If we use Starvation-free/Deadlock-free Consensus Objects result is respectively Starvation-free/Deadlock-free

Benevolent Schedulers

- Consider an algorithm that guarantees minimal progress.
- A scheduler is benevolent for that algorithm if it guarantees maximal progress in every allowable history.
- Many real-world operating system schedulers are benevolent
- They do not single out any indiviudual thread

In Practice

On a multiprocessor we will write code expecting maximal progress.

Progress conditions will then define the scheduling assumptions needed in order to provide it.

This Means

We will mostly write lock-free and lock-based deadlock-free algorithms…

and expect them to behave as if they are wait-free…

because modern schedulers can be made benevolent and fair.

Principles to Practice

- We learned how to define the safety (correctness) and liveness (progress) of concurrent programs and objects
- We are ready to start the practice of implementing them
- Next lecture: implementing spin locks on multiprocesor machines…