Correctness of parallel programs

Shaz Qadeer
Research in Software Engineering

CSEP 506
Spring 2011

Why is correctness important?

- Software development is expensive
  - Testing and debugging significant part of the cost

- Testing and debugging of parallel and concurrent programs is even more difficult and expensive

The Heisenbug problem

- Sequential program: program execution fully determined by the input

- Parallel program: program execution may depend both on the input and the communication among the parallel tasks

- Communication dependencies are invariably not reproducible

Essence of reasoning about correctness

- Modeling
- Abstraction
- Specification
- Verification

What is a data race?

- An execution in which two tasks simultaneously access the same memory location

```
Parallel.For(0, filenames.Length, (int i) =>
            {
                int len = filenames[i].Length;
                count[len]++;
            });
```

Example

filenames.Length == 2
filenames[0].Length == filenames[1].Length == 1

```
Parallel.For(0, filenames.Length, (int i) =>
        {
           int len = filenames[i].Length;
           count[len]++;
        });
```

```
Task(0)

int len, t;

len = 1;
t = count[1];
t++;
count[1] = t;
```

```
Task(1)

int len, t;

len = 1;
t = count[1];
t++;
count[1] = t;
```

Two executions

E1

Task(0)	Task(1)
int len, t;	int len, t;
len = 1;	
	len = 1;
t = count[1];	
t++;	
count[1] = t;	
	t = count[1];
	t++;
	count[1] = t;

E2

Task(0)	Task(1)
int len, t;	int len, t;
len = 1;	
	len = 1;
t = count[1];	
t++;	
	t = count[1];
	t++;
count[1] = t;	
	count[1] = t;

Which execution has a data race?

Example

count.Length == 2

```
for (int i = 0; i < count.Length; i++) { count[i] = 0; }
Parallel.For(0, count.Length, (int i) => { count[i]++; });
for (int i = 0; i < count.Length; i++) { Console.WriteLine(count[0]); }
```

An execution

Task(0)	Parent()	Task(1)
int t;		int t;
	count[0] = 0;	
	count[1] = 0;	
t = count[0];		
t++;		
		t = count[1];
		t++;
count[0] = t;		
		count[1] = t;
	WriteLine(count[0]);	
	WriteLine(count[1]);	

What is a parallel execution?

- Happens-before graph: directed acyclic graph over the set of events in the execution

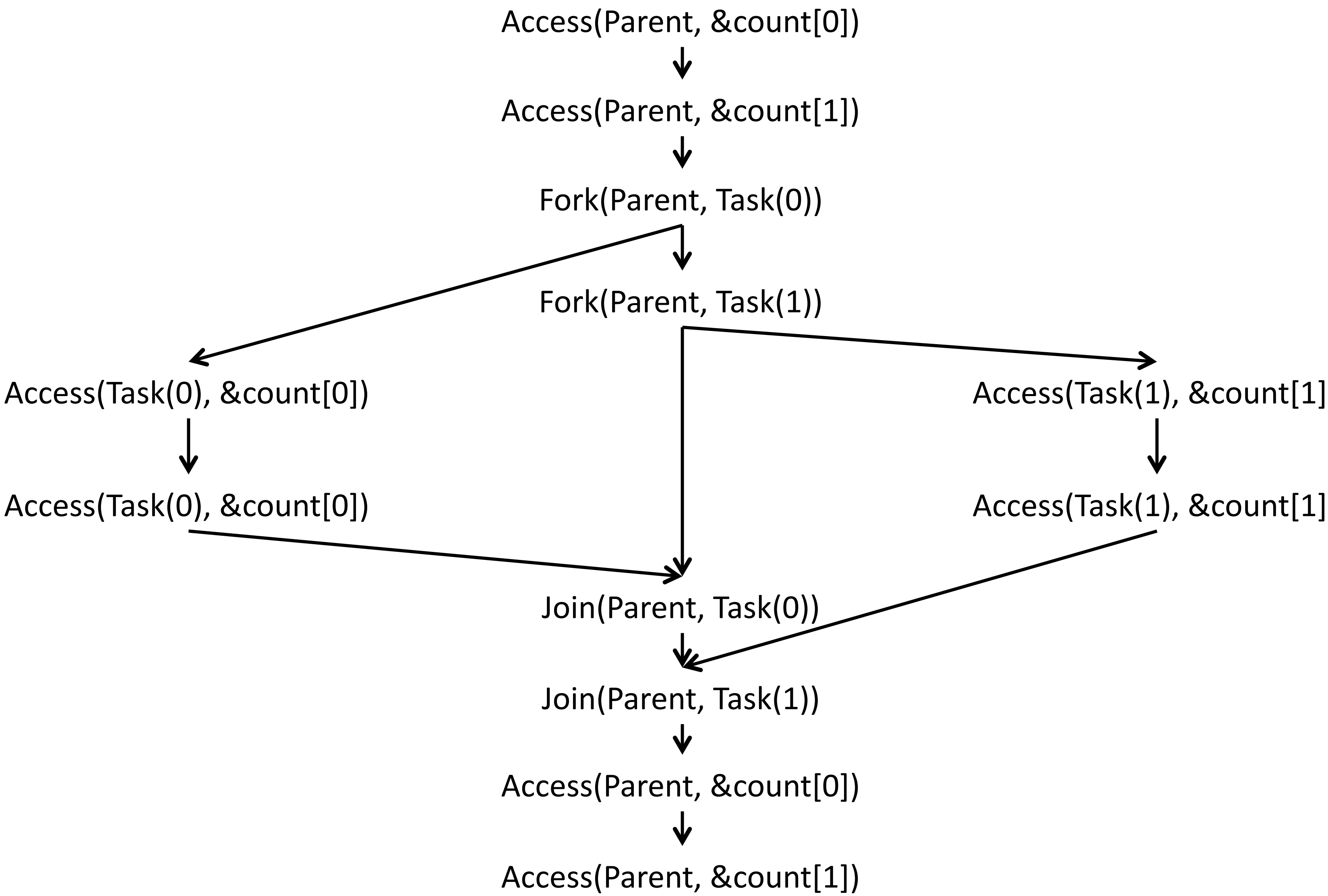
- Three kinds of events
  - Access(t, a): task t accessed address a
  - Fork(t, u): task t created task u
  - Join(t, u): task t waited for task u

- Three kinds of edges
  - program order: edge from an event by a particular task to subsequent event by the same task
  - fork: edge from Fork(t, u) to first event performed by task u
  - join: edge from last event performed by task u to Join(t, u)

A note on modeling executions

- A real execution on a live system is complicated
  - instruction execution on the CPU
  - scheduling in the runtime
  - hardware events, e.g., cache coherence messages
- Focus on what is relevant to the specification
  - memory accesses because data-races are about conflicting memory accesses
  - fork and join operations because otherwise we cannot reason precisely

Example of happens-before graph



Definition of data race

- e < f in an execution iff there is a path in the happens-before graph from e to f
  - e happens before f

- An execution has a data-race on address x iff there are different events e and f
  - both e and f access x
  - not e < f
  - not f < e

- An execution is race-free iff there is no data-race on any address x

Racy execution

```
Task(0)

int len, t;

len = 1;

t = count[1];
t++;
count[1] = t;
```

```
Task(1)

int len, t;


len = 1;



t = count[1];
t++;
count[1] = t;
```

Access(Task(0), &count[1]

↓

Access(Task(0), &count[1]

Access(Task(1), &count[1]

↓

Access(Task(1), &count[1]

```
Task(0)

int len, t;

len = 1;

t = count[1];
t++;


count[1] = t;
```

```
Task(1)

int len, t;


len = 1;



t = count[1];
t++;

count[1] = t;
```

Access(Task(0), &count[1]

↓

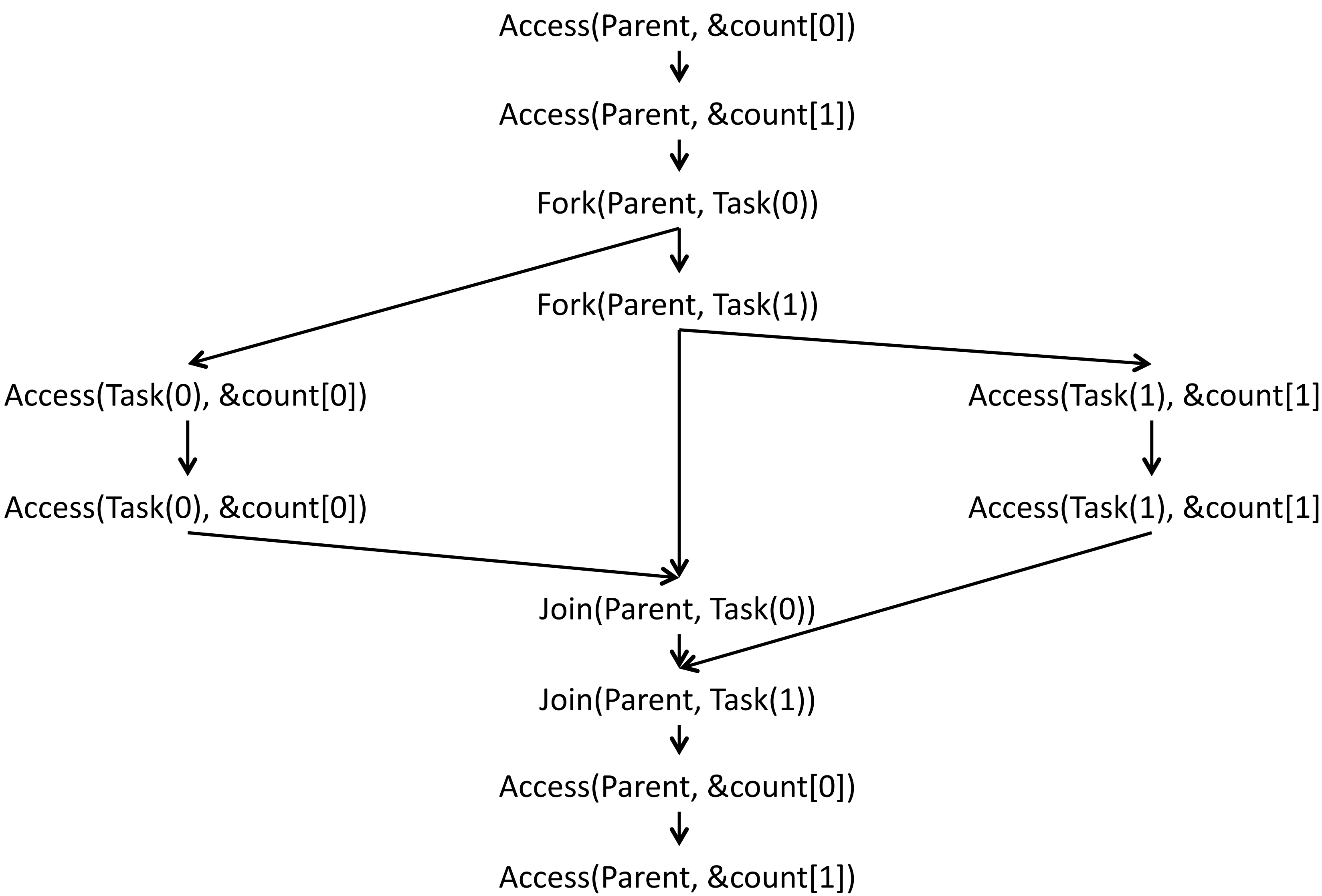
Access(Task(0), &count[1]

Access(Task(1), &count[1]

↓

Access(Task(1), &count[1]

Race-free execution



Vector-clock algorithm

- Vector clock: an array of integers indexed by task identifiers
- For each task t, maintain a vector clock C(t)
  - each clock in C(t) initialized to 0
- For each address a, maintain a vector clock X(a)
  - each clock in X(a) initialized to 0

Vector-clock operations

- Task t executes an event
  - increment C(t)[t] by one
- Task t forks task u
  - initialize C(u) to C(t)
- Task t joins with task u
  - update C(t) to max(C(t), C(u))
- Task t accesses address a
  - data race unless X(a) < C(t)
  - update X(a) to C(t)

	C(Parent)	C(Task(0))	C(Task(1))	X(&count[0])	X(&count[0])
	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]
Access(Parent, &count[0])	[1, 0, 0]			[1, 0, 0]	
Access(Parent, &count[1])	[2, 0, 0]				[2, 0, 0]
Fork(Parent, Task(0))	[3, 0, 0]	[3, 0, 0]			
Fork(Parent, Task(1))	[4, 0, 0]		[4, 0, 0]		
Access(Task(0), &count[0])		[3, 1, 0]		[3, 1, 0]	
Access(Task(1), &count[1])			[4, 0, 1]		[4, 0, 1]
Access(Task(0), &count[0])		[3, 2, 0]		[3, 2, 0]	
Access(Task(1), &count[1])			[4, 0, 2]		[4, 0, 2]
Join(Parent, Task(0))	[5, 2, 0]				
Join(Parent, Task(1))	[6, 2, 2]				
Access(Parent, &count[0])	[7, 2, 2]			[7, 2, 2]	
Access(Parent, &count[1])	[8, 2, 2]				[8, 2, 2]

	C(Task(0))	C(Task(1))	X(&count[0])	X(&count[0])
	[0, 0]	[0, 0]	[0, 0]	[0, 0]
Access(Task(0), &count[1])	[1, 0]			[1, 0]
Access(Task(1), &count[1])		[0, 1]		
Access(Task(0), &count[1])				
Access(Task(1), &count[1])				

Correctness of vector-clock algorithm

- For any execution and any two events e@t followed by f@u in the execution
  - e@t < f@u iff C[t]@e < C[u]@f

Synchronizing using locks

```
Parallel.For(0, filenames.Length, (int i) =>
        {
            int len = filenames[i].Length;
            lock (lock[len]) { count[len]++; }
        });
```

filenames.Length == 2
filenames[0].Length == filenames[1].Length == 1

```
count[1] = 0;
Parallel.For(0, filenames.Length, (int i) =>
        {
            int len = filenames[i].Length;
            lock (lock[len]) { count[len]++; }
        });
Console.WriteLine(count[1]);
```

Parent	Task(0)	Task(1)

`count[1] = 0;`
`Console.WriteLine(count[1]);`
	`int len, t;`
	
	`len = 1;`
	`acquire(lock[1]);`
	`t = count[1];`
	`t++;`
	`count[1] = t;`
	`release(lock[1]);`
		`int len, t;`
		
		`len = 1;`
		`acquire(lock[1]);`
		`t = count[1];`
		`t++;`
		`count[1] = t;`
		`release(lock[1]);`

Execution I

Parent	Task(0)	Task(1)
	int len, t;	int len, t;
count[1] = 0;		
	len = 1;	
		len = 1;
	acquire(lock[1]);	
	t = count[1];	
	t++;	
	count[1] = t;	
	release(lock[1]);	
		acquire(lock[1]);
		t = count[1];
		t++;
		count[1] = t;
		release(lock[1]);
Console.WriteLine(count[1]);		

Execution II

```
Parent



count[1] = 0;













Console.WriteLine(count[1]);
```

```
Task(0)

int len, t;

len = 1;






acquire(lock[1]);
t = count[1];
t++;
count[1] = t;
release(lock[1]);
```

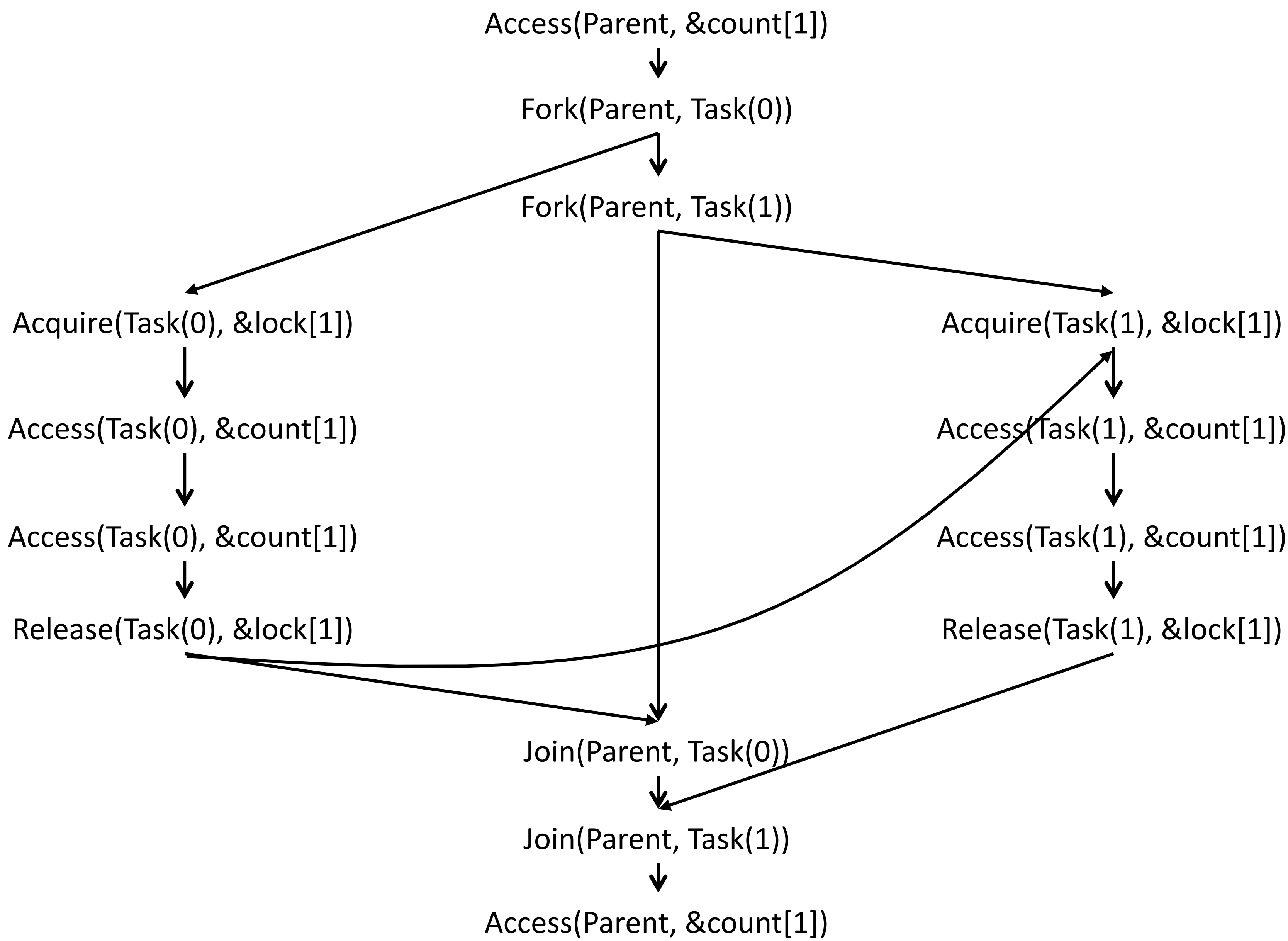
```
Task(1)

int len, t;


len = 1;
acquire(lock[1]);
t = count[1];
t++;
count[1] = t;
release(lock[1]);
```

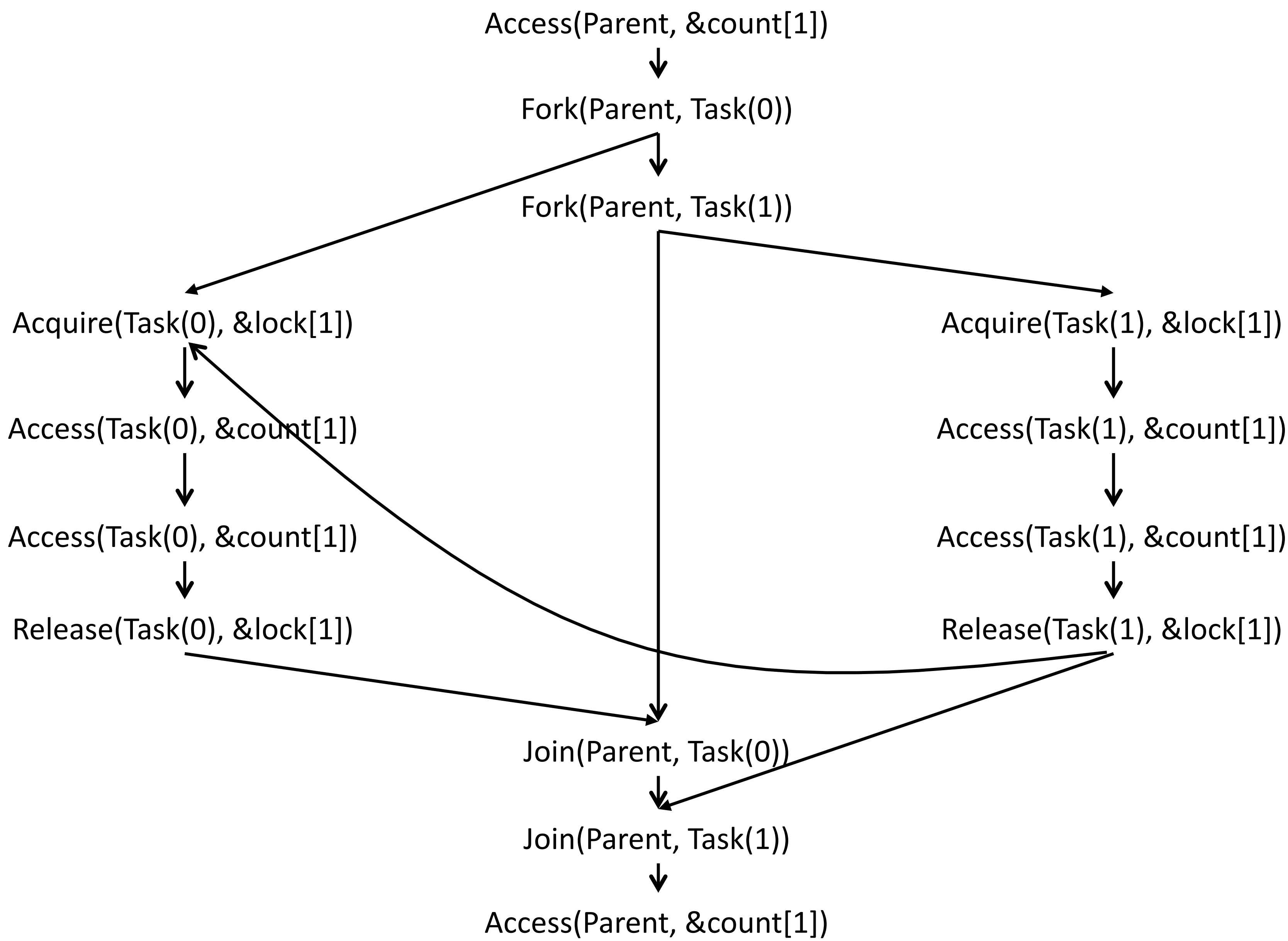
What is a parallel execution?

- Happens-before graph: directed acyclic graph over the set of events in an execution

- Five kinds of events
  - Access(t, a): task t accessed address a
  - Fork(t, u): task t created task u
  - Join(t, u): task t waited for task u
  - Acquire(t, l): task t acquired lock l
  - Release(t, l): task t released lock l

- Three kinds of edges
  - program order: edge from an event by a particular task to subsequent event by the same task
  - fork: edge from Fork(t, u) to first event performed by task u
  - join: edge from last event performed by task u to Join(t, u)
  - release-acquire: edge from Release(t, l) to subsequent Acquire(u, l)


Access(Parent, &count[1])
Fork(Parent, Task(0))
Fork(Parent, Task(1))
Acquire(Task(0), &lock[1])
Access(Task(0), &count[1])
Access(Task(0), &count[1])
Release(Task(0), &lock[1])
Acquire(Task(1), &lock[1])
Access(Task(1), &count[1])
Access(Task(1), &count[1])
Release(Task(1), &lock[1])
Join(Parent, Task(0))
Join(Parent, Task(1))
Access(Parent, &count[1])


Access(Parent, &count[1])
Fork(Parent, Task(0))
Fork(Parent, Task(1))
Acquire(Task(0), &lock[1])
Access(Task(0), &count[1])
Access(Task(0), &count[1])
Release(Task(0), &lock[1])
Acquire(Task(1), &lock[1])
Access(Task(1), &count[1])
Access(Task(1), &count[1])
Release(Task(1), &lock[1])
Join(Parent, Task(0))
Join(Parent, Task(1))
Access(Parent, &count[1])

Vector-clock algorithm extended

- Vector clock: an array of integers indexed by the set of tasks
- For each task t, maintain a vector clock C(t)
  - each clock in C(t) initialized to 0
- For each address a, maintain a vector clock X(a)
  - each clock in X(a) initialized to 0
- For each lock l, maintain a vector clock S(l)
  - each clock in S(l) initialized to 0

Vector-clock operations extended

- Task t executes an event
  - increment C(t)[t] by one
- Task t forks task u
  - initialize C(u) to C(t)
- Task t joins with task u
  - update C(t) to max(C(t), C(u))
- Task t accesses address a
  - data race unless X(a) < C(t)
  - update X(a) to C(t)
- Task t acquires lock l
  - update C(t) to max(C(t), S(l))
- Task t releases lock l
  - update S(l) to C(t)

	C(Parent)	C(Task(0))	C(Task(1))	S(&lock[1])	X(&count[1])
	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]	[0, 0, 0]
Access(Parent, &count[1])	[1, 0, 0]				[1, 0, 0]
Fork(Parent, Task(0))	[2, 0, 0]	[2, 0, 0]			
Fork(Parent, Task(1))	[3, 0, 0]		[3, 0, 0]		
Acquire(Task(0), &lock[1])		[2, 1, 0]			
Access(Task(0), &count[1])		[2, 2, 0]			[2, 2, 0]
Access(Task(0), &count[1])		[2, 3, 0]			[2, 3, 0]
Release(Task(0), &lock[1])		[2, 4, 0]		[2, 4, 0]	
Acquire(Task(1), &lock[1])			[3, 4, 1]		
Access(Task(1), &count[1])			[3, 4, 2]		[3, 4, 2]
Access(Task(1), &count[1])			[3, 4, 3]		[3, 4, 3]
Release(Task(1), &lock[1])			[3, 4, 4]	[3, 4, 4]	
Join(Parent, Task(0))	[4, 4, 0]				
Join(Parent, Task(1))	[5, 4, 4]				
Access(Parent, &count[1])	[6, 4, 4]				[6, 4, 4]