What is the smallest value of `counter`, among those listed, after the threads terminate?

```
int counter = 0;
```

thread `t`

```
int cnt;
for (int i = 0; i < 5; i++) {
  cnt = counter;
  counter = cnt + 1;
}
```

thread `u`

```
int cnt;
for (int i = 0; i < 5; i++) {
  cnt = counter;
  counter = cnt + 1;
}
```

1. 1
2. 5
3. 6
4. 10

What is the smallest value of `counter`, among those listed, after the threads terminate?

```
int counter = 0;
```

thread `t`

```
int cnt;
for (int i = 0; i < 5; i++) {
  cnt = counter;
  counter = cnt + 1;
}
```

thread `u`

```
int cnt;
for (int i = 0; i < 5; i++) {
  cnt = counter;
  counter = cnt + 1;
}
```

1. 1
2. [✓ 5]
3. 6
4. 10

The final value of `counter` is `5` when both threads read `counter == 0`, one thread proceeds and increments it to `5`, and the other thread overwrites the same values up to `5`.

But there are schedules where there is an even more destructive interference between the two threads, so that the final value of `counter` can be as low as 2!

| # | t's LOCAL | u's LOCAL | SHARED |
|---|---|---|---|
| 1 | $pc_t$ : 2 $cnt_t$ : ⊥ | $pc_u$ : 6 $cnt_u$ : ⊥ | counter : 0 |
| 2 | $pc_t$ : 2 $cnt_t$ : ⊥ | $pc_u$ : 7 $cnt_u$ : 0 | counter : 0 |
| 3 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 7 $cnt_u$ : 0 | counter : 0 |
| 4 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 6 $cnt_u$ : 0 | counter : 1 |
| 5 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 1 |
| 6 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 6 $cnt_u$ : 1 | counter : 2 |
| 7 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 7 $cnt_u$ : 2 | counter : 2 |
| 8 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 6 $cnt_u$ : 2 | counter : 3 |
| 9 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 7 $cnt_u$ : 3 | counter : 3 |
| 10 | $pc_t$ : 3 $cnt_t$ : 0 | $pc_u$ : 6 $cnt_u$ : 3 | counter : 4 |
| 11 | $pc_t$ : 2 $cnt_t$ : 0 | $pc_u$ : 6 $cnt_u$ : 3 | counter : 1 |
| 12 | $pc_t$ : 2 $cnt_t$ : 0 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 1 |
| 13 | $pc_t$ : 3 $cnt_t$ : 1 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 1 |
| 14 | $pc_t$ : 2 $cnt_t$ : 1 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 2 |
| 15 | $pc_t$ : 3 $cnt_t$ : 2 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 2 |
| 16 | $pc_t$ : 2 $cnt_t$ : 2 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 3 |
| 17 | $pc_t$ : 3 $cnt_t$ : 3 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 3 |
| 18 | $pc_t$ : 2 $cnt_t$ : 3 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 4 |
| 19 | $pc_t$ : 3 $cnt_t$ : 4 | $pc_u$ : 7 $cnt_u$ : 1 | counter : 4 |
| 20 | done | $pc_u$ : 7 $cnt_u$ : 1 | counter : 5 |
| 21 | done | done | counter : 2 |

What is the value of `n` after 8 concurrent threads terminate?

```
int n = 0; Semaphore s = new Semaphore(1); // capacity 1
```

thread $t_k$

```
int x;
s.down();
x = n;
n = x + 1;
s.up();
```

1. Between 1 and 8
2. Between 4 and 8
3. Always 4
4. Always 8

What is the value of `n` after 8 concurrent threads terminate?

```
int n = 0; Semaphore s = new Semaphore(1); // capacity 1
```

thread $\texttt{t}_k$

```
int x;
s.down();
x = n;
n = x + 1;
s.up();
```

1. Between 1 and 8
2. Between 4 and 8
3. Always 4
4. [✓ Always 8]

## What is the value of `n` after 8 concurrent threads terminate?

```
int n = 0; Semaphore s = new Semaphore(2); // capacity 2
```

thread $t_k$

```
int x;
s.down();
x = n;
n = x + 1;
s.up();
```

1. Between 1 and 8
2. Between 4 and 8
3. Always 4
4. Always 8

## What is the value of `n` after 8 concurrent threads terminate?

```
int n = 0; Semaphore s = new Semaphore(2); // capacity 2
```

thread $t_k$

```
int x;
s.down();
x = n;
n = x + 1;
s.up();
```

1. Between 1 and 8
2. Between 4 and 8
3. Always 4
4. Always 8

The value 1 occurs if one thread *t* reads 0 initially, and then waits inside its critical section, while the other threads go through their critical section in mutual exclusion. Then, *t* finishes by writing 1, thus overwriting the increments of all other threads.

What do threads continuously calling `x()` and `y()` print?

```
monitor class CountPrint {
  private Condition isX = new Condition();
  private Condition isY = new Condition();
  public void x()
  { isX.wait(); System.out.print("X"); isY.signal(); }
  public void y()
  { isY.wait(); System.out.print("Y"); isX.signal(); }
}
```

1. A sequence of alternating `X` and `Y`.
2. The first answer, if the monitor uses “signal and wait”.
3. The first answer, if the monitor uses “signal and continue”.
4. The program deadlocks.

What do threads continuously calling `x()` and `y()` print?

```
monitor class CountPrint {
  private Condition isX = new Condition();
  private Condition isY = new Condition();
  public void x()
  { isX.wait(); System.out.print("X"); isY.signal(); }
  public void y()
  { isY.wait(); System.out.print("Y"); isX.signal(); }
}
```

1. A sequence of alternating `X` and `Y`.
2. The first answer, if the monitor uses “signal and wait”.
3. The first answer, if the monitor uses “signal and continue”.
4. The program deadlocks.

What do threads continuously calling `x()` and `y()` print?

```
monitor class CountPrint {
  private Condition isY = new Condition();
  public void x()
  { System.out.print("X"); isY.signal(); }
  public void y()
  { isY.wait(); System.out.print("Y"); }
}
```

1. A sequence with at least one `X` between every pair of `Y`.
2. The first answer, if the monitor uses “signal and wait”.
3. The first answer, if the monitor uses “signal and continue”.
4. The program deadlocks.

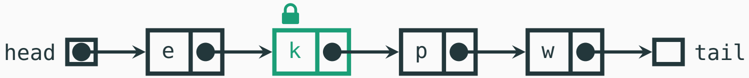
What do threads continuously calling `x()` and `y()` print?

```
monitor class CountPrint {
  private Condition isY = new Condition();
  public void x()
  { System.out.print("X"); isY.signal(); }
  public void y()
  { isY.wait(); System.out.print("Y"); }
}
```

1. A sequence with at least one `X` between every pair of `Y`.
2. The first answer, if the monitor uses “signal and wait”.
3. The first answer, if the monitor uses “signal and continue”.
4. The program deadlocks.

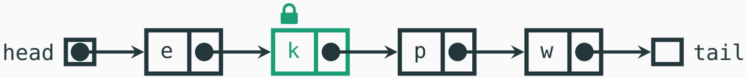
Under “signal and continue”, it is possible that two unblocked calls to `y()` get in the entry queue and then execute one after another.

Thread `t` is adding a node `m` while locking node `k`. How can this operation go wrong?



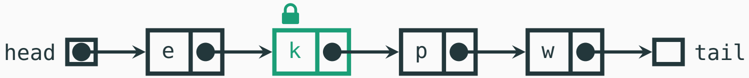
1. Another thread may add a node `n` before `m`
2. Another thread may add a node `g` before `k`
3. Another thread may remove node `k`
4. Another thread may invalidate node `k`

Thread `t` is adding a node `m` while locking node `k`. How can this operation go wrong?



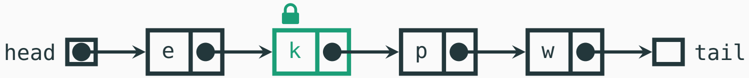
1. Another thread may add a node `n` before `m`
2. Another thread may add a node `g` before `k`
3. Another thread may remove node `k`
4. Another thread may invalidate node `k`

Thread `t` is removing node `p` while locking node `k`. How can this operation go wrong?



1. Another thread may add a node `m` after `k`
2. Another thread may add a node `q` after `p`
3. Another thread may remove node `w`
4. Another thread may add a node `g` before `k`

Thread `t` is removing node `p` while locking node `k`. How can this operation go wrong?



1. Another thread may add a node `m` after `k`
2. Another thread may add a node `q` after `p`
3. Another thread may remove node `w`
4. Another thread may add a node `g` before `k`

What does `twice([1,2,3,4])` return?

```
twice([]) -> [];
twice([H|T]) -> [2*H|twice(T)].
```

1. `[1,2,3,4]`
2. `[4,3,2,1]`
3. `[2,4,6,8]`
4. `[2,2,2,2]`

What does `twice([1,2,3,4])` return?

```
twice([]) -> [];
twice([H|T]) -> [2*H|twice(T)].
```

1. `[1,2,3,4]`
2. `[4,3,2,1]`
3. `[2,4,6,8]`
4. `[2,2,2,2]`

What does `mtwice([1,2,3,4])` return?

```
mtwice(L) -> map(fun (X) -> 2*X end, L).
```

1. `[2,4,6,8]`
2. `[2,2,2,2]`
3. The list `[2,4,6,8]` with the elements in any order
4. The list `[1,2,3,4]` with the elements in any order

What does `mtwice([1,2,3,4])` return?

```
mtwice(L) -> map(fun (X) -> 2*X end, L).
```

1. `[2,4,6,8]`
2. `[2,2,2,2]`
3. The list `[2,4,6,8]` with the elements in any order
4. The list `[1,2,3,4]` with the elements in any order

## What does process `Q` print?

| process `P` | process `Q` |
|---|---|
| `p() ->  % Q is Q's pid` | `q() ->  % P is P's pid` |
| `  Q ! {self(), 1},` | `  receive {P, N} ->` |
| `  Q ! {self(), 2},` | `    io:format("~p", [2*N]) end,` |
| `  Q ! {self(), 3}.` | `  q().` |

1. The numbers `1`, `2`, `3` in any order
2. The numbers `2`, `4`, `6` in any order
3. The numbers `1`, `2`, `3` in this order
4. The numbers `2`, `4`, `6` in this order

## What does process `Q` print?

| process `P` | process `Q` |
|---|---|
| `p() ->  % Q is Q's pid` | `q() ->  % P is P's pid` |

```erlang
% process P
p() ->  % Q is Q's pid
  Q ! {self(), 1},
  Q ! {self(), 2},
  Q ! {self(), 3}.
```

```erlang
% process Q
q() ->  % P is P's pid
  receive {P, N} ->
    io:format("~p", [2*N]) end,
  q().
```

1. The numbers `1`, `2`, `3` in any order
2. The numbers `2`, `4`, `6` in any order
3. The numbers `1`, `2`, `3` in this order
4. The numbers `2`, `4`, `6` in this order

## What does process R print?

process P

```
p() ->
  R ! x,
  p().
```

process Q

```
q() ->
  R ! y,
  q().
```

process R

```
r() ->
  receive
    x -> io.format("X");
    y -> io.format("Y")
  end,
  r().
```

1. The sequence XYXYXY. . . .
2. The sequence YXYXYX. . . .
3. Any sequence of letters X and Y.
4. Any sequence of uppercase letters.

## What does process R print?

process P

```
p() ->
  R ! x,
  p().
```

process Q

```
q() ->
  R ! y,
  q().
```

process R

```
r() ->
  receive
    x -> io.format("X");
    y -> io.format("Y")
  end,
  r().
```

1. The sequence XYXYXY. . . .
2. The sequence YXYXYX. . . .
3. Any sequence of letters X and Y.
4. Any sequence of uppercase letters.

## What does process R print?

process P

```
p() ->
  R ! x,
  p().
```

process Q

```
q() ->
  R ! y,
  q().
```

process R

```
r() ->
  receive
    x -> io.format("X")
  end,
  receive
    y -> io.format("Y")
  end,
  r().
```

1. The sequence XYXYXY. . . .
2. The sequence YXYXYX. . . .
3. Any sequence of letters X and Y.
4. Any sequence of uppercase letters.

## What does process R print?

process P

```
p() ->
  R ! x,
  p().
```

process Q

```
q() ->
  R ! y,
  q().
```

process R

```
r() ->
  receive
    x -> io.format("X")
  end,
  receive
    y -> io.format("Y")
  end,
  r().
```

1. The sequence XYXYXY. . . .
2. The sequence YXYXYX. . . .
3. Any sequence of letters X and Y.
4. Any sequence of uppercase letters.

What does `ptwice([1,2,3,4], [], 4)` return?

```
ptwice([], R, 0) ->
  R;
ptwice([], R, N) ->
  receive X -> ptwice([], [X|R], N-1) end;
ptwice([H|T], R, N) ->
  Me = self(),
  spawn(fun ()-> Me ! 2*H end),
  ptwice(T, R, N).
```

1. `[2,4,6,8]`
2. `[8,6,4,2]`
3. The list `[2,4,6,8]` with the elements in any order
4. The list `[1,2,3,4]` with the elements in any order

What does `ptwice([1,2,3,4], [], 4)` return?

```
ptwice([], R, 0) ->
  R;
ptwice([], R, N) ->
  receive X -> ptwice([], [X|R], N-1) end;
ptwice([H|T], R, N) ->
  Me = self(),
  spawn(fun ()-> Me ! 2*H end),
  ptwice(T, R, N).
```

1. `[2,4,6,8]`
2. `[8,6,4,2]`
3. The list `[2,4,6,8]` with the elements in any order
4. The list `[1,2,3,4]` with the elements in any order

Each send is executed by a different spawned process; hence there is no guarantee on the receiving order.