# Efficient Code Generation from SHIM Models

Stephen A. Edwards and Olivier Tardieu

Columbia University

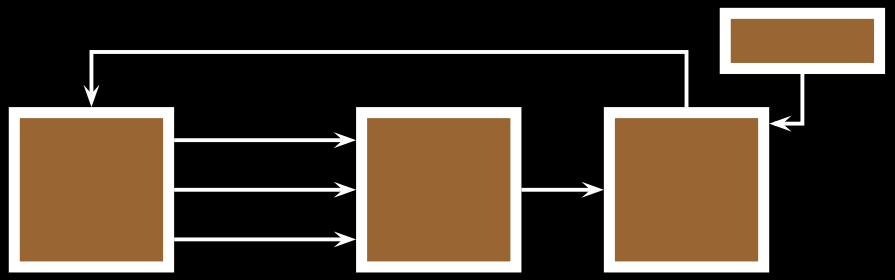
# Definition

**shim** \'shim\ *n*

1 : a thin often tapered piece of material (as wood, metal, or stone) used to fill in space between things (as for support, leveling, or adjustment of fit).

2 : *Software/Hardware Integration Medium*, a model for describing hardware/software systems

# The SHIM Model



Sequential processes

Unbuffered point-to-point communication channels exchange data tokens

Fixed topology

Asynchronous

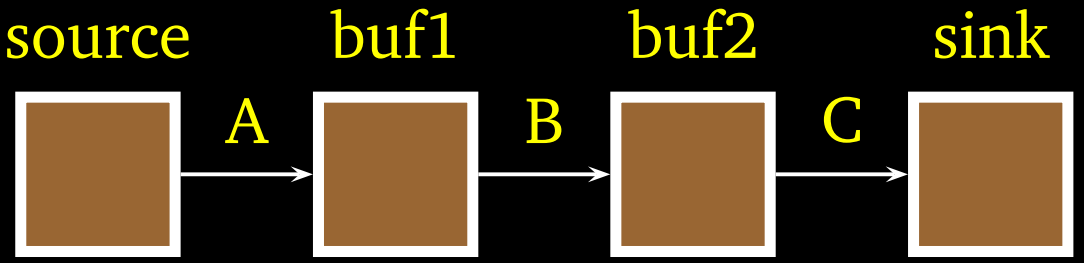
Synchronous communication events

Delay-insensitive: sequence of data through any channel independent of scheduling policy (the Kahn principle)

“Kahn networks with rendezvous communication”

# An Example

```
void main() {
  uint8 A, B, C;
  {                    // source: generate four values
    next A = 17;
    next A = 42;
    next A = 157;
    next A = 8;
  } par {              // buf1: copy from input to output
    for (;;)
      next B = next A;
  } par {              // buf2: copy, add 1 alternately
    for (;;) {
      next C = next B;
      next C = next B + 1;
    }
  } par {              // sink
    for (;;)
      next C;
  }
}
```



# Faking Concurrency in C

One function

```
void run() {
  for (;;) {
    switch (pc1) {
    case 0: block A
            pc1 = 1;
            break;
    case 1: block C
    }

    switch (pc2) {
    case 0: block B
            pc2 = 1;
            break;
    case 1: block D
    }
  }
}
```

# Faking Concurrency in C

One function

```
void run() {
  for (;;) {
    switch (pc1) {
    case 0: block A
            pc1 = 1;
            break;
    case 1: block C
    }

    switch (pc2) {
    case 0: block B
            pc2 = 1;
            break;
    case 1: block D
    }
  }
}
```

Multiple Functions

```
void run() {
  for (;;)
    run1(), run2();
}

void run1() {
  static pc1;
  switch (pc1) {
  case 0: block A
          pc1 = 1;
          return;
  case 1: block C
} }

void run2() {
  static pc2;
  switch (pc2) {
  case 0: block B
          pc2 = 1;
          return;
  case 1: block D
} }
```

# Faking Concurrency in C

## One function

```
void run() {
  for (;;) {
    switch (pc1) {
    case 0: block A
            pc1 = 1;
            break;
    case 1: block C
    }

    switch (pc2) {
    case 0: block B
            pc2 = 1;
            break;
    case 1: block D
    }
  }
}
```

## Multiple Functions

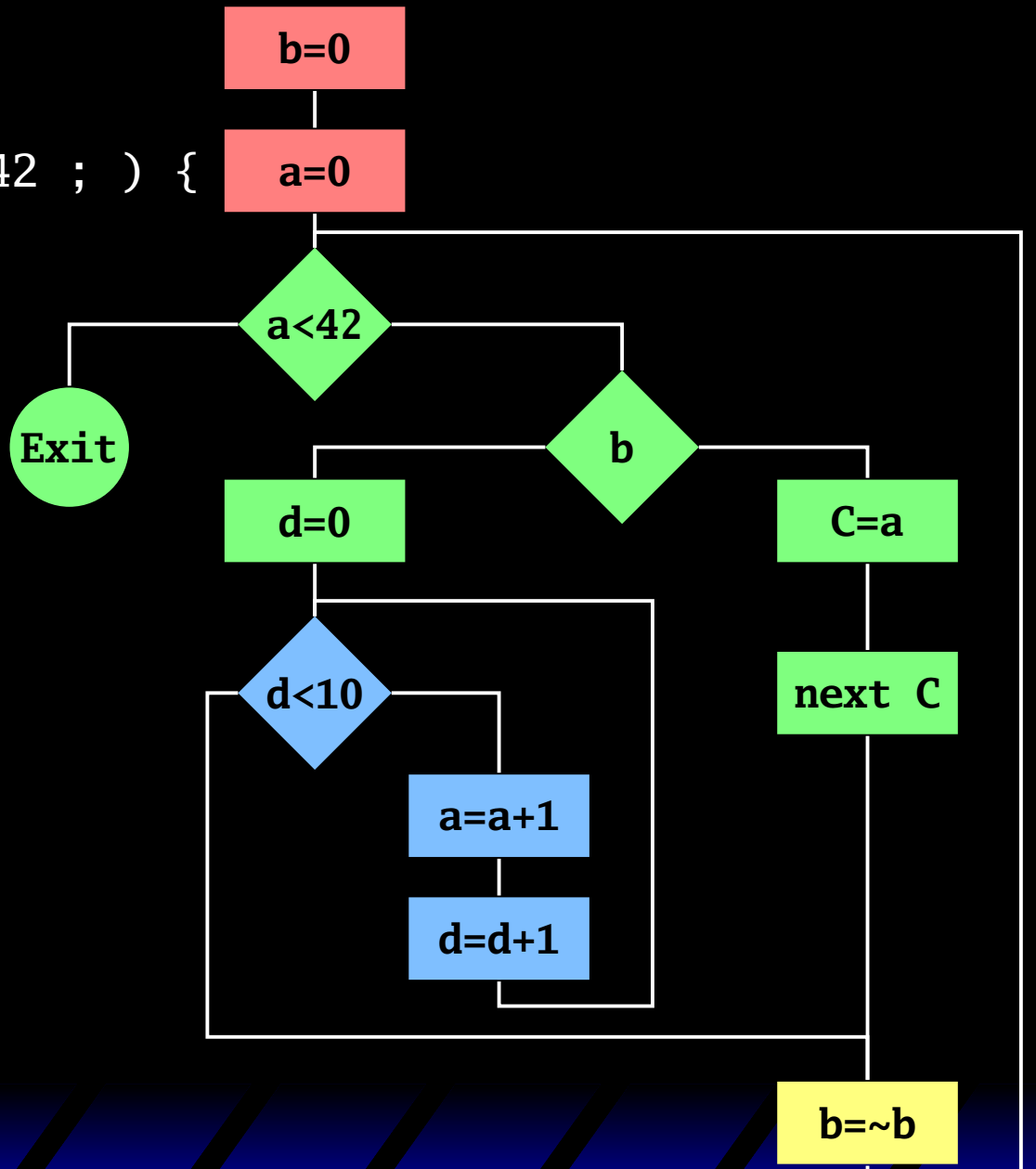
```
void run() {
  for (;;)
    run1(), run2();
}

void run1() {
  static pc1;
  switch (pc1) {
  case 0: block A
          pc1 = 1;
          return;
  case 1: block C
} }

void run2() {
  static pc2;
  switch (pc2) {
  case 0: block B
          pc2 = 1;
          return;
  case 1: block D
} }
```

## Tail Recursion

```
void run1a() {
  block A
  *(sp++) = run2a;
  (*(--sp))(); return;
}

void run1b() {
  block C
  *(sp++) = run2b;
  (*(--sp))(); return;
}

void run2a() {
  block B
  *(sp++) = run1b;
  (*(--sp))(); return;
}

void run2b() {
  block D
  (*(--sp))(); return;
}
```

# Dividing into Fragments

```
void source(int32 &C) {
  bool b = 0;
  for (int32 a = 0 ; a < 42 ; ) {
    if (b) {
      next C = a;
    } else {
      for (int32 d = 0 ;
           d < 10 ; ++d)
        a = a + 1;
    }
    b = ~b;
  }
}
```




Extended basic blocks...

# Global Data

```
void (*stack[3])(void);
void (**sp)(void);

struct channel {
  void (*reader)(void);
  void (*writer)(void);
};

struct channel A_ch = { 0, 0 };
struct channel B_ch = { 0, 0 };
struct channel C_ch = { 0, 0 };
unsigned char A, B, C;

struct {
  char blocked;
} source = { 0 };

struct {
  char blocked;
  unsigned char tmp;
} buf1 = { 0 };

struct {
  char blocked;
  unsigned char tmp;
} buf2 = { 0 };

struct {
  char blocked;
} sink = { 0 };
```

- Stack of pointers to runnable functions
- Each channel holds pointer to function (process) to resume after communication
- *Blocked* flag for each process
- Each process broken into tail-recursive atomic functions

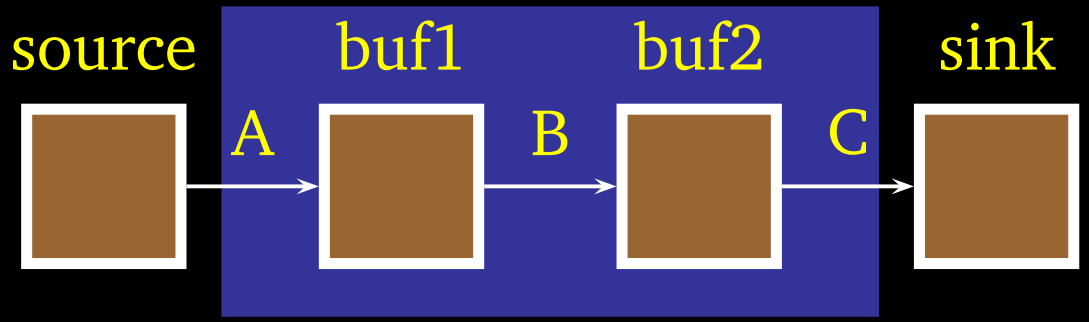
# Source writing to buf1 via A

```
void source_0() {
  A = 17;                                  // Write to channel
  if (buf1.blocked && A_ch.reader) {       // If buffer is blocked reading,
    buf1.blocked = 0;                      // Unblock A
    *(sp++) = A_ch.reader;                 // schedule the reader, and
    A_ch.reader = 0;                       // clear the channel.
  }
  source.blocked = 1;                      // Block us
  A_ch.writer = source_1;                  // to continue below
  (*(--sp))(); return;                     // Run next process
}

void source_1() {                          // Continue here after the write
 /* ... */
}
```

# Buf1 reading from source via A

```
void buf1_0() {
  if (source.blocked && A_ch.writer) { // If source is blocked,
    buf1_1(); return;                  // “goto” buf1_1
  }
  buf1.blocked = 1;                    // Block us
  A_ch.reader = buf1_1;                // to continue below
  (*(--sp))(); return;                 // Run next process
}

void buf1_1() {
  buf1.tmp = A;                        // Read from the channel
  source.blocked = 0;                  // Unblock the source,
  *(sp++) = A_ch.writer;               // schedule it, and
  A_ch.writer = 0;                     // clear the channel.
}
```

# Compiling Processes Together




Build an automaton through abstract simulation

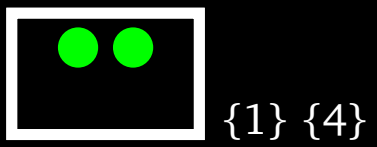
State signature:

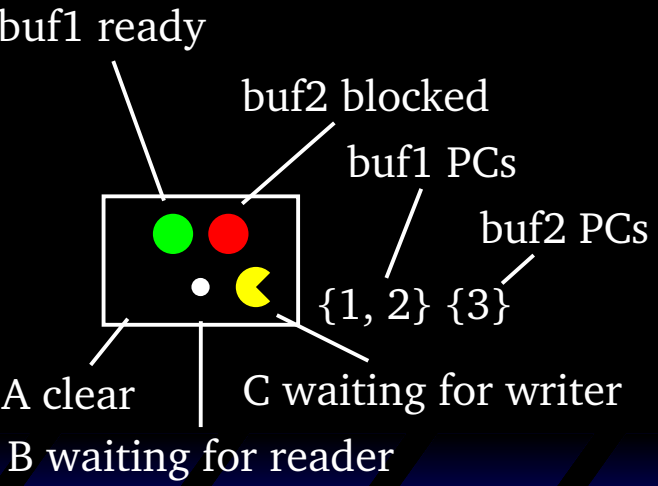
- Running/blocked status of each process
- Blocked on reading/writing status of each channel

*Trick: does not include control or data state of each process*

# Abstract Simulation

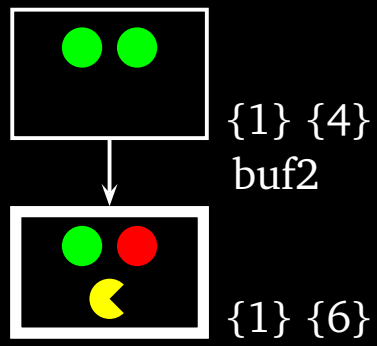
```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```

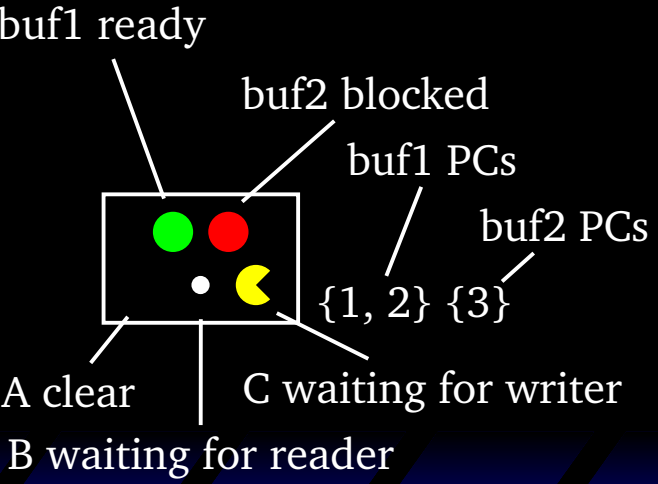


# Abstract Simulation

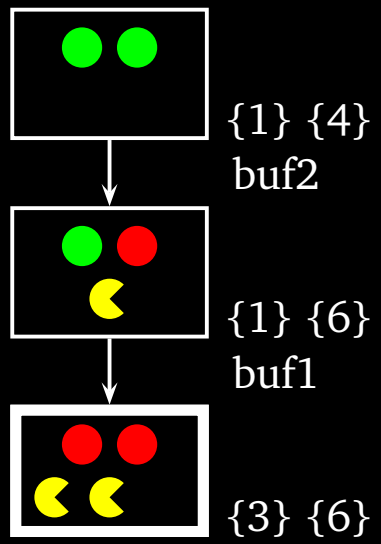
```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```

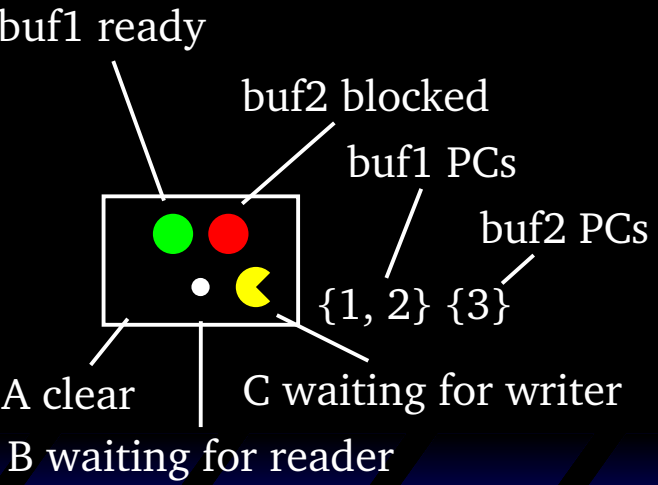


# Abstract Simulation

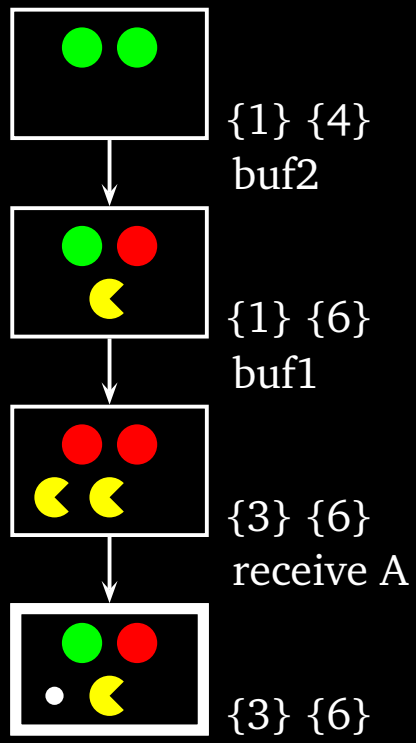
```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```

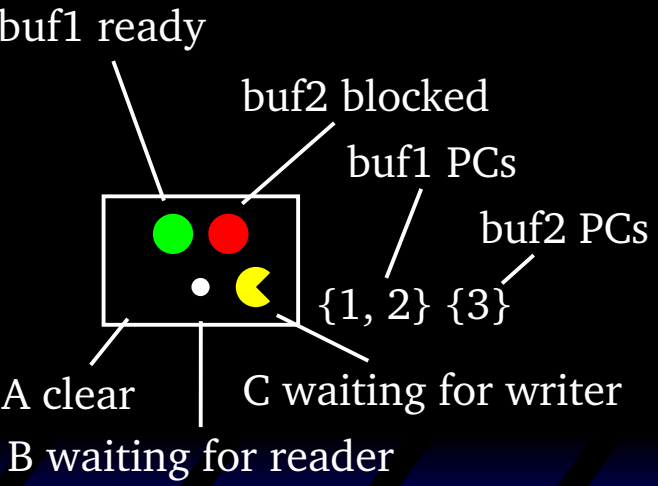


# Abstract Simulation

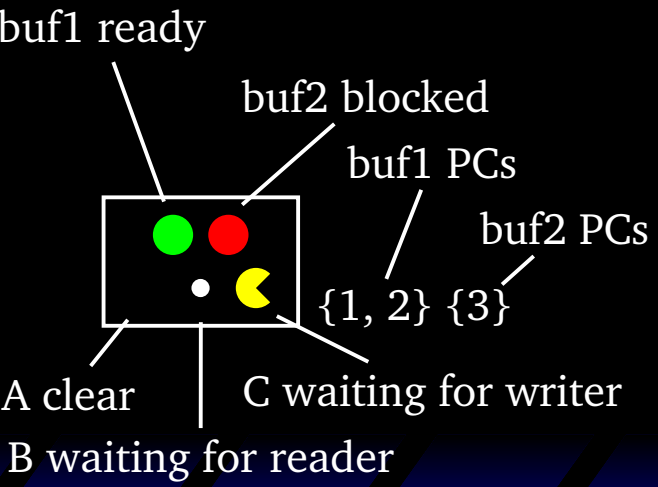
```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```

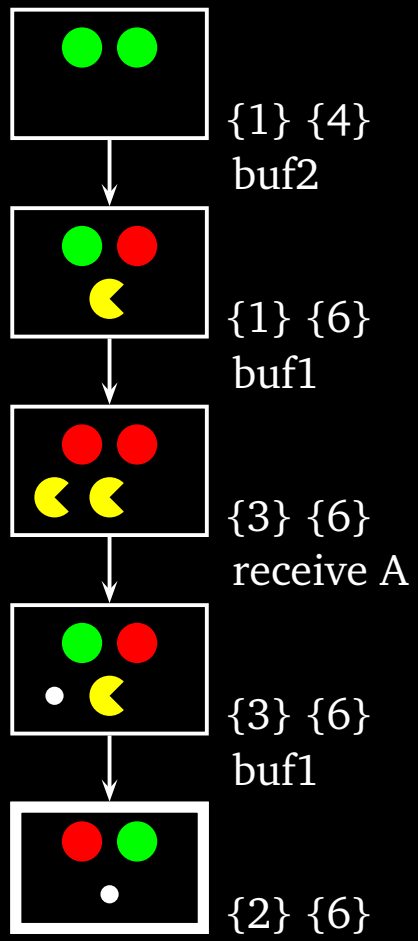


# Abstract Simulation

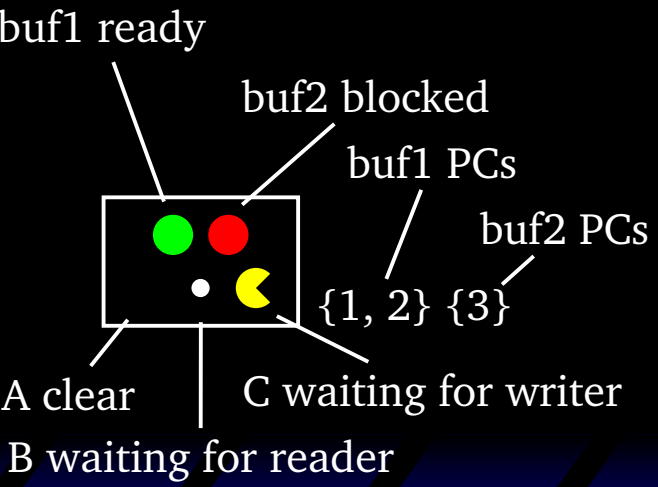
```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```

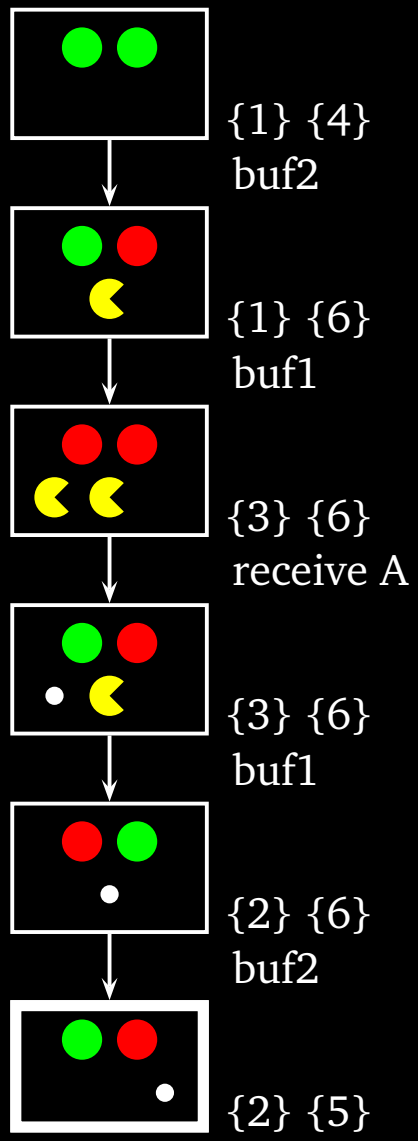


# Abstract Simulation

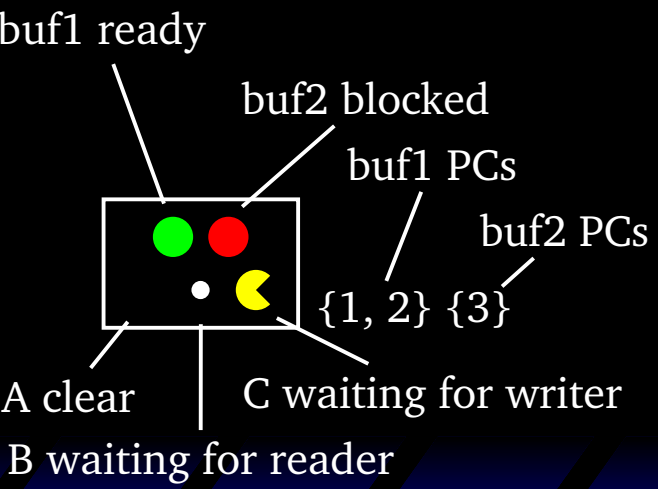
```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```

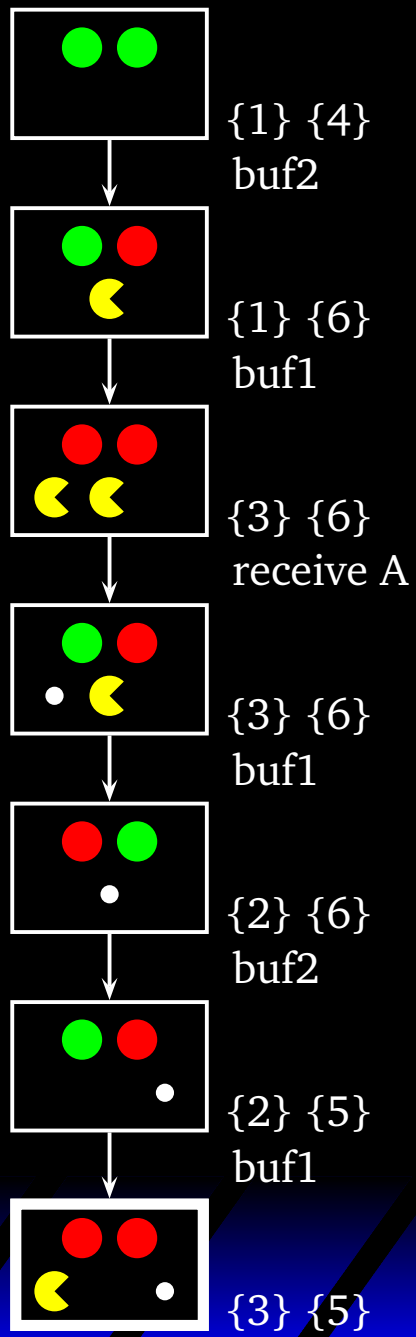


# Abstract Simulation

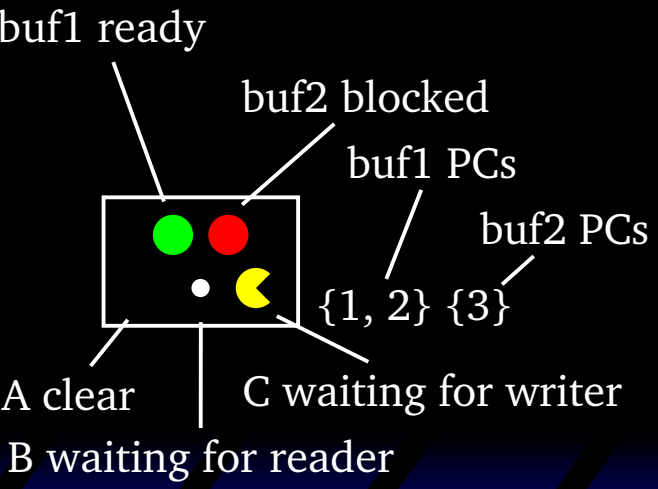
```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```

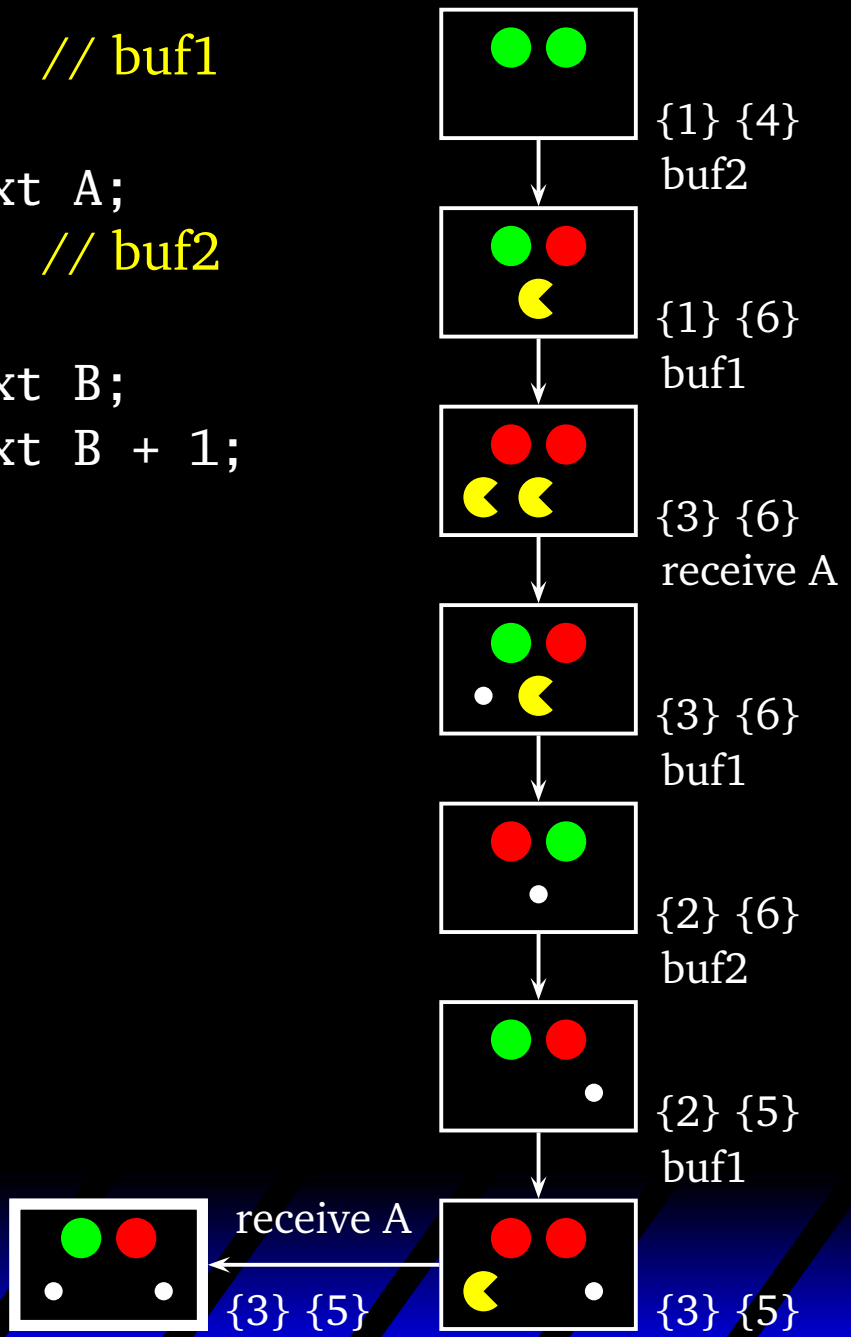


# Abstract Simulation

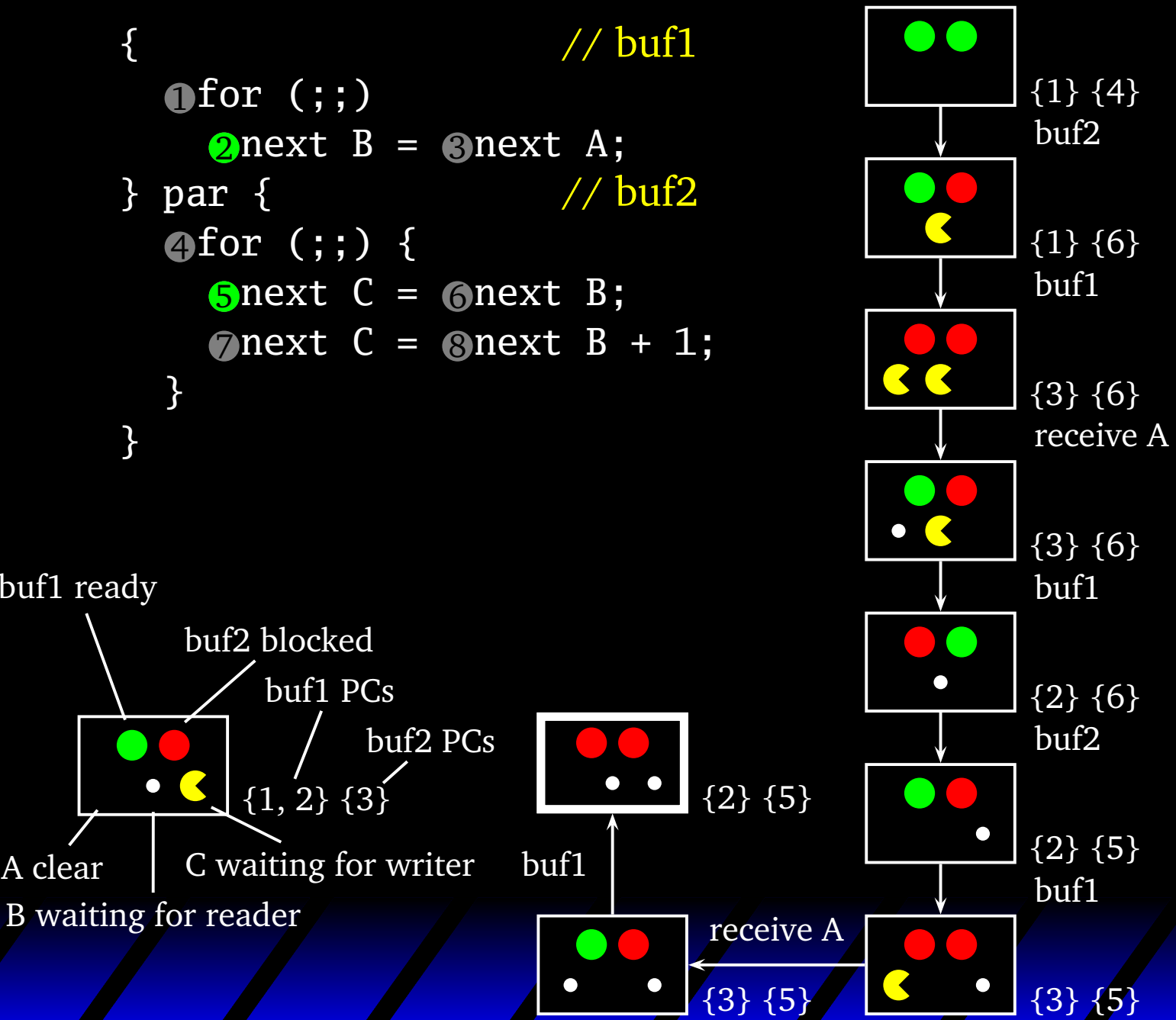
```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



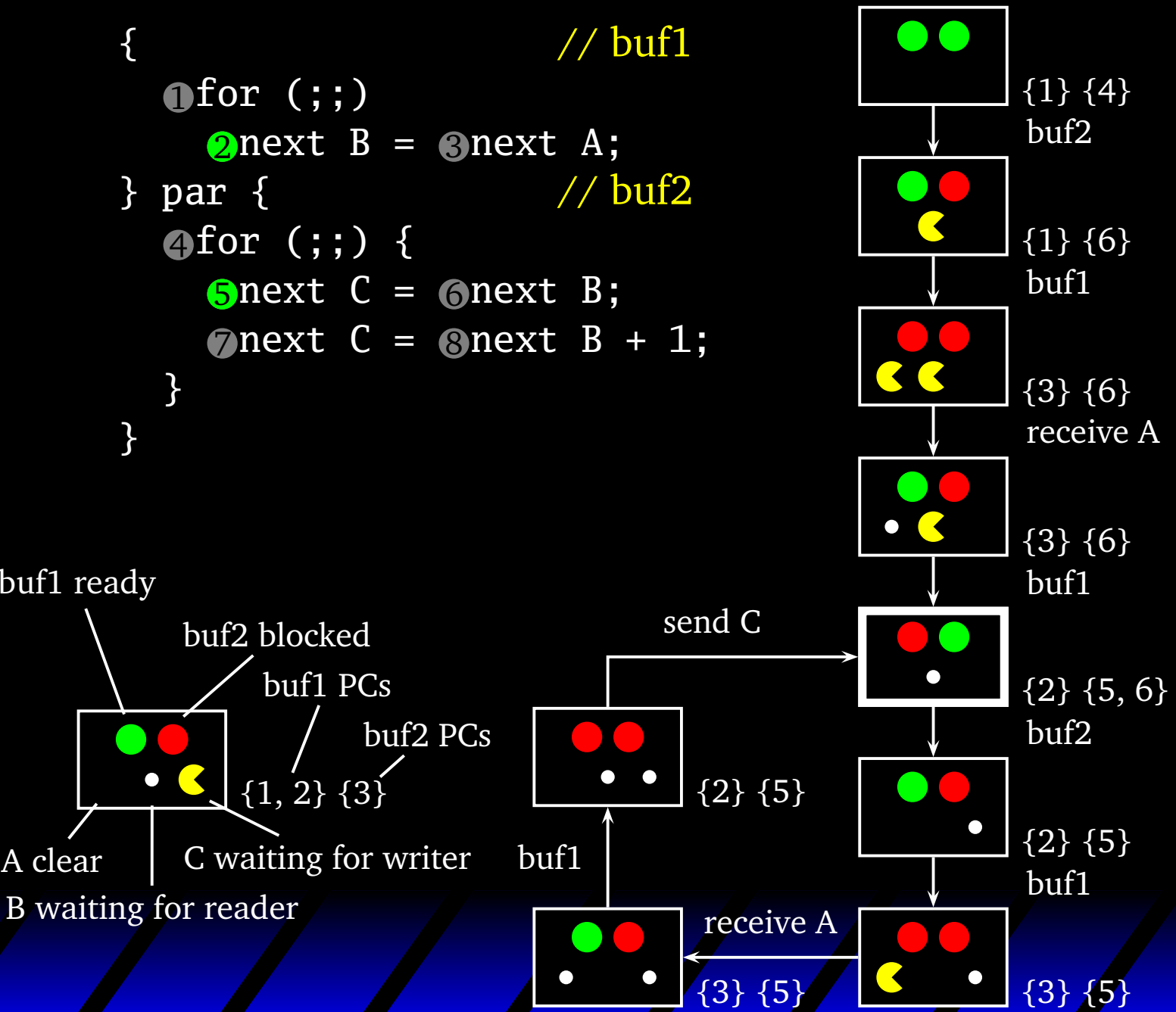
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



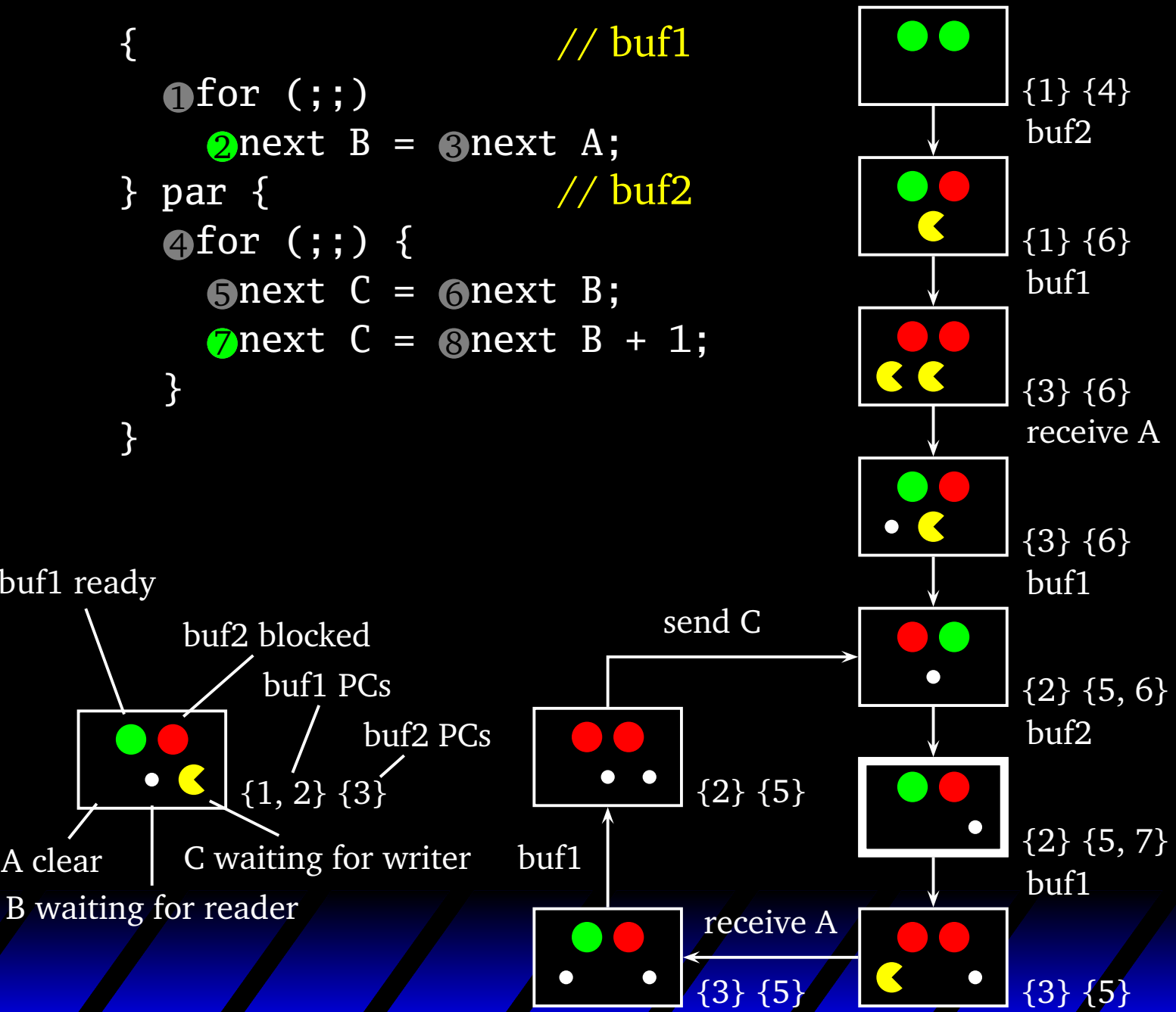
# Abstract Simulation

```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



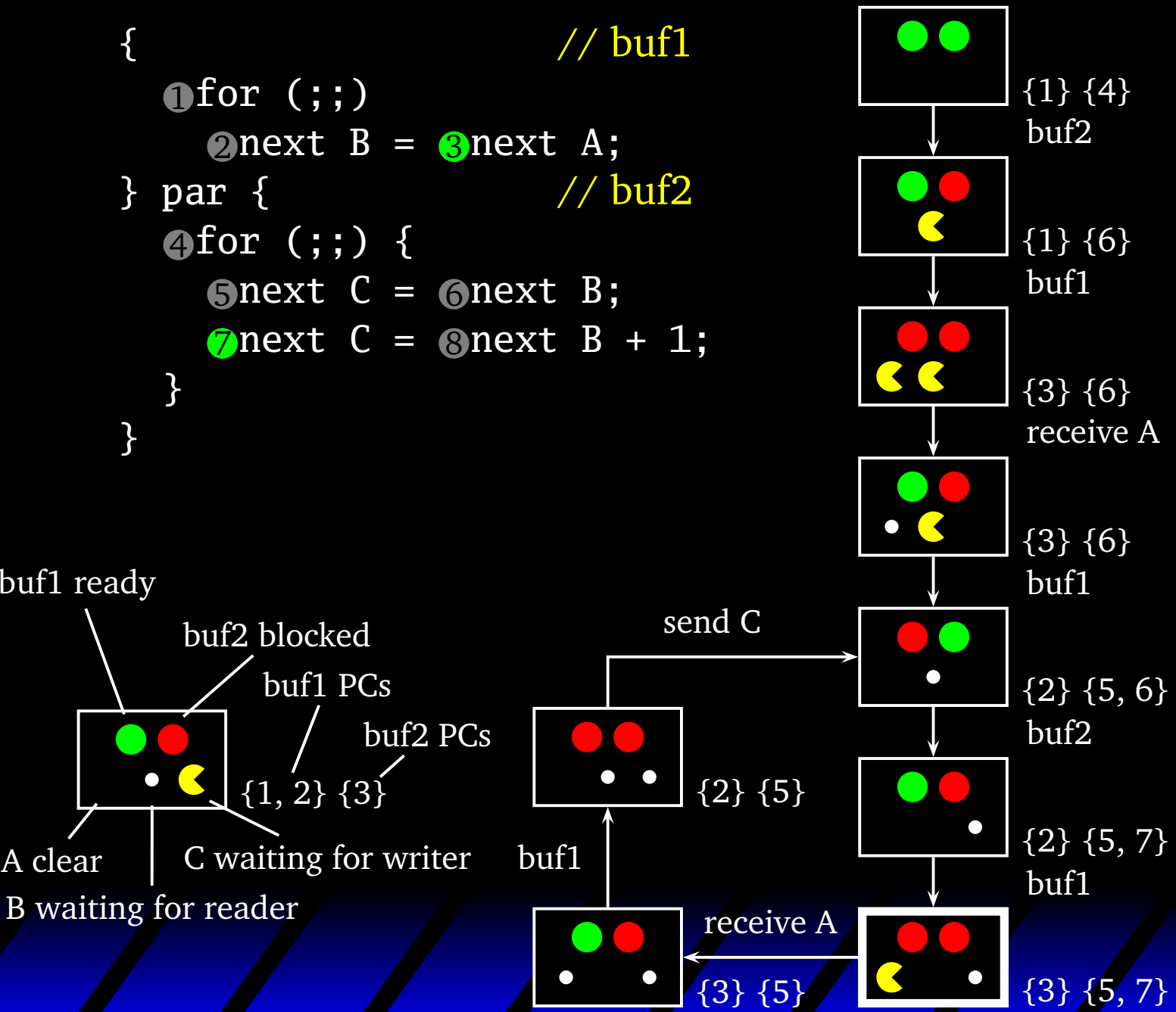
# Abstract Simulation

```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



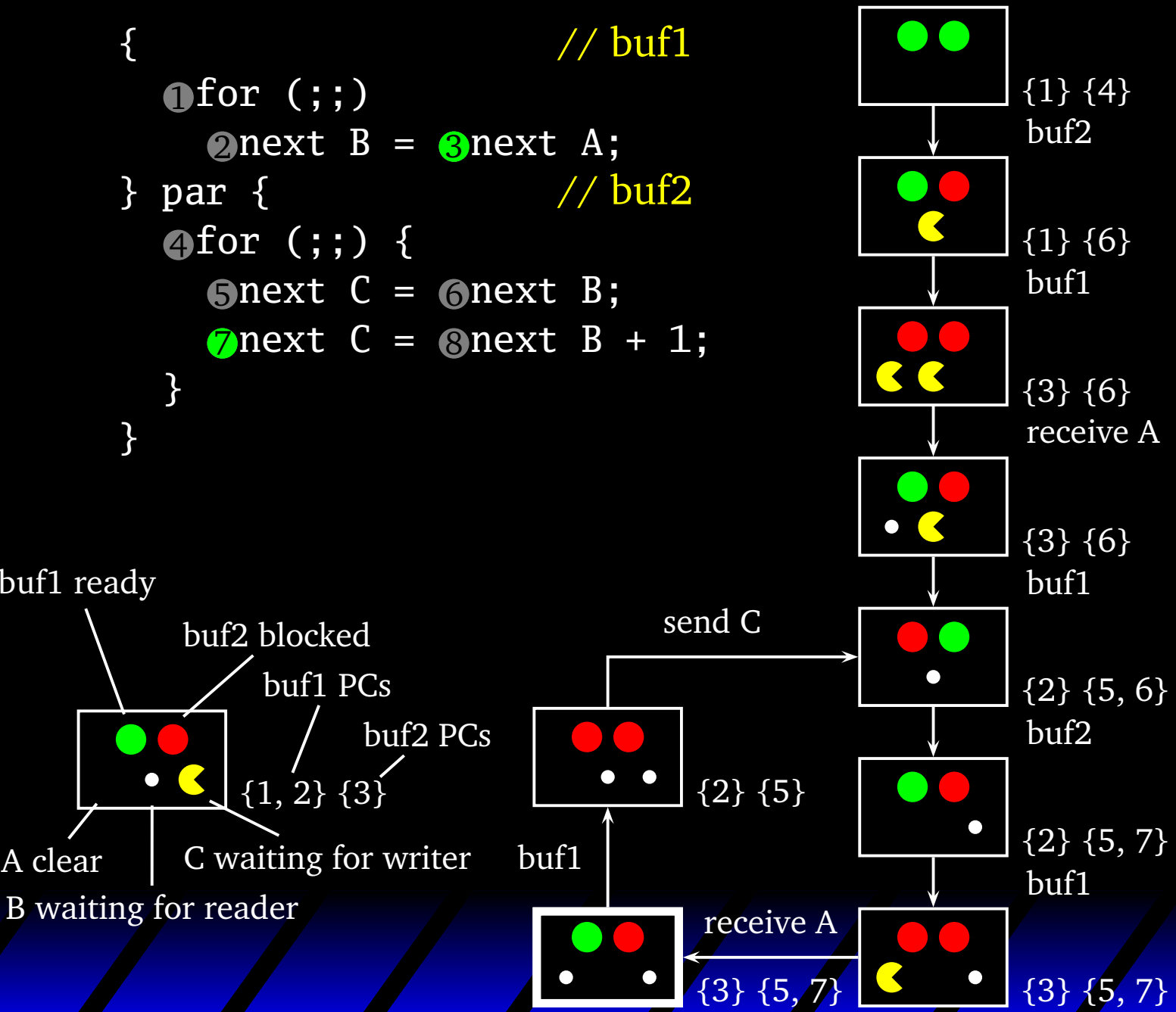
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



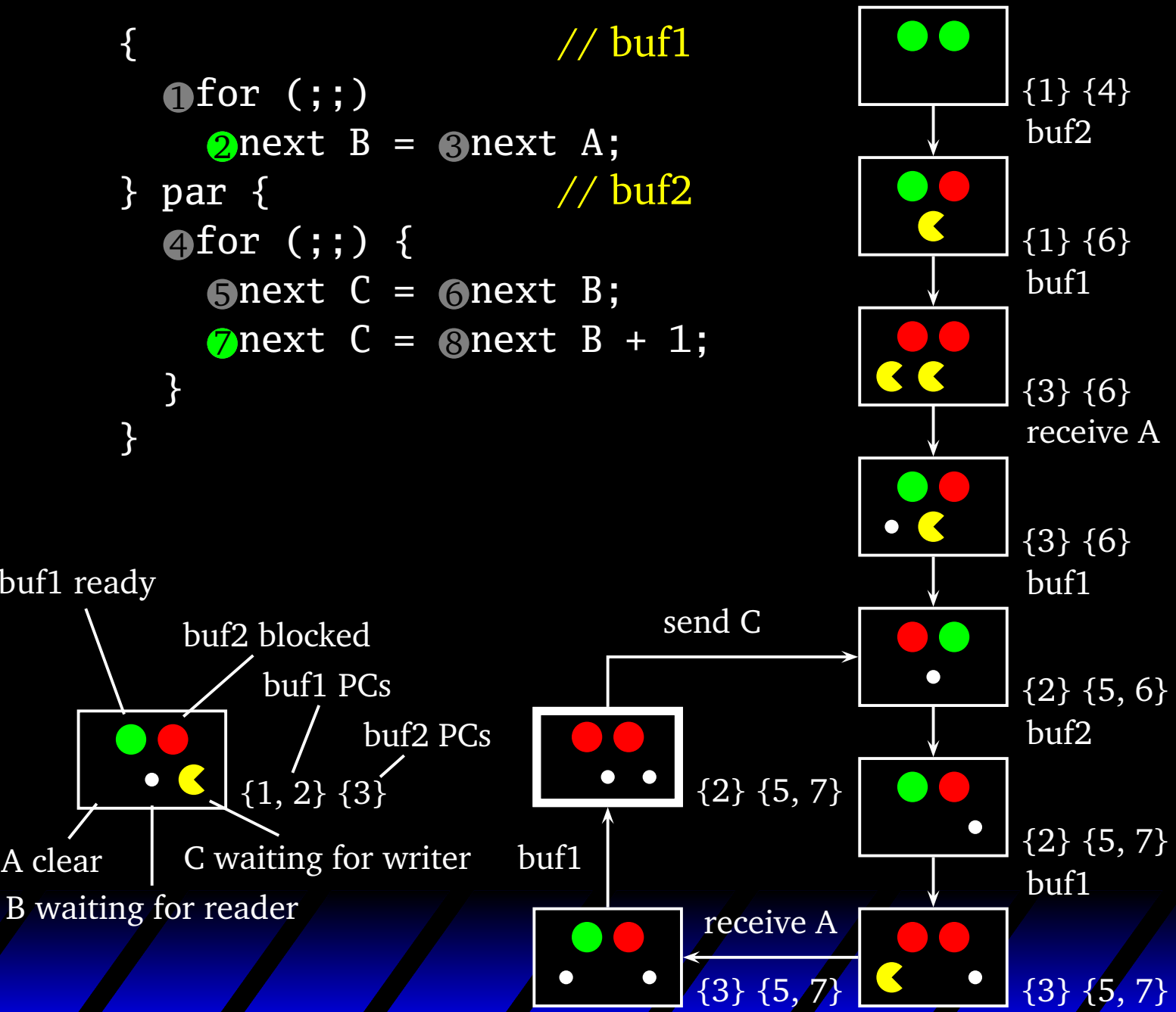
# Abstract Simulation

```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



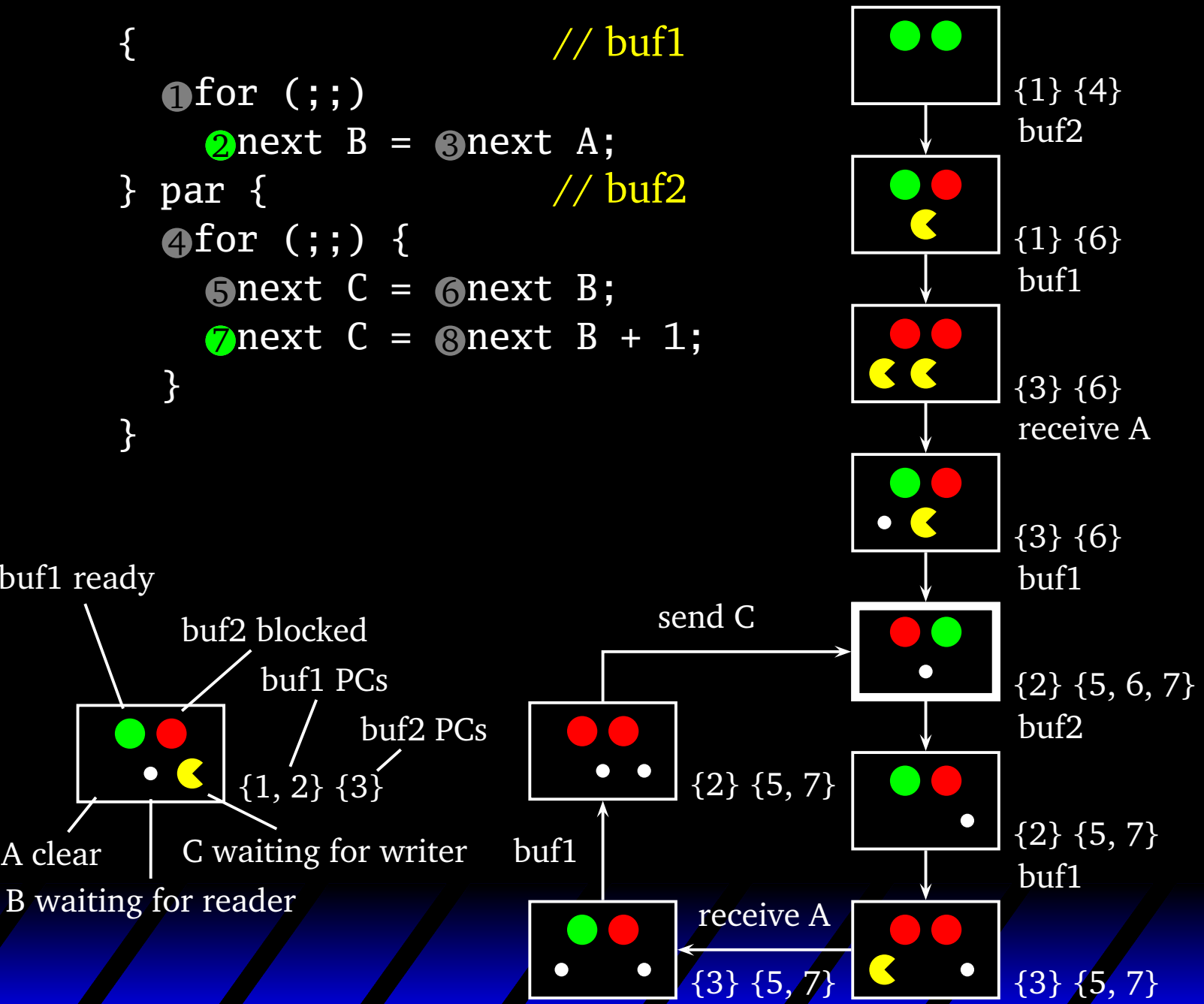
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



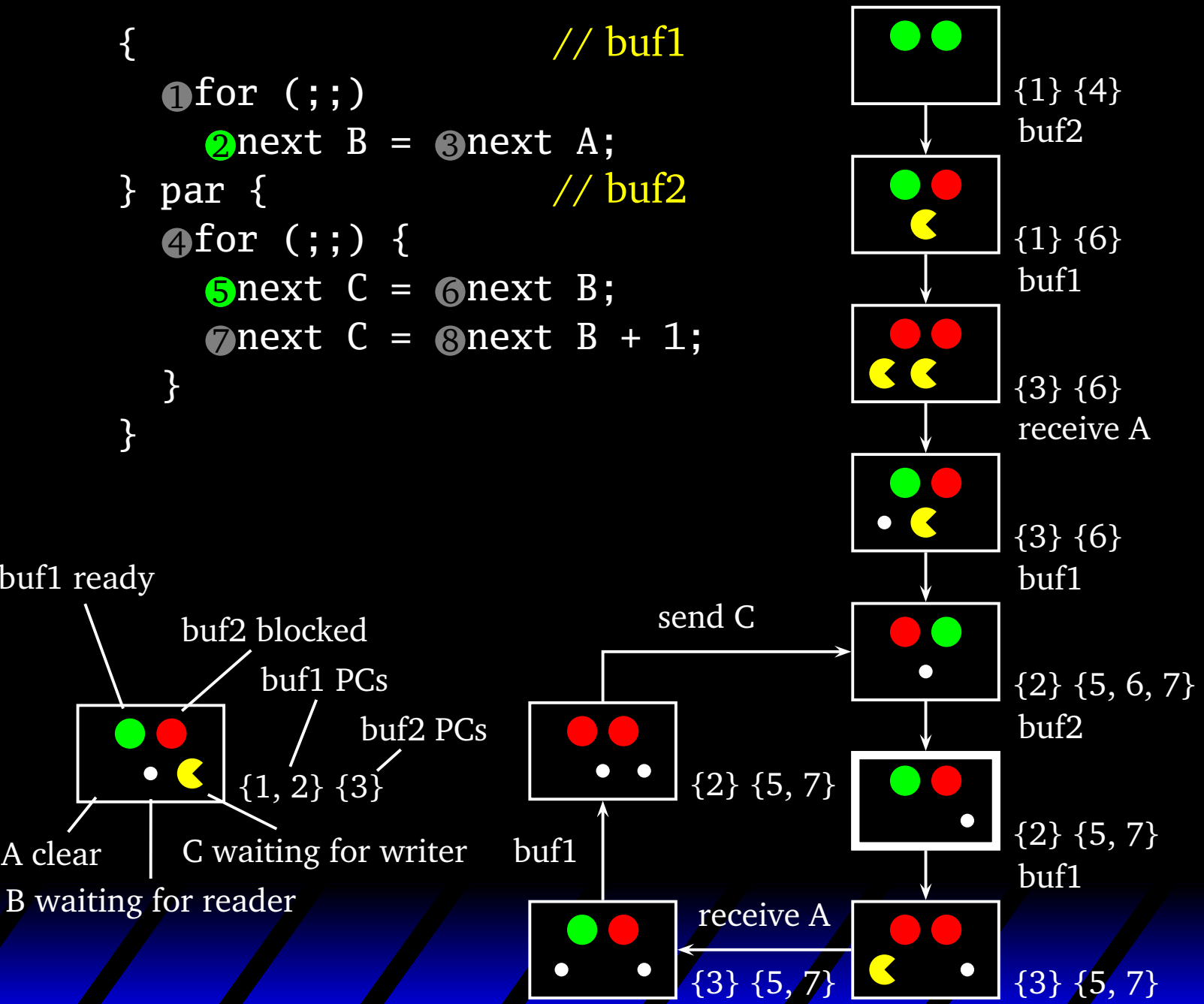
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



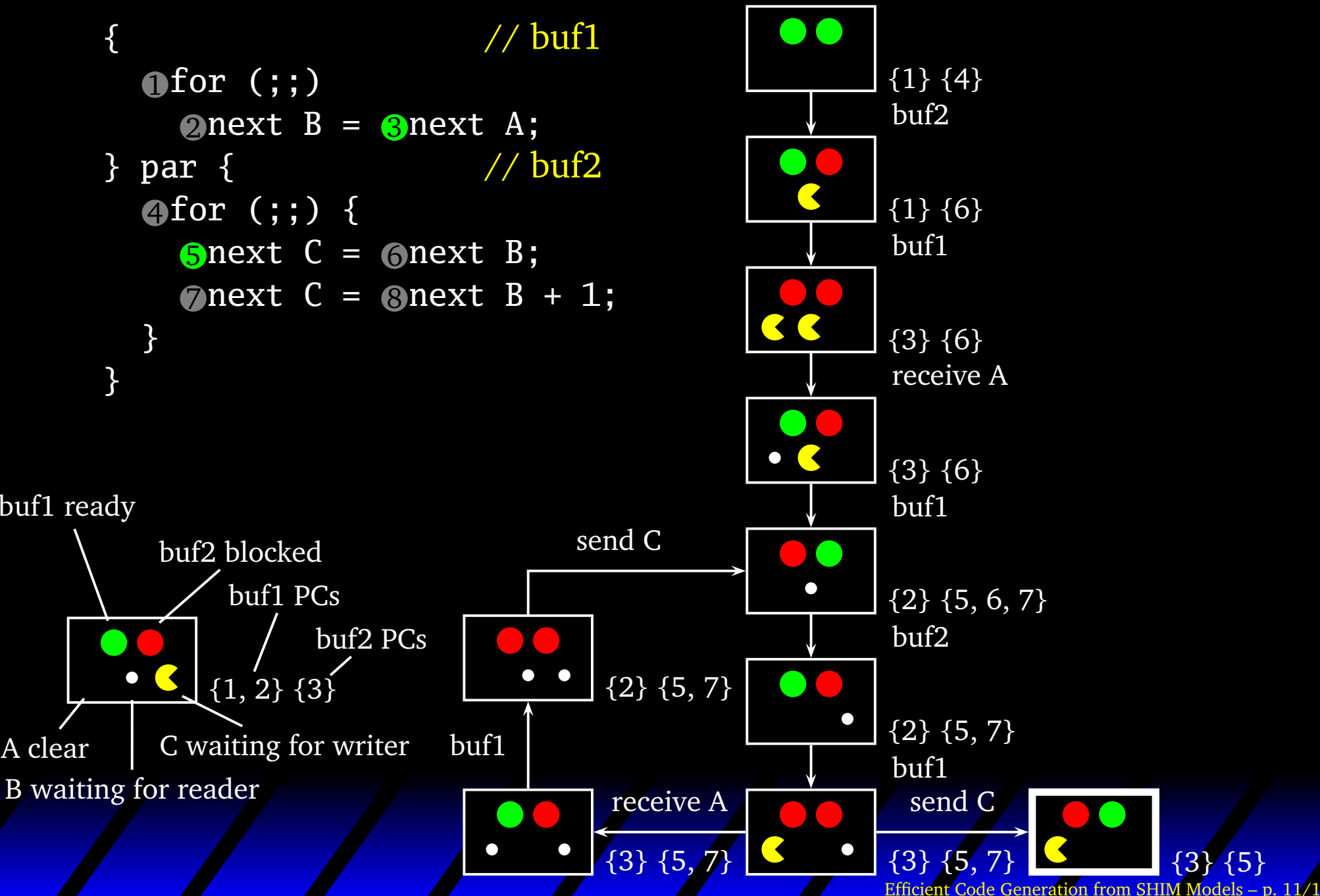
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



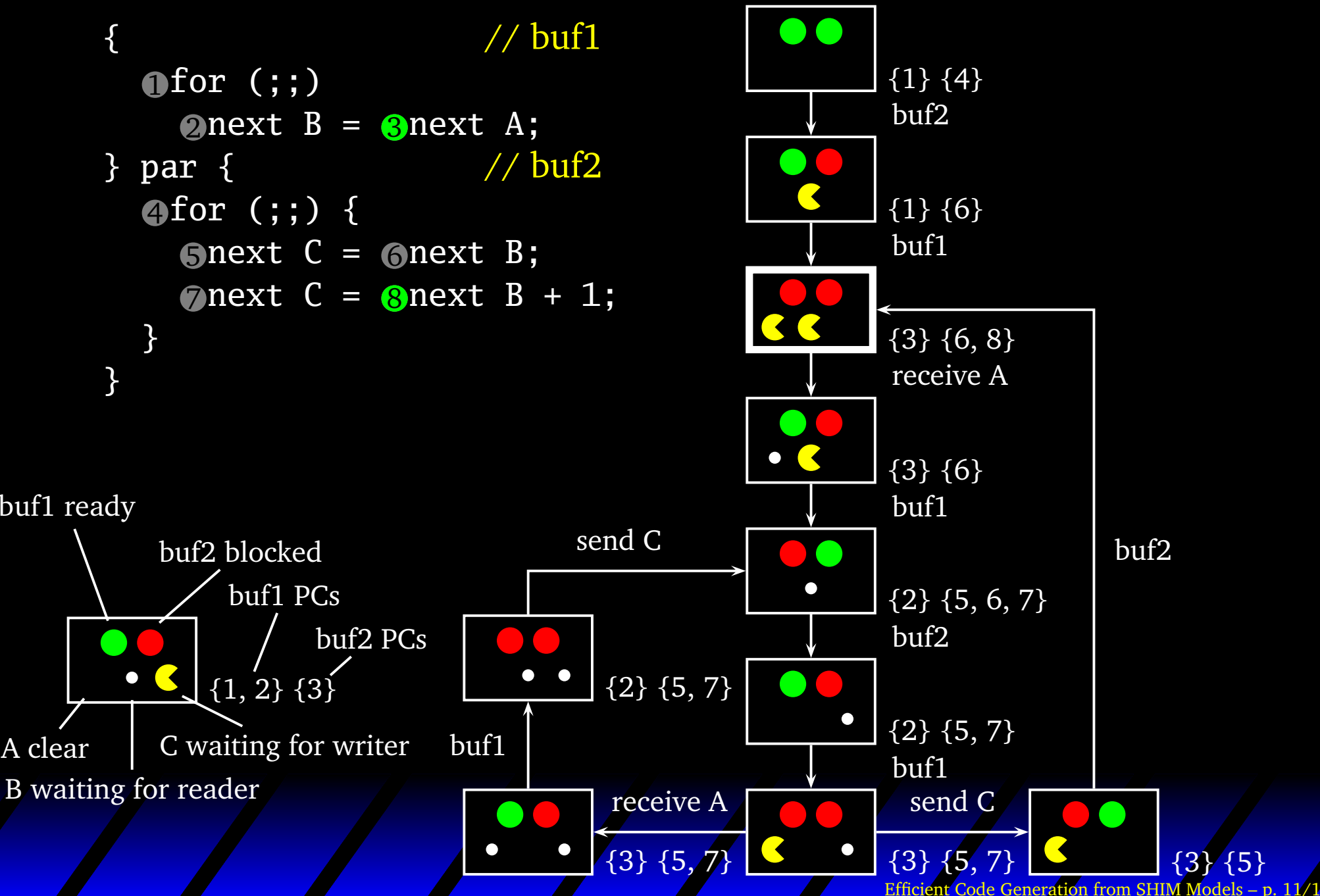
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



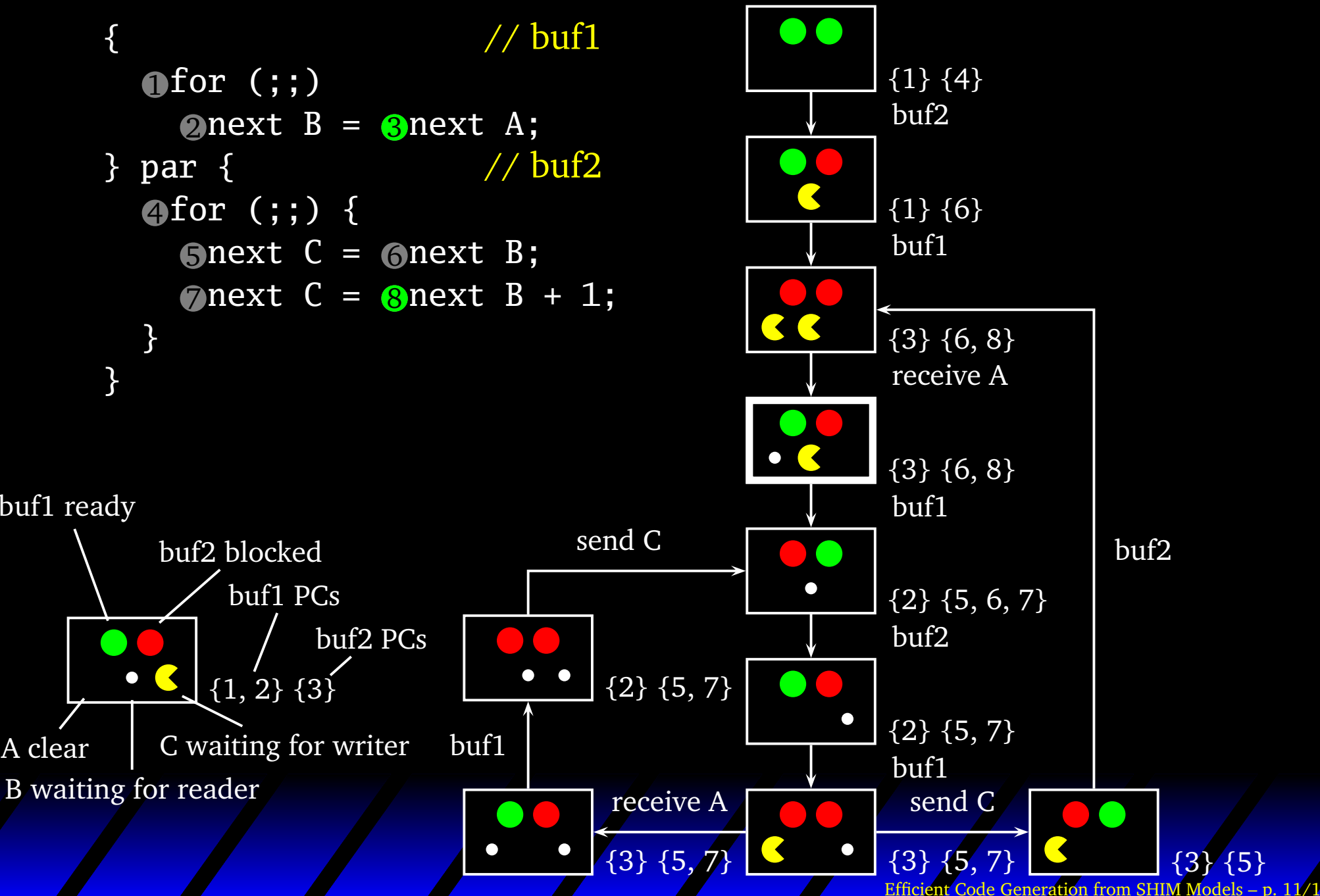
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



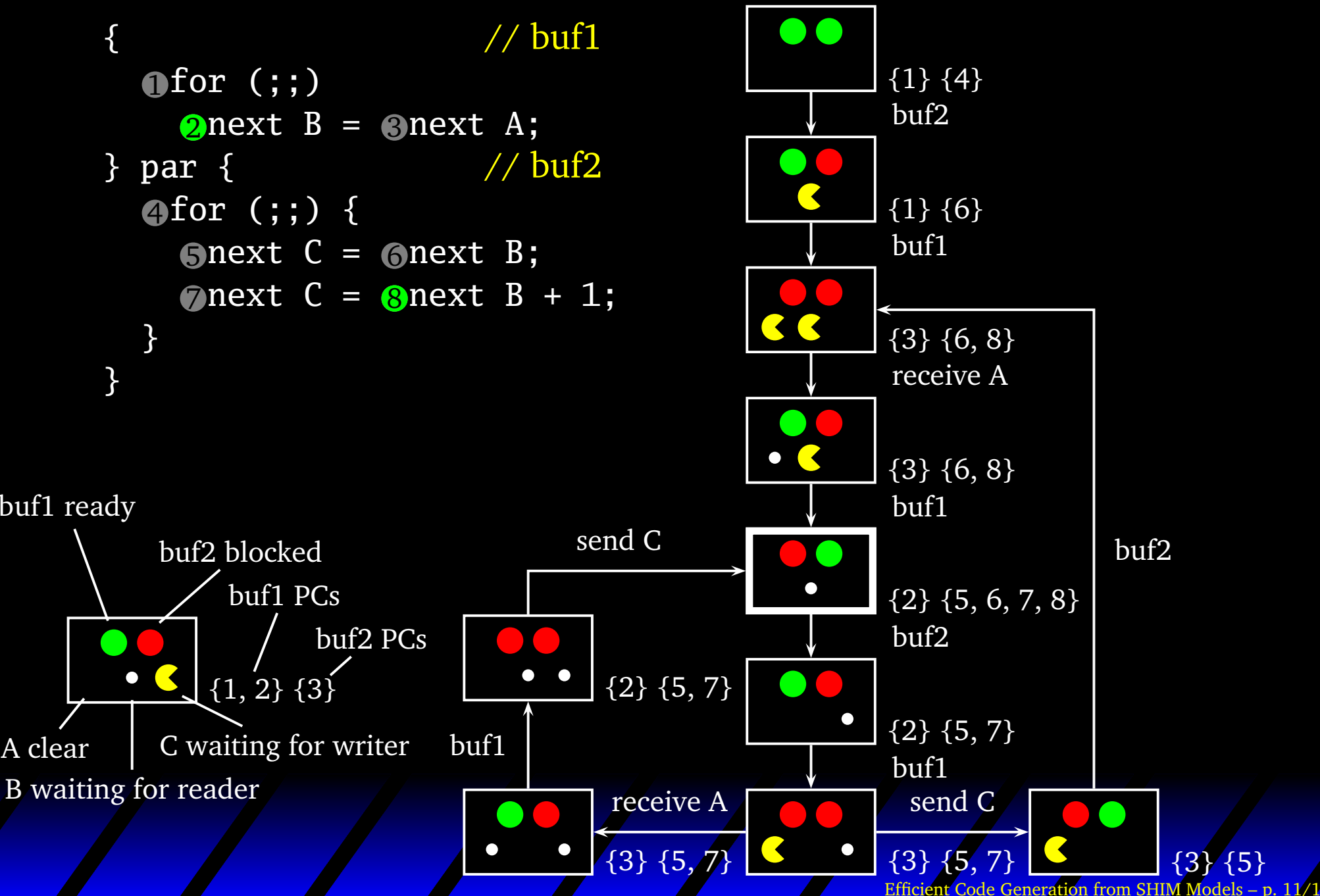
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



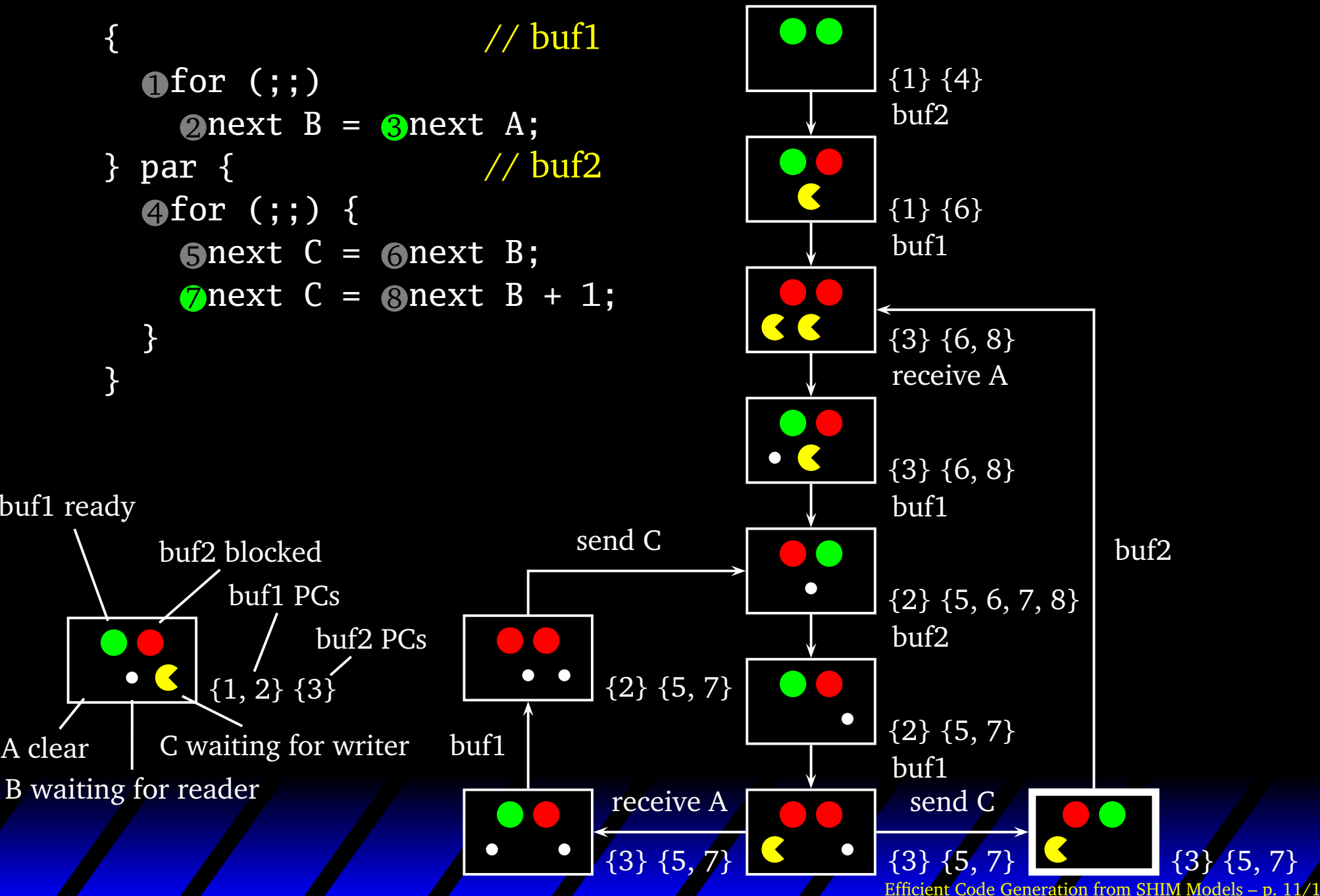
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



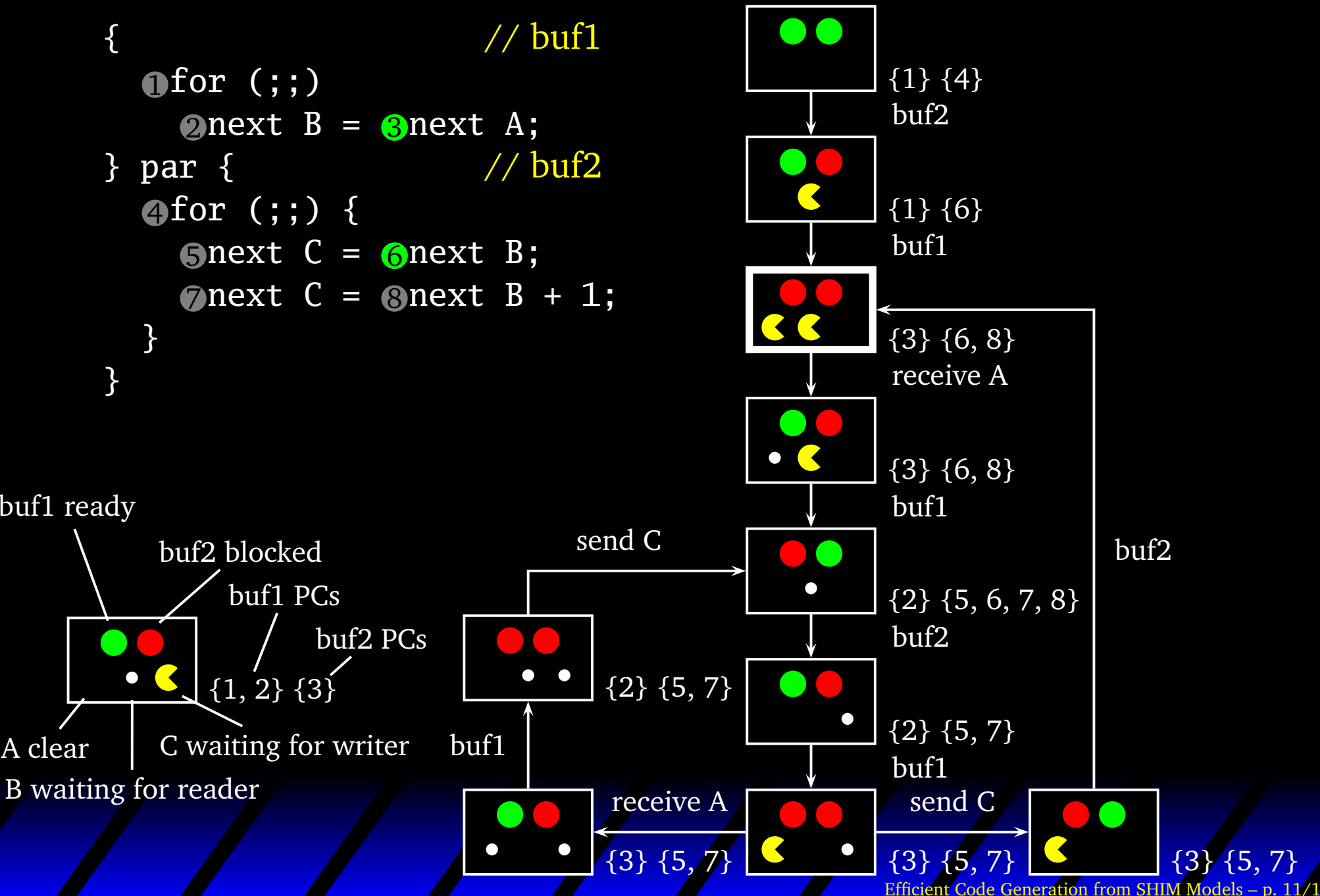
# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



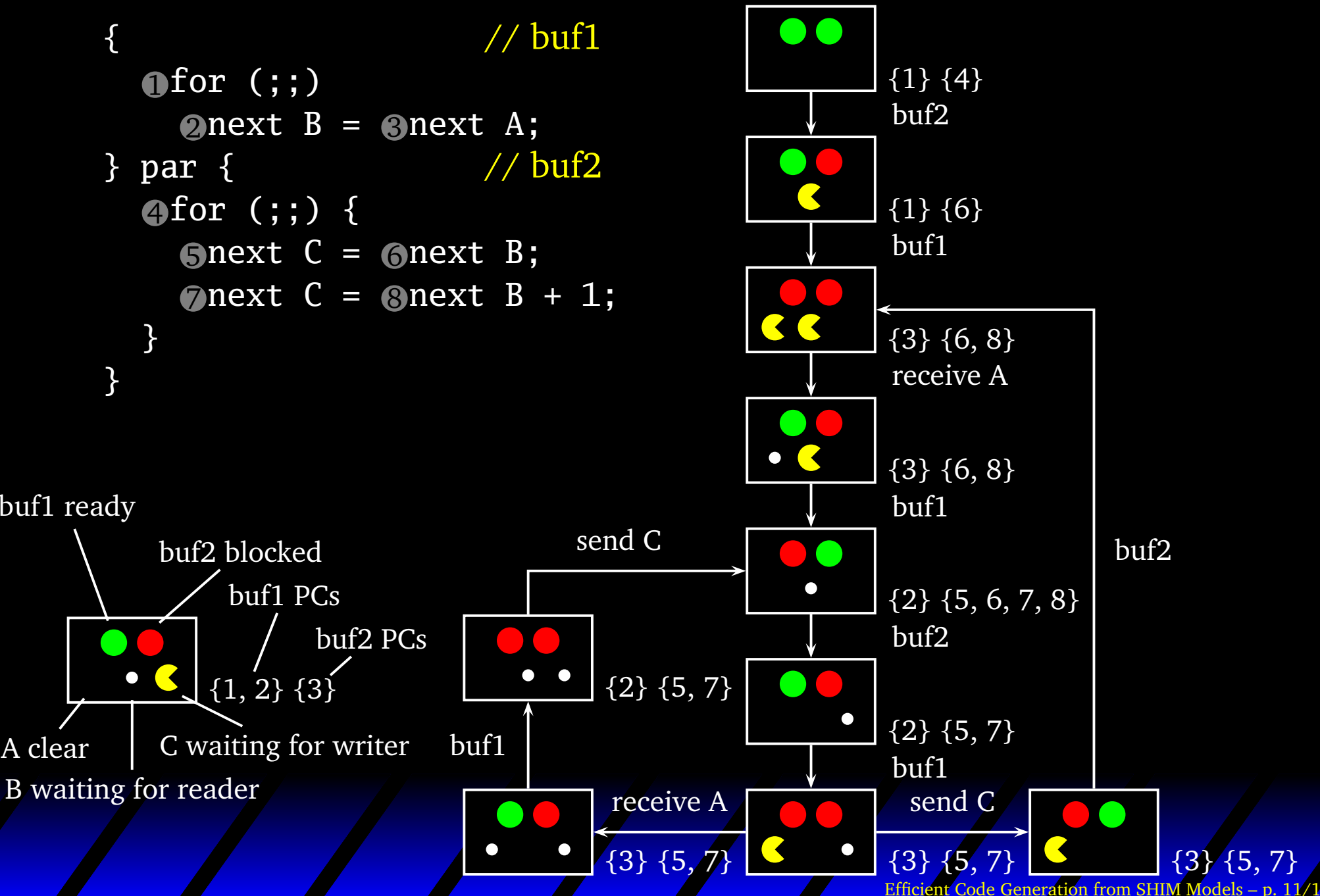
# Abstract Simulation

```
{                                  // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                            // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



# Abstract Simulation

```
{                              // buf1
  ❶for (;;)
    ❷next B = ❸next A;
} par {                        // buf2
  ❹for (;;) {
    ❺next C = ❻next B;
    ❼next C = ❽next B + 1;
  }
}
```



# Benchmarks

| Example | Lines | Processes |
|---|---|---|
| Berkeley | 36 | 3 |
| Buffer2 | 25 | 4 |
| Buffer3 | 26 | 5 |
| Buffer10 | 33 | 12 |
| Esterel1 | 144 | 5 |
| Esterel2 | 127 | 5 |
| FIR5 | 78 | 19 |
| FIR19 | 190 | 75 |

# Executable Sizes

| Example | Switch | Tail- | Static (partial) | | Static (full) | |
|---|---|---|---|---|---|---|
| | | Recursive | size | states | size | states |
| Berkeley | 860 | 1299 | 1033 | 5 | 551 | 6 |
| Buffer2 | 832 | 1345 | 1407 | 10 | 403 | 8 |
| Buffer3 | 996 | 1579 | 1771 | 20 | 443 | 10 |
| Buffer10 | 2128 | 3249 | 5823 | 174 | 687 | 24 |
| Esterel1 | 3640 | 5971 | 8371 | 49 | 5611 | 56 |
| Esterel2 | 4620 | 7303 | 6871 | 24 | 2539 | 18 |
| FIR5 | 4420 | 6863 | 6819 | 229 | 1663 | 79 |
| FIR19 | 17052 | 25967 | 67823 | 2819 | 7287 | 372 |

# Speedups vs. Switch

| Example | Tail-Recursive | Static (partial) | Static (full) |
|---|---|---|---|
| Berkeley | 2.9× | 2.6 | 7.8 |
| Buffer2 | 2.0 | 2.4 | 11 |
| Buffer3 | 2.1 | 2.6 | 10 |
| Buffer10 | 1.7 | 4.8 | 12 |
| Esterel1 | 1.9 | 2.9 | 5.9 |
| Esterel2 | 2.0 | 2.5 | 5.2 |
| FIR5 | 0.92 | 4.8 | 7 |
| FIR19 | 0.90 | 5.9 | 7.1 |

# Conclusions

- The SHIM Model: Sequential processes communicating through rendezvous
- Tail-recursion: postponed *goto* in C
- Dynamic code maintains stack of function pointers to runnable processes
- Each process broken into one function per extended basic block
- Compiling processes together through abstract simulation
- Main trick: do not consider PC values part of state

# Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation

# Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core

# Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core
- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics

# Future Work

- Automata abstract communication patterns
  Useful for deadlock detection, protocol violation
- Synthesis for multicore processors
  Compile together the processes on each core
- Hardware/software cosynthesis
  Bounded subset has reasonable hardware semantics
- Convince world: deterministic concurrency is good