Embedded Systems Programming

Lecture 4

Verónica Gaspes
www2.hh.se/staff/vero

Center for Research on Embedded Systems
School of Information Science, Computer and Electrical Engineering

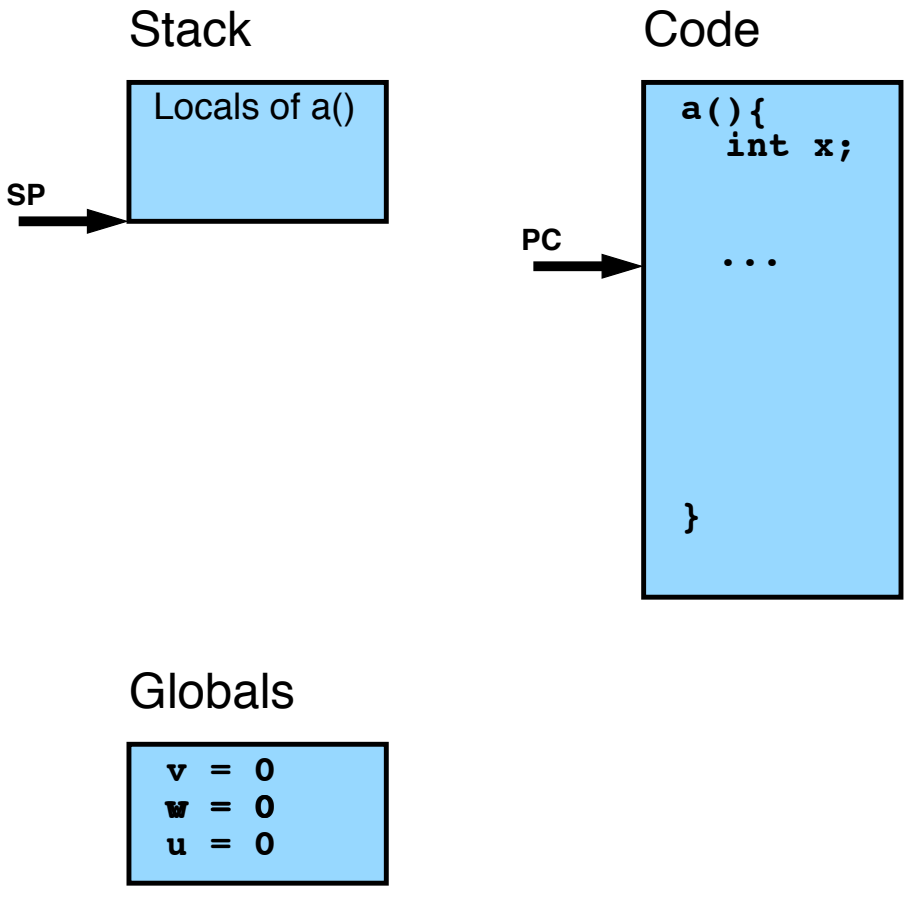
Featuring slides by Johan Nordlander

Execution of a C program

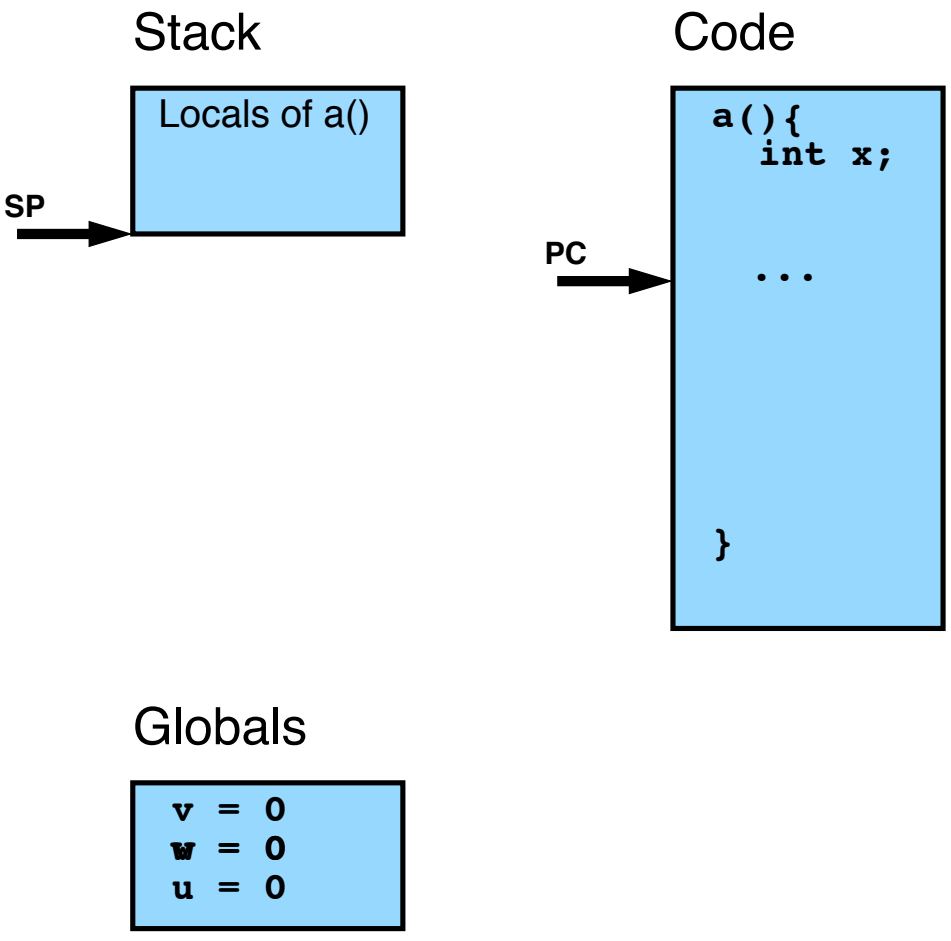
The following slides will help us understand what we will need to do to implement threads and automatic interleaving.

Execution of a C program

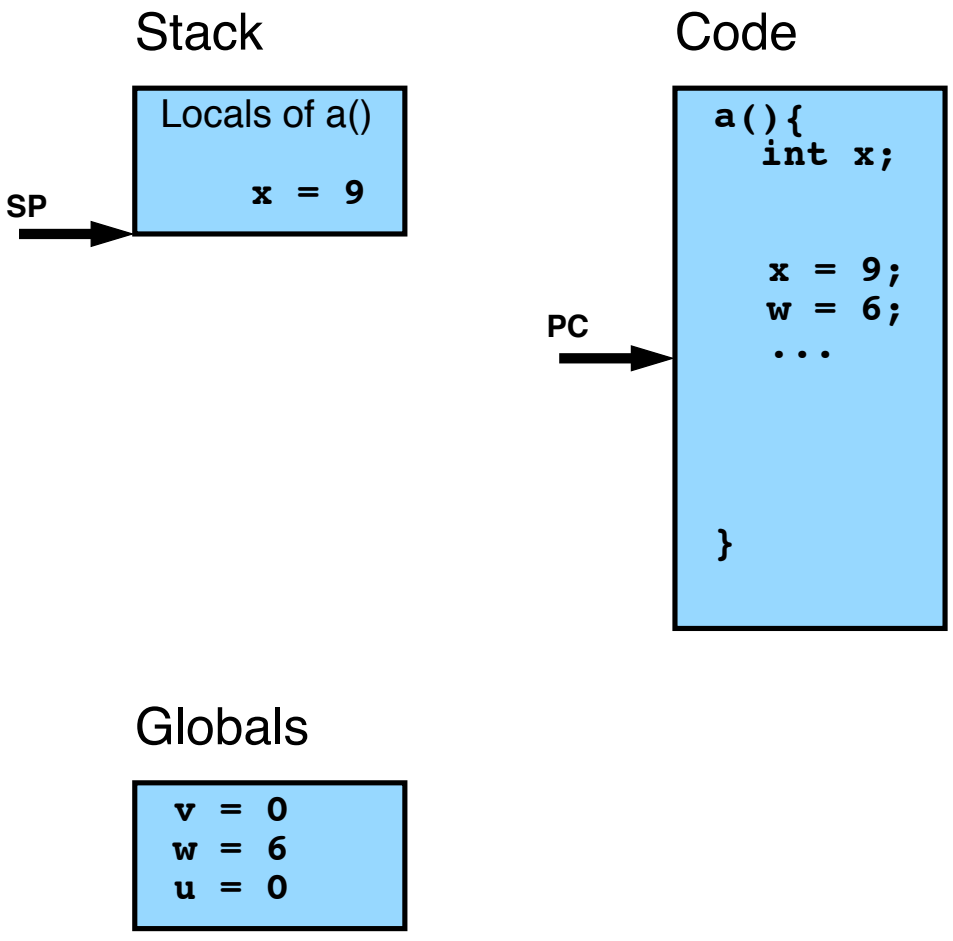
The following slides will help us understand what we will need to do to implement threads and automatic interleaving.



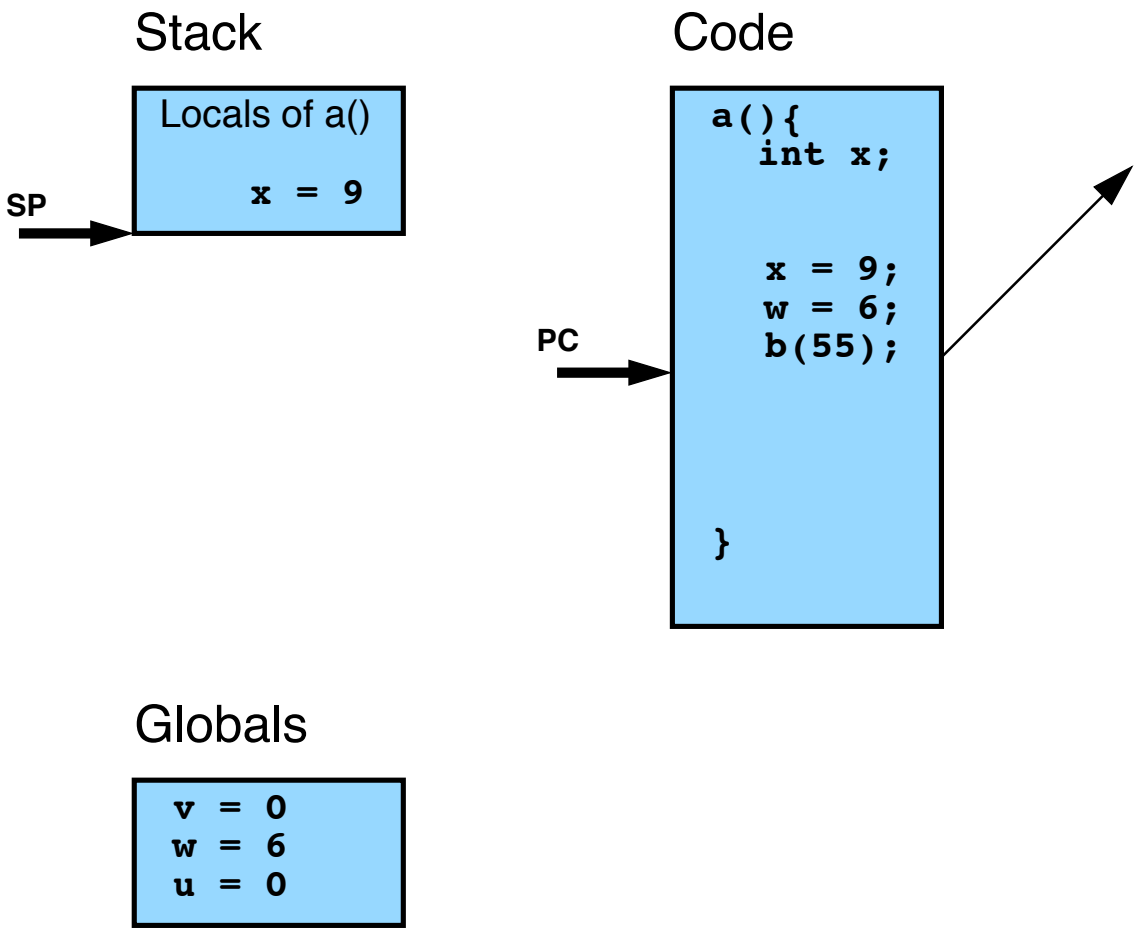
Execution of a C program



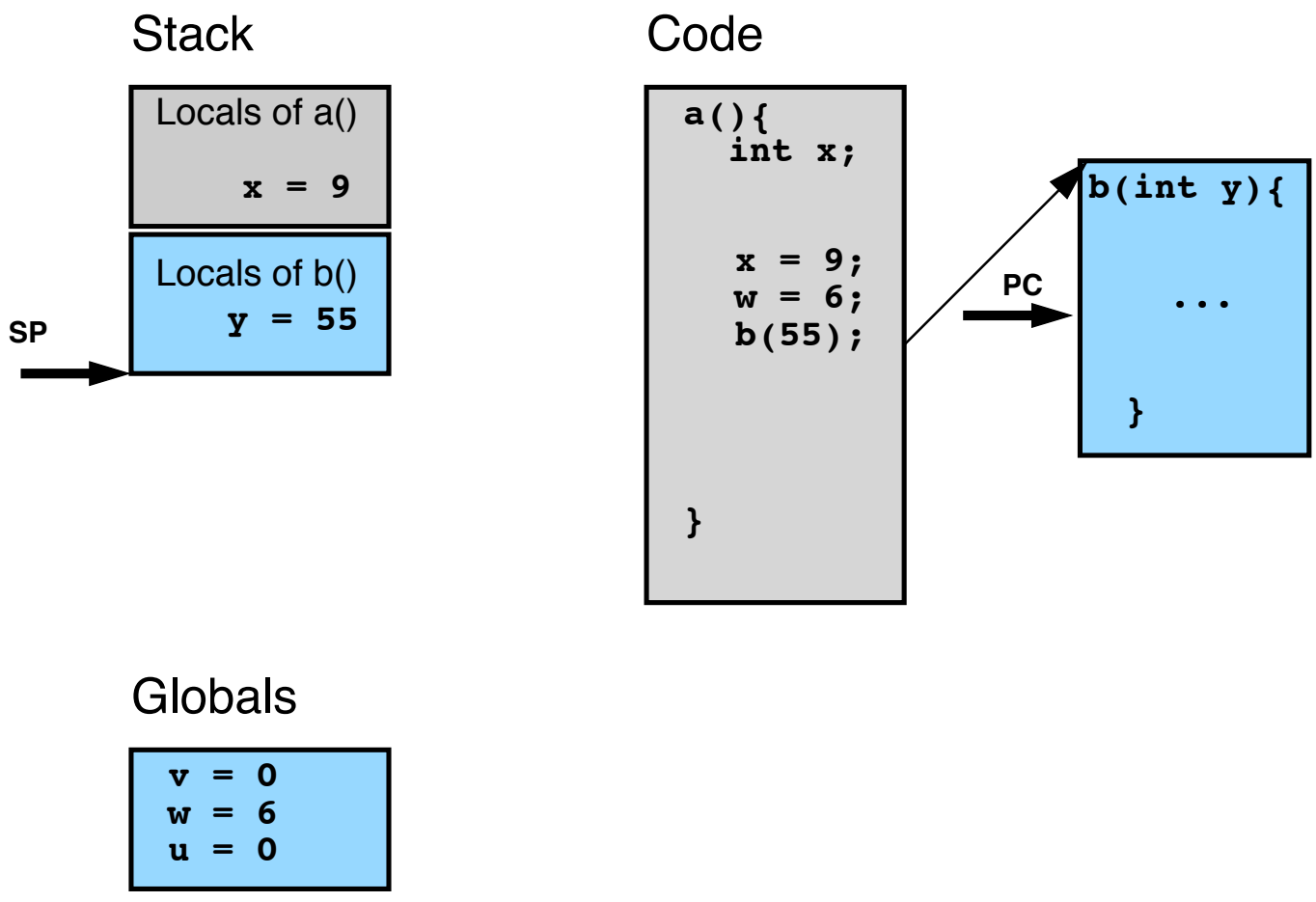
Execution of a C program



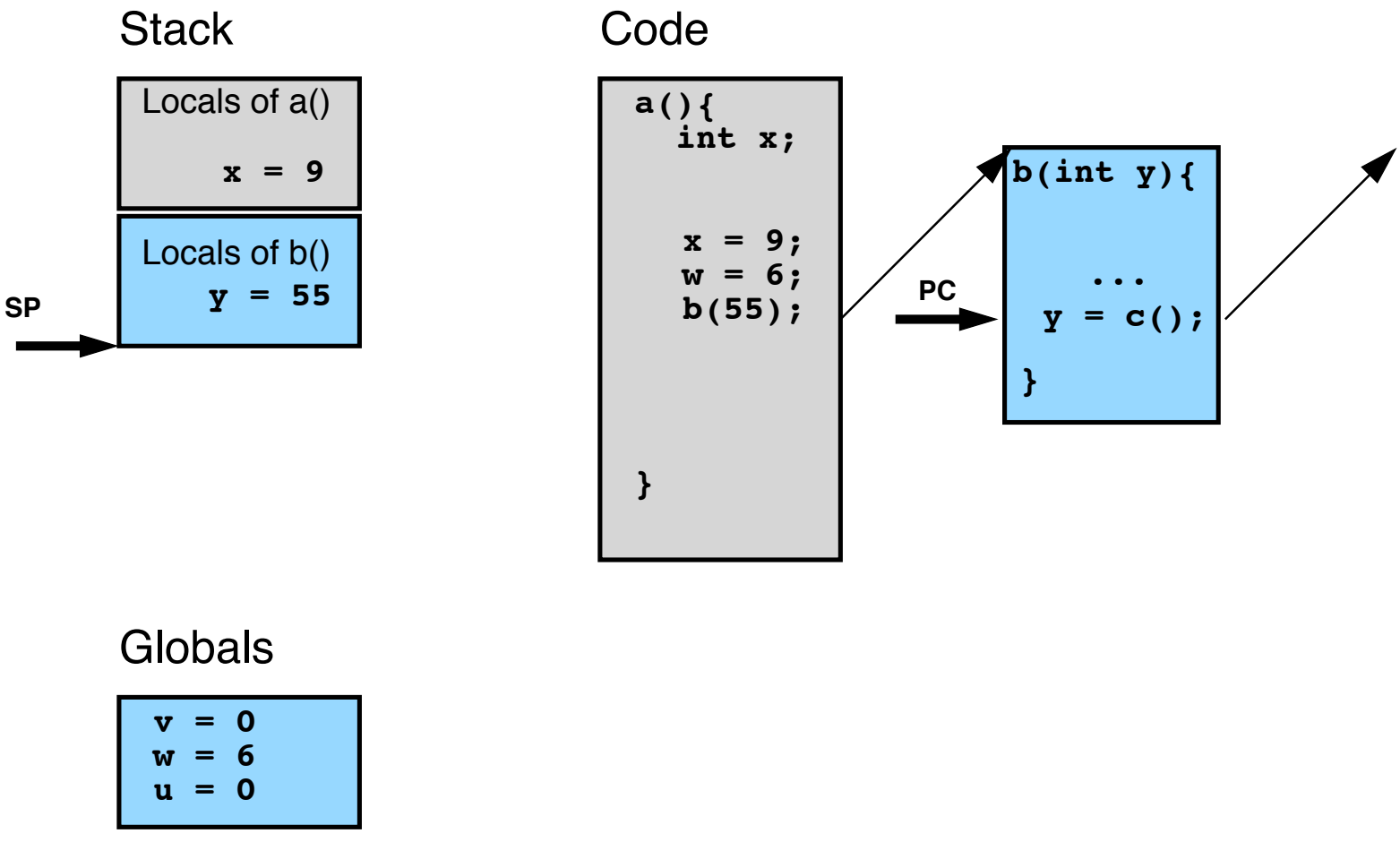
Execution of a C program



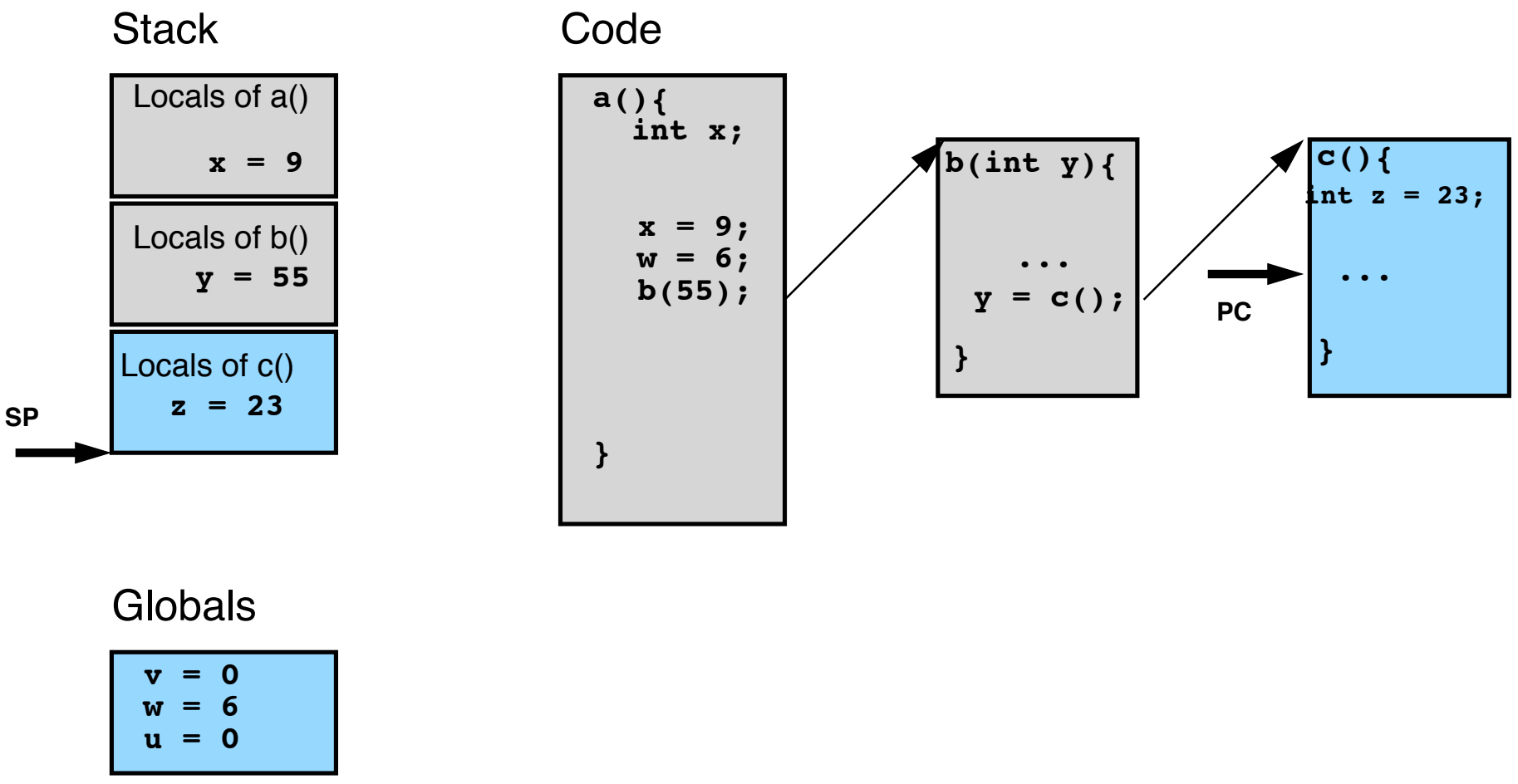
Execution of a C program



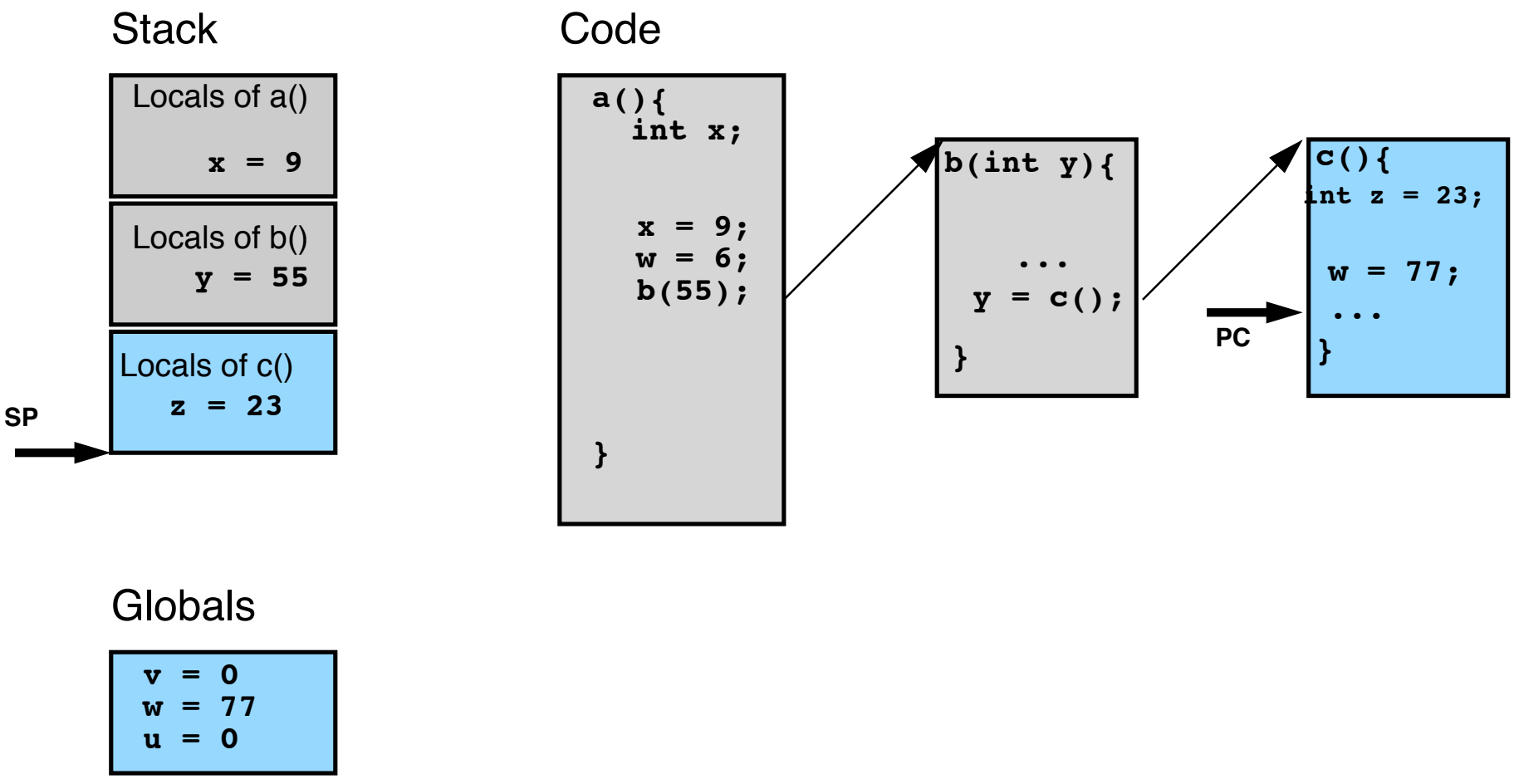
Execution of a C program



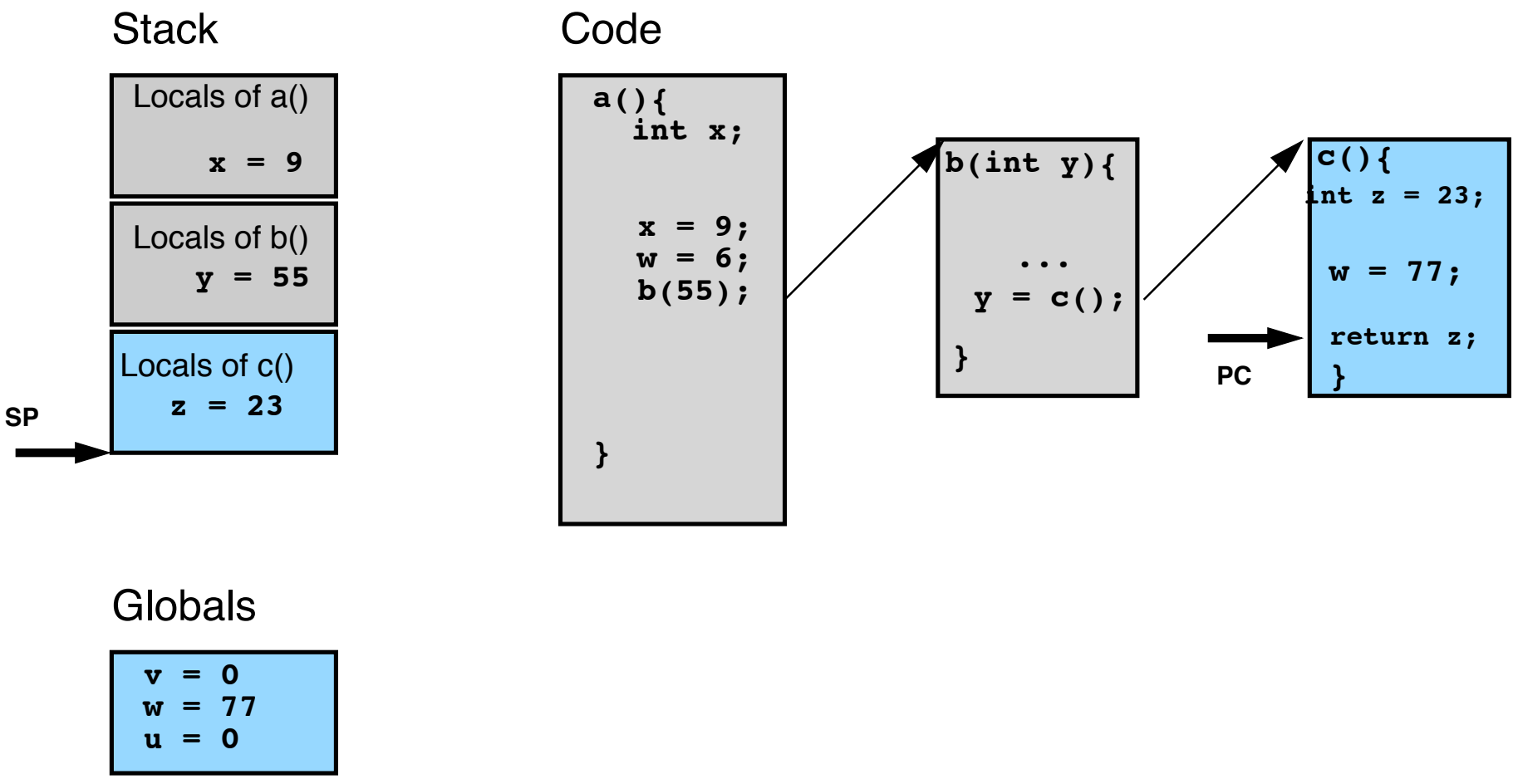
Execution of a C program



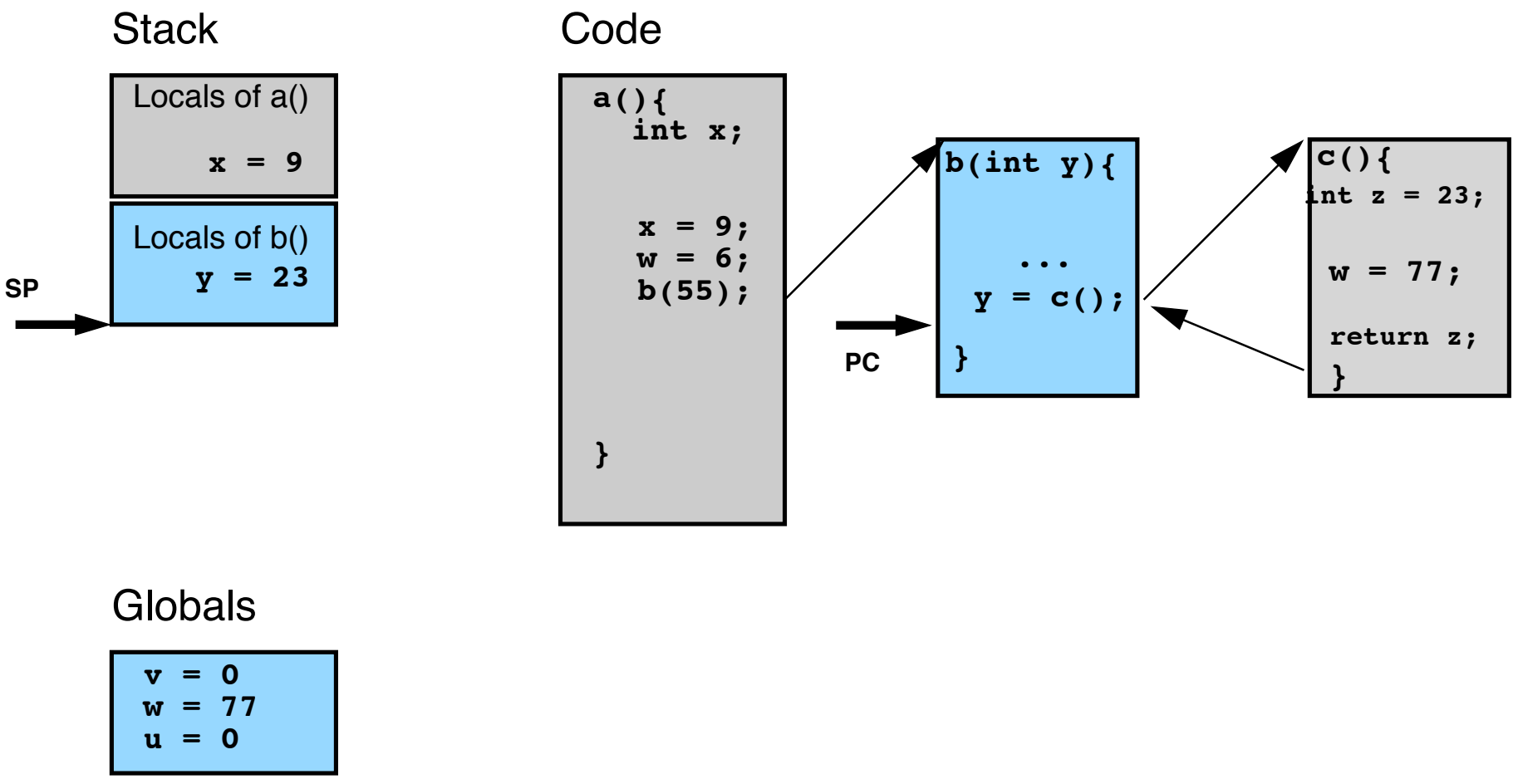
Execution of a C program



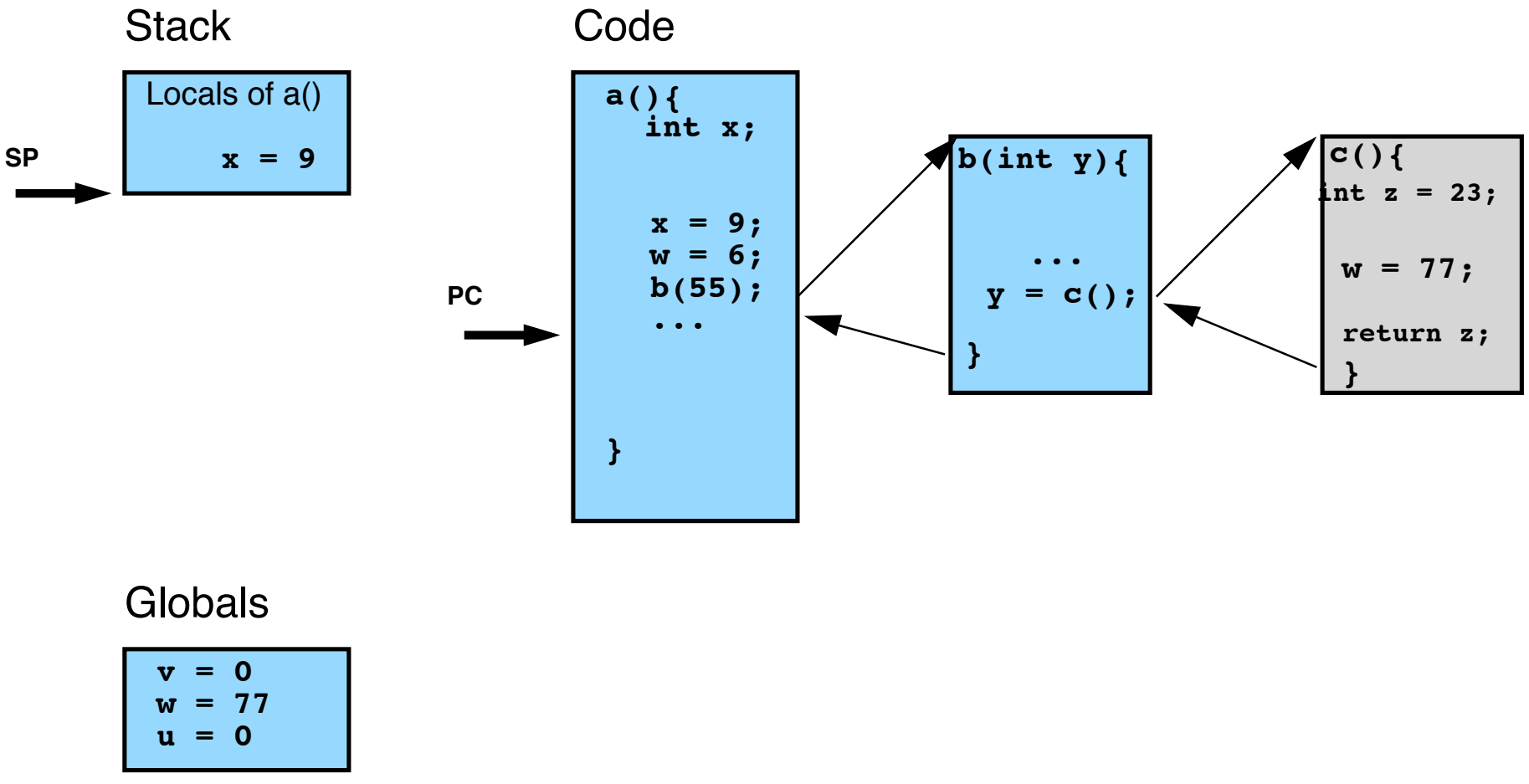
Execution of a C program



Execution of a C program



Execution of a C program



2 CPUs?

Imagine we had 2 CPUs, then we could run two programs at the same time!

One way of programming this in only 1 CPU is to keep track of 2 stack pointers and 2 program counters! How to do this is the content of lecture 4.

What is it about?

```
struct Params params;
```

```
void controller_main() {
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main() {
    struct Packet packet;
    while(1){
        radio_read(&packet);
        decode(&packet,&params);
    }
}
```

We want to provide means for these two mains to execute concurrently! As if we had 2 CPUs!

What is it about?

```
struct Params params;
```

```
void controller_main() {
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main() {
    struct Packet packet;
    while(1){
        radio_read(&packet);
        decode(&packet,&params);
    }
}
```

We want to provide means for these two mains to execute concurrently! As if we had 2 CPUs!

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function spawn takes a *function* as an argument!

The role of spawn is to provide one extra **Program Counter** and **Stack Pointer**

We should also provide a way of interleaving fragments of the threads.

First we will introduce a way of yielding execution so that another thread can take over. Later we will also see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function `spawn` takes a *function* as an argument!

The role of `spawn` is to provide one extra **Program Counter** and **Stack Pointer**

We should also provide a way of interleaving fragments of the threads.

First we will introduce a way of yielding execution so that another thread can take over. Later we will also see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function `spawn` takes a *function* as an argument!

The role of `spawn` is to provide one extra **Program Counter** and **Stack Pointer**

We should also provide a way of interleaving fragments of the threads.

First we will introduce a way of yielding execution so that another thread can take over. Later we will also see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function `spawn` takes a *function* as an argument!

The role of `spawn` is to provide one extra **Program Counter** and **Stack Pointer**

We should also provide a way of interleaving fragments of the threads.

First we will introduce a way of yielding execution so that another thread can take over. Later we will also see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function `spawn` takes a *function* as an argument!

The role of `spawn` is to provide one extra **Program Counter** and **Stack Pointer**

We should also provide a way of interleaving fragments of the threads.

First we will introduce a way of yielding execution so that another thread can take over. Later we will also see how this function might be invoked.

The kernel as a C library

tinythreads.h

```
struct thread_block;
typedef struct thread_block *thread;
void spawn(void (*code)(int), int arg);
void yield(void); // Your task in lab2!

struct mutex_block{
    int locked;
    thread waitQ;
};
typedef struct mutex_block mutex;

#define MUTEX_INIT {0,0}
void lock(mutex *m);    // Your task in lab2!
void unlock(mutex *m);  // Your task in lab2!
```

Using the kernel

Application.c

```
#include "tinythreads.h"
struct Params params;
int sonar_read();
void servo_write(int s);
void control(int d, int *s, struct Params *p);
void controller_main(int x);
void radio_read(struct Packet *p);
void decode(struct Packet *pk, struct Params *pr);
void decoder_main(int x);

void main(){
  spawn(controller_main,0);
  decoder_main(0);
}
```

Introducing setjmp/longjmp (non-local goto)

```
#include<stdio.h>
#include<setjmp.h>

jmp_buf bf;
```

```
main(int argc, char*argv[]){
  int x = atoi(argv[1]);
  while(1){
    if( setjmp(bf) ) break;
    f(argv[2],x);
  }
  printf("... Got here!\n\n");
}
```

```
void f(char * s, int n){
  int y = n-1;
  printf(s);
  if(y==0) longjmp(bf,1) ;
  else f(s, y);
}
```

Introducing setjmp/longjmp (non-local goto)

```
#include<stdio.h>
#include<setjmp.h>

jmp_buf bf;

main(int argc, char*argv[]){
  int x = atoi(argv[1]);
  while(1){
    if( setjmp(bf) ) break;
    f(argv[2],x);
  }
  printf("... Got here!\n\n");
}
```

```
void f(char * s, int n){
  int y = n-1;
  printf(s);
  if(y==0) longjmp(bf,1) ;
  else f(s, y);
}
```

Introducing setjmp/longjmp (non-local goto)

```
#include<stdio.h>
#include<setjmp.h>

jmp_buf bf;
```

```
main(int argc, char*argv[]){
  int x = atoi(argv[1]);
  while(1){
    if( setjmp(bf) ) break;
    f(argv[2],x);
  }
  printf("... Got here!\n\n");
}
```

```
void f(char * s, int n){
  int y = n-1;
  printf(s);
  if(y==0) longjmp(bf,1) ;
  else f(s, y);
}
```

setjmp/longjmp

1. The type `jmp_buf` is declared in the library `<setjmp.h>`. It is known to be an array.
2. The operation `setjmp(bf)` saves the **program counter** and the **stack pointer** to the `jmp_buf bf`.
   It returns 0.
3. The operation `longjmp(bf,n)` loads the content of `bf` to the **program counter** and the **stack pointer**.
   It returns `n`.

`longjmp(bf,n)` returns after the registers have been restored so it looks as if it is `setjmp(bf)` that returns `n`!

setjmp/longjmp

1. The type `jmp_buf` is declared in the library `<setjmp.h>`. It is known to be an array.
2. The operation `setjmp(bf)` saves the **program counter** and the **stack pointer** to the `jmp_buf bf`.
   It returns 0.
3. The operation `longjmp(bf,n)` loads the content of `bf` to the **program counter** and the **stack pointer**.
   It returns `n`.

`longjmp(bf,n)` returns after the registers have been restored so it looks as if it is `setjmp(bf)` that returns `n`!

setjmp/longjmp

1. The type `jmp_buf` is declared in the library `<setjmp.h>`. It is known to be an array.
2. The operation `setjmp(bf)` saves the **program counter** and the **stack pointer** to the `jmp_buf bf`.
   It returns 0.
3. The operation `longjmp(bf,n)` loads the content of `bf` to the **program counter** and the **stack pointer**.
   It returns `n`.

`longjmp(bf,n)` returns after the registers have been restored so it looks as if it is `setjmp(bf)` that returns `n`!

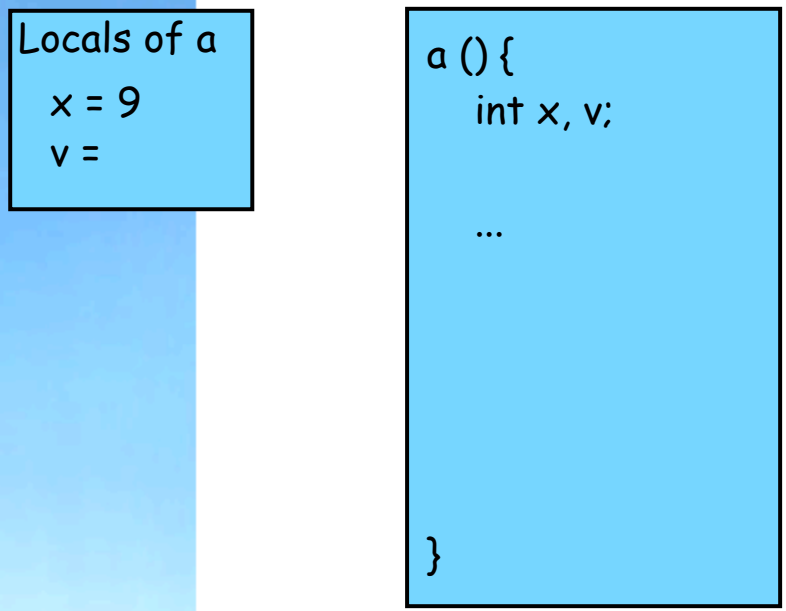
setjmp/longjmp

1. The type `jmp_buf` is declared in the library `<setjmp.h>`. It is known to be an array.
2. The operation `setjmp(bf)` saves the **program counter** and the **stack pointer** to the `jmp_buf bf`.
   It returns 0.
3. The operation `longjmp(bf,n)` loads the content of `bf` to the **program counter** and the **stack pointer**.
   It returns `n`.

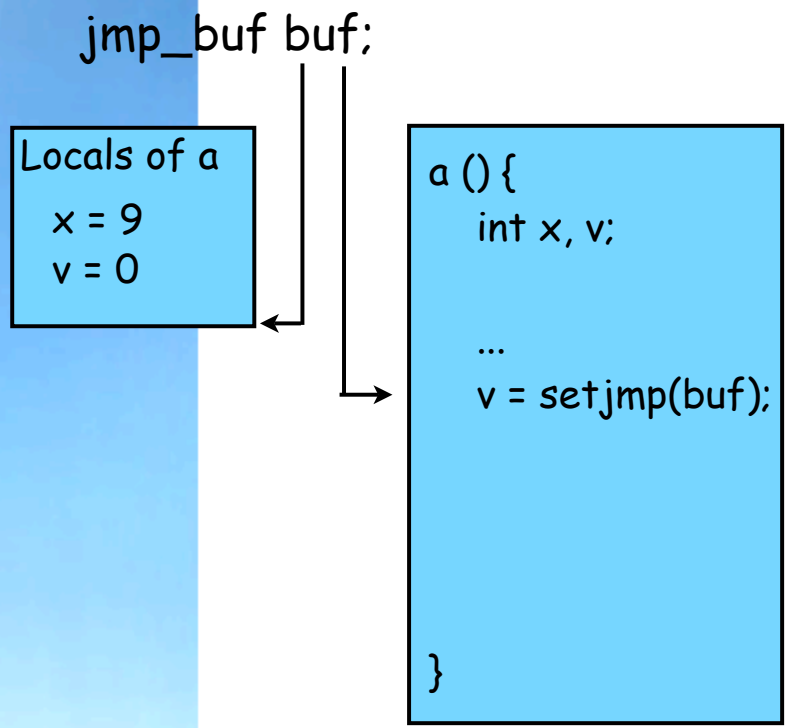
`longjmp(bf,n)` returns after the registers have been restored so it looks as if it is `setjmp(bf)` that returns `n`!

Using setjmp/longjmp

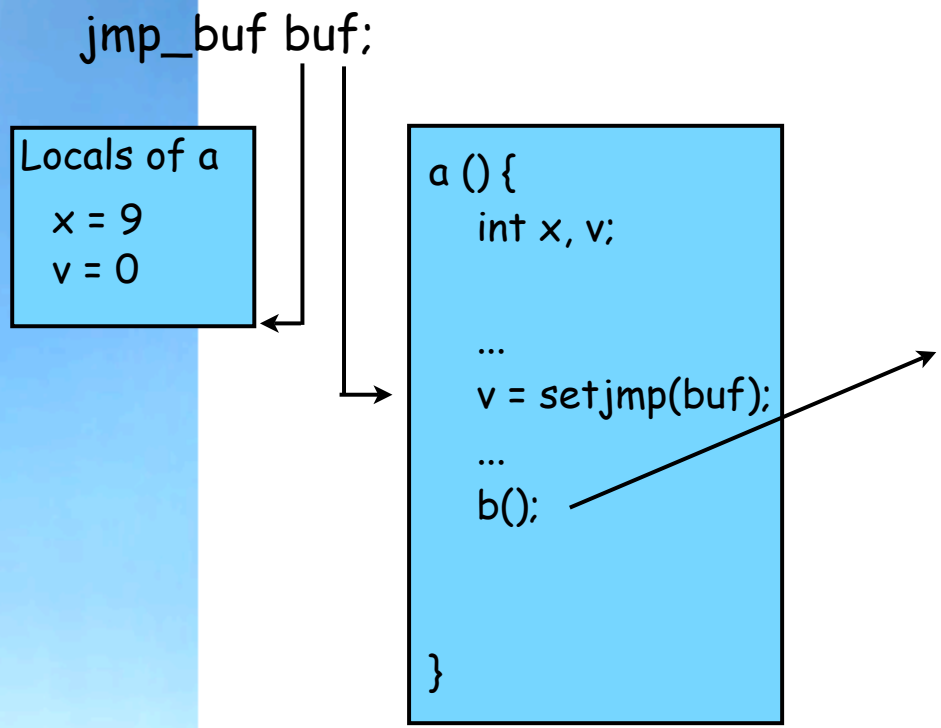
jmp_buf buf;



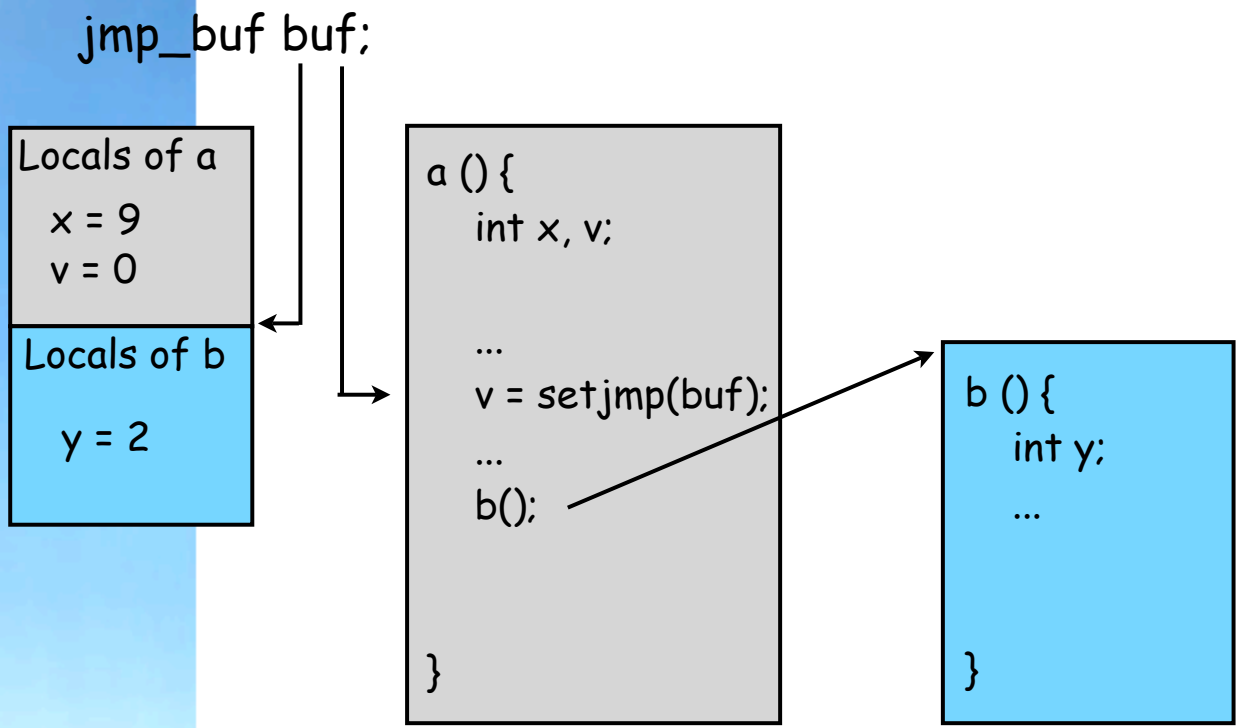
Using setjmp/longjmp



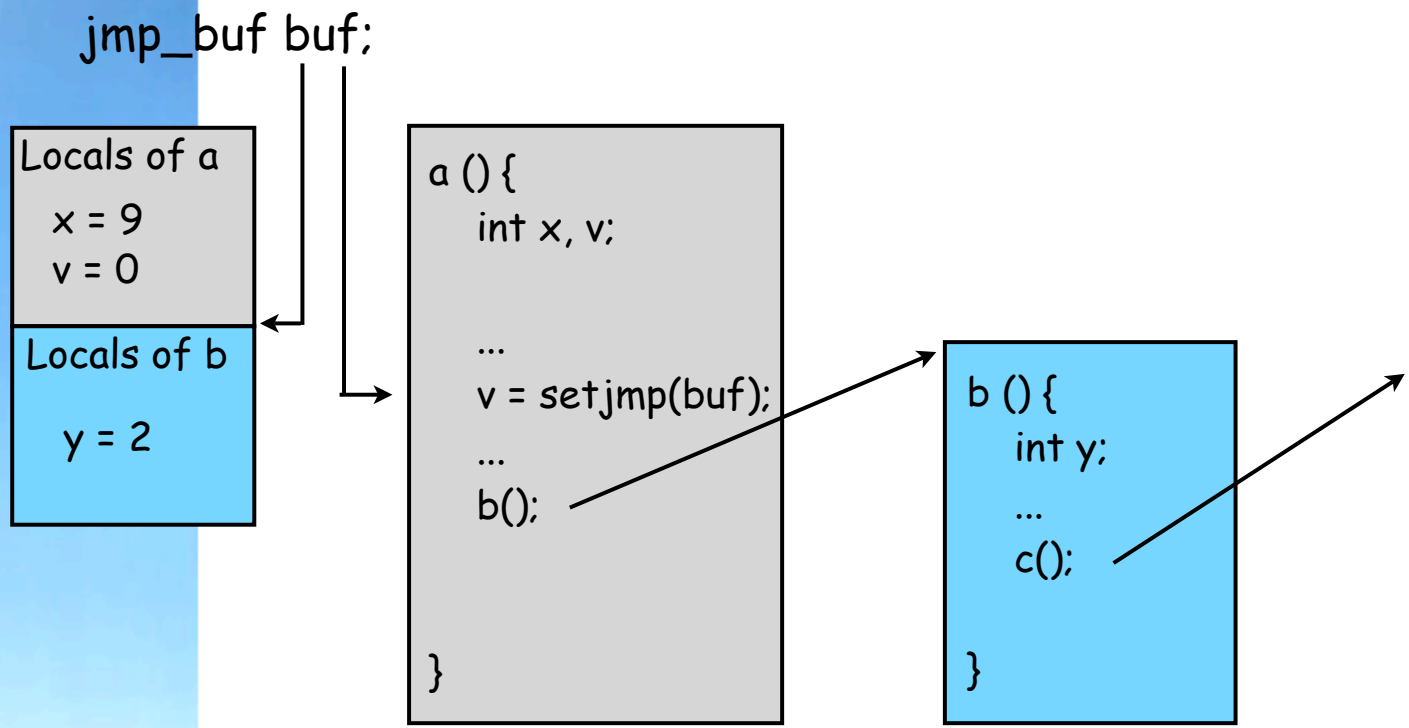
Using setjmp/longjmp



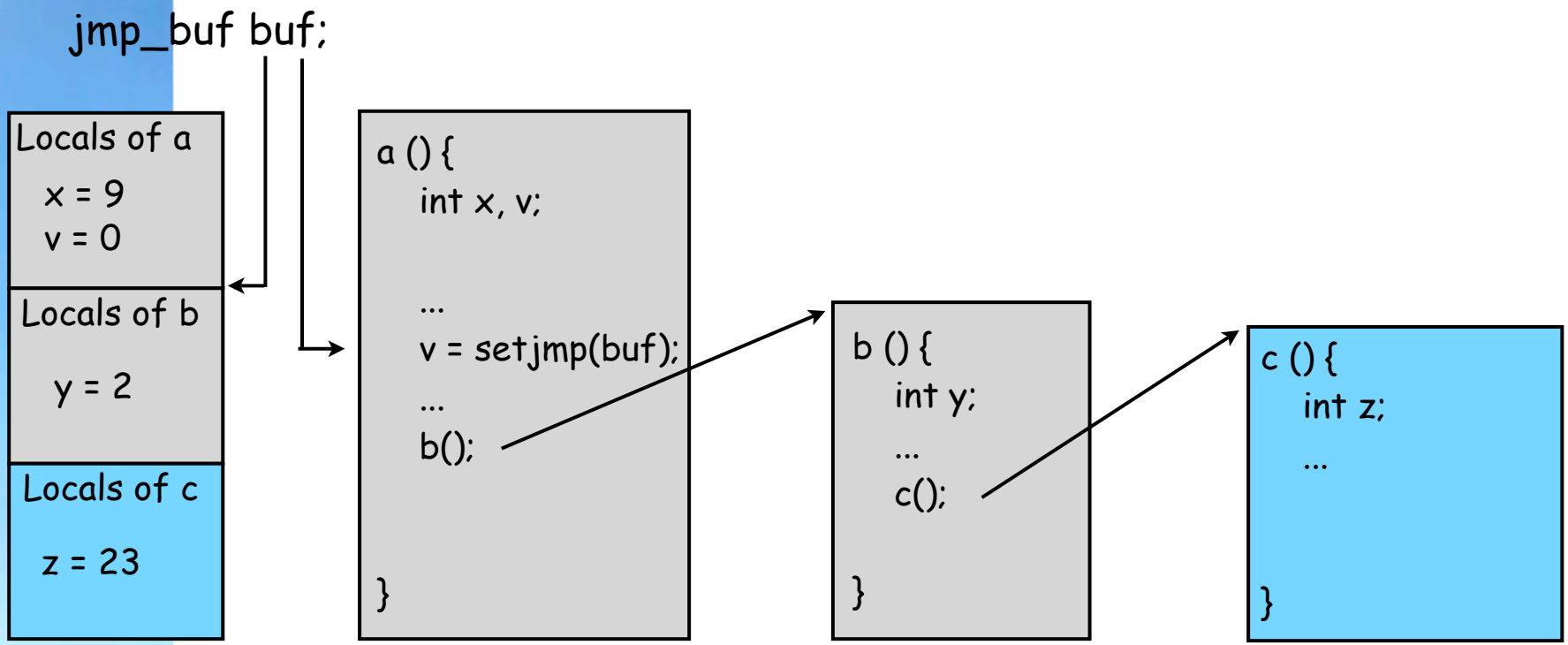
Using setjmp/longjmp



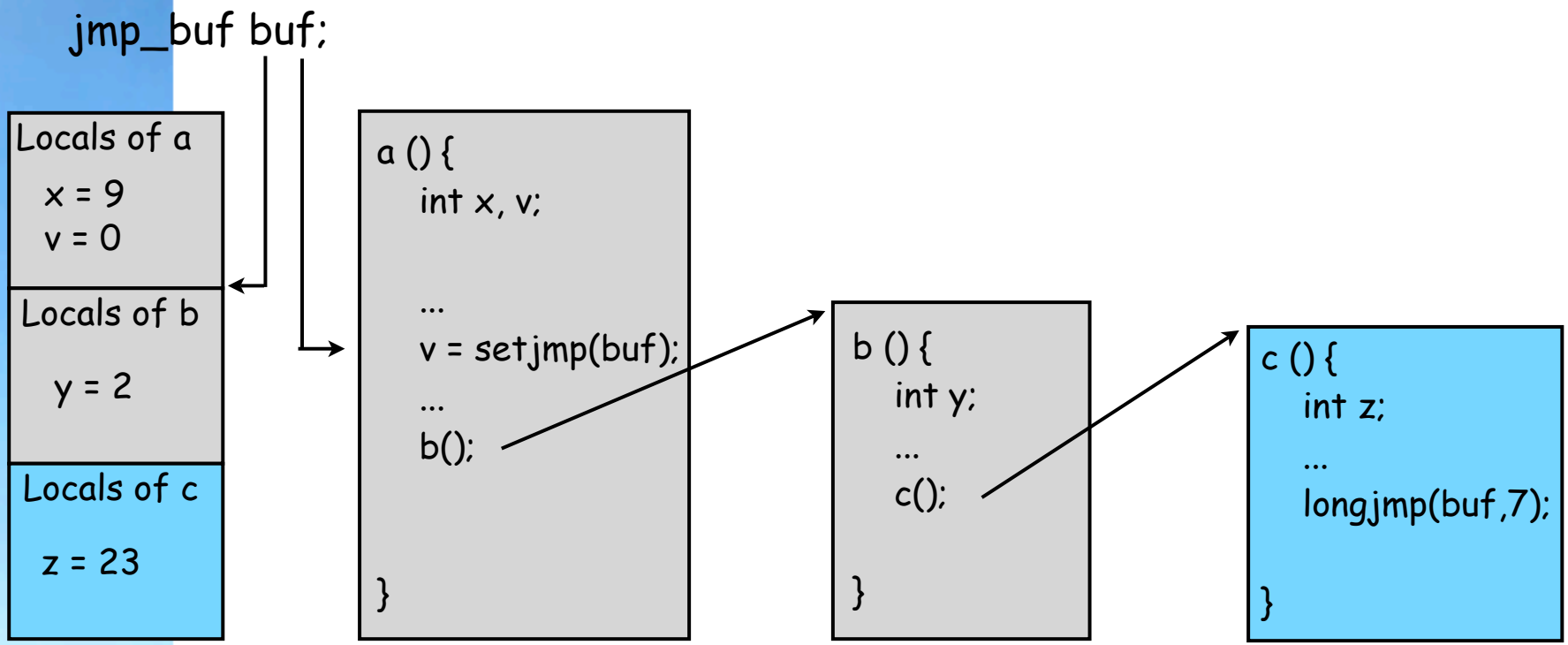
Using setjmp/longjmp



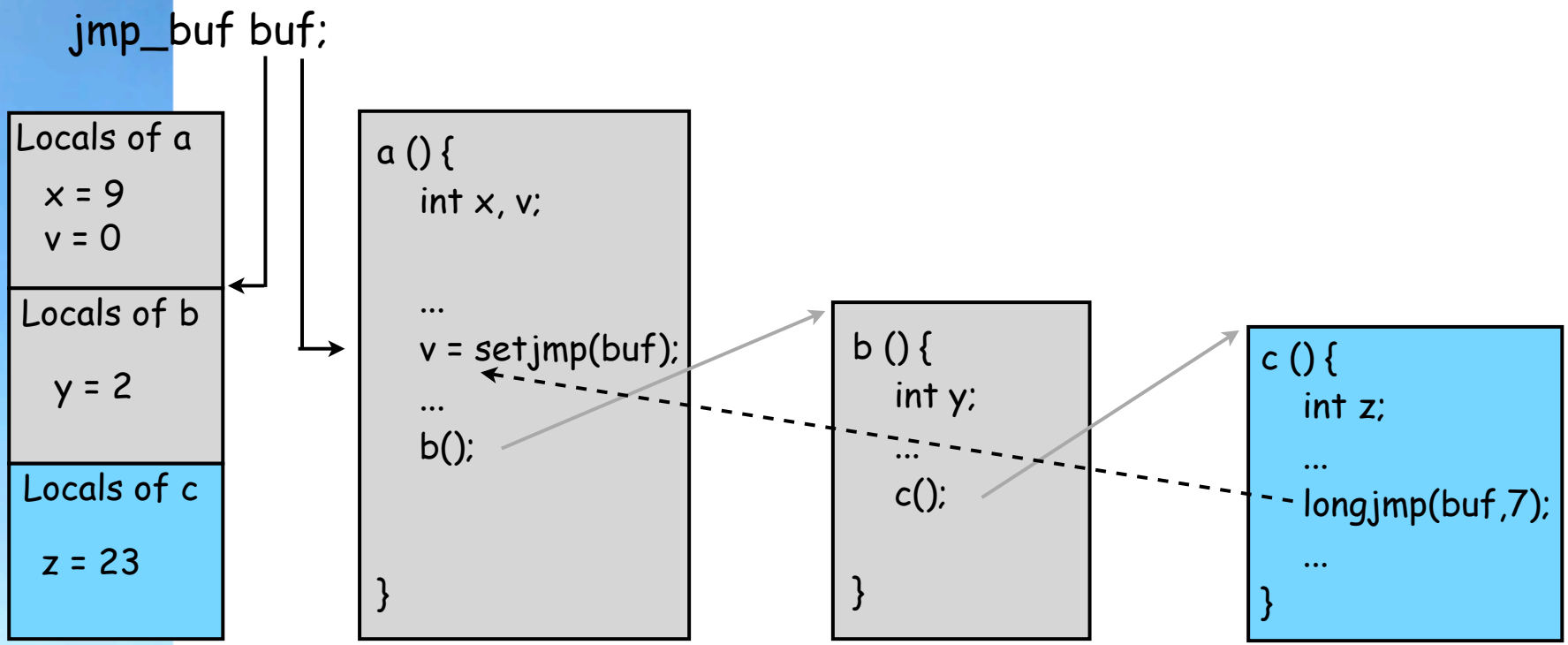
Using setjmp/longjmp



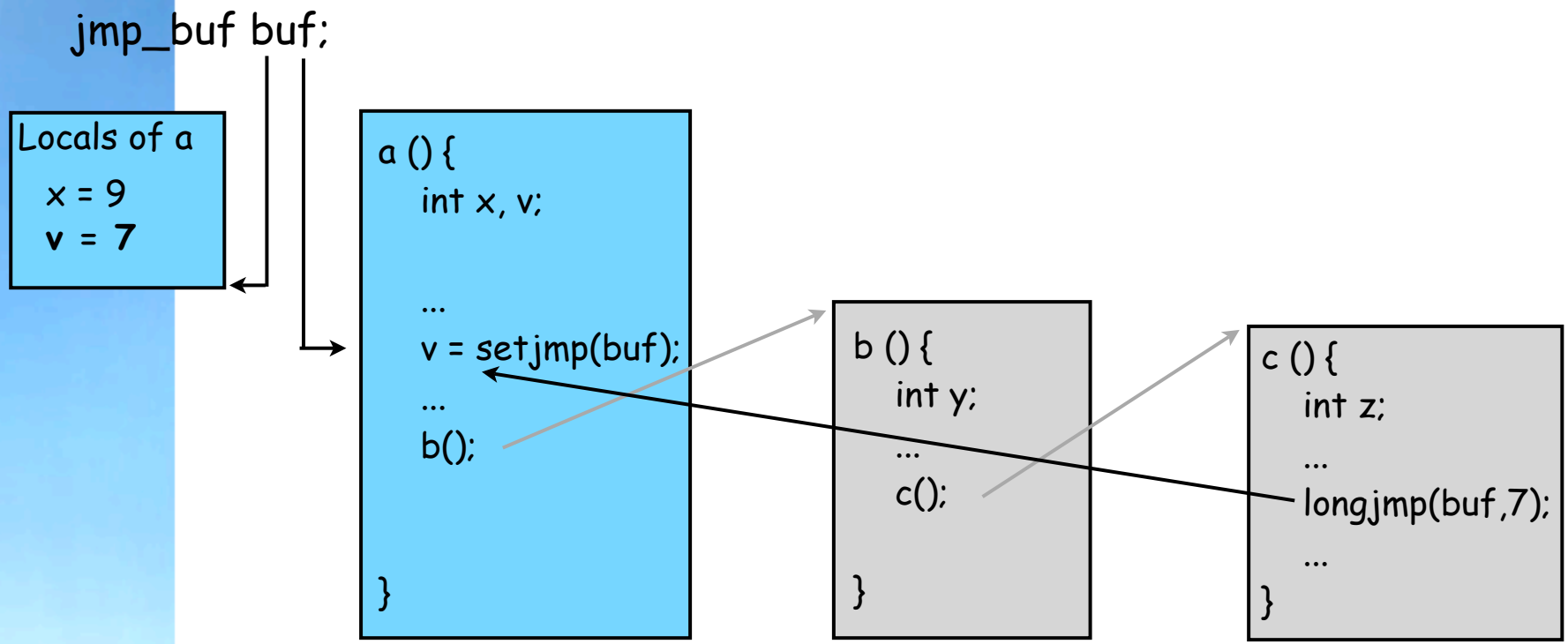
Using setjmp/longjmp



Using setjmp/longjmp



Using setjmp/longjmp



Using `setjmp/longjmp` for our purposes

```
setjmp(buf);
STACKPTR(buf) = malloc(...);
```

The macro `STACKPTR` will select the **stack pointer** part of the buffer.

We assign to it the address of a new chunk of memory.

The size of this chunk of memory is some number of times the size of a function-call stack frame.

This is very much platform dependent!

Using `setjmp/longjmp` for our purposes

```
setjmp(buf);
STACKPTR(buf) = malloc(...);
```

The macro `STACKPTR` will select the **stack pointer** part of the buffer.

We assign to it the address of a new chunk of memory.

The size of this chunk of memory is some number of times the size of a function-call stack frame.

This is very much platform dependent!

Using `setjmp/longjmp` for our purposes

```
setjmp(buf);
STACKPTR(buf) = malloc(...);
```

The macro `STACKPTR` will select the **stack pointer** part of the buffer.

We assign to it the address of a new chunk of memory.

The size of this chunk of memory is some number of times the size of a function-call stack frame.

This is very much platform dependent!

Using `setjmp/longjmp` for our purposes

```
setjmp(buf);
STACKPTR(buf) = malloc(...);
```

The macro `STACKPTR` will select the **stack pointer** part of the buffer.

We assign to it the address of a new chunk of memory.

The size of this chunk of memory is some number of times the size of a function-call stack frame.

This is very much platform dependent!

The trick

`spawn` will do the trick

```
setjmp(buf);
STACKPTR(buf) = malloc(...);
```

before calling the function to be executed. The function will then run with its own stack!

After this, the context-switching between threads is just

```
setjmp(A);longjmp(B);
setjmp(B);longjmp(C);
setjmp(C);longjmp(A);
```

The trick

`spawn` will do the trick

```
setjmp(buf);
STACKPTR(buf) = malloc(...);
```

before calling the function to be executed. The function will then run with its own stack!

After this, the context-switching between threads is just

```
setjmp(A);longjmp(B);
setjmp(B);longjmp(C);
setjmp(C);longjmp(A);
```

Completing the trick

We will have to keep track of the threads, so we introduce a data structure describing a thread.

```
struct Thread_Block{
    void    (*fun)(int)   // function to run
    int     arg;          // argument to the above
    jmp_buf context,      // pc and sp
    ...                   // ...
};
typedef struct Thread_Block *Thread
```

We will keep track of threads using global variables for

1. a queue of `Threads`: the ready queue
2. and the `current` thread.

Completing the trick

We will have to keep track of the threads, so we introduce a data structure describing a thread.

```
struct Thread_Block{
    void    (*fun)(int)   // function to run
    int     arg;          // argument to the above
    jmp_buf context,      // pc and sp
    ...                   // ...
};
typedef struct Thread_Block *Thread
```

We will keep track of threads using global variables for

1. a queue of `Threads`: the ready queue
2. and the `current` thread.

Defining spawn

```
void spawn(void (*fun)(int), int arg){
  Thread t = malloc(...);
  t->fun    = fun;
  t->arg    = arg;
  if(setjmp(t->context)!=0){
      // this will be done when a longjmp leads here:
      current->fun(current->arg);
      // What has to be done when the thread
      // we spawn for fun terminates?
  }
  // DO the stack pointer trick:
  STACKPTR(t->context) = malloc(...);
  // enqueue t in the ready queue!
}
```

The ready queue, the current thread and yield

Yielding control

By keeping the runnable threads in a queue, we can define a function `yield()` to switch execution to another thread.

`yield()` must

- enqueue the current thread in the ready queue
- pick a new thread from the ready queue and make it the current thread
- perform the `setjmp(A);longjmp(B);` trick (also called dispatch)

The ready queue, the current thread and yield

Yielding control

By keeping the runnable threads in a queue, we can define a function `yield()` to switch execution to another thread.

`yield()` must

- enqueue the current thread in the ready queue
- pick a new thread from the ready queue and make it the current thread
- perform the `setjmp(A);longjmp(B);` trick (also called dispatch)

The ready queue, the current thread and yield

Yielding control

By keeping the runnable threads in a queue, we can define a function `yield()` to switch execution to another thread.

`yield()` must

- enqueue the current thread in the ready queue
- pick a new thread from the ready queue and make it the current thread
- perform the `setjmp(A);longjmp(B);` trick (also called dispatch)

The ready queue, the current thread and yield

Yielding control

By keeping the runnable threads in a queue, we can define a function `yield()` to switch execution to another thread.

`yield()` must

- enqueue the current thread in the ready queue
- pick a new thread from the ready queue and make it the current thread
- perform the `setjmp(A);longjmp(B);` trick (also called dispatch)

The ready queue, the current thread and yield

Yielding control

By keeping the runnable threads in a queue, we can define a function `yield()` to switch execution to another thread.

`yield()` must

- enqueue the current thread in the ready queue
- pick a new thread from the ready queue and make it the current thread
- perform the `setjmp(A);longjmp(B);` trick (also called dispatch)

Dispatch

```
void dispatch(Thread next){
  if(setjmp(current->context)==0){
      current = next;
      longjmp(next->context,1);
  }
}
```

The function returns directly when control later enters via the *fake return* from `setjmp`.

Dispatch

```
void dispatch(Thread next){
  if(setjmp(current->context)==0){
      current = next;
      longjmp(next->context,1);
  }
}
```

The function returns directly when control later enters via the *fake return* from `setjmp`.

Defining spawn, more details

```
void spawn(void (*fun)(int), int arg){
  Thread t = malloc(...);
  t->fun    = fun;
  t->arg    = arg;
  if(setjmp(t->context)!=0){
       // this will be done when a longjmp leads here:
       current->fun(current->arg);
       // free memory allocated to t!
       dispatch(dequeue(&readyQ));
  }
  // DO the stack pointer trick:
  STACKPTR(t->context) = malloc(...);
  enqueue(newp, &readyQ);
}
```

Who is the first current thread?

`main`

For `main` there is a **PC** and a **SP** and execution is set off when turning power on!

But `yield` should be able to deal with it as any other thread!

We introduce a thread block without initialization to be the first current thread. The first dispatch will set the **PC** and **SP** before enqueuing it in the ready queue!

Who is the first current thread?

`main`

For `main` there is a **PC** and a **SP** and execution is set off when turning power on!

But `yield` should be able to deal with it as any other thread!

We introduce a thread block without initialization to be the first current thread. The first dispatch will set the **PC** and **SP** before enqueuing it in the ready queue!

Who is the first current thread?

`main`

For `main` there is a **PC** and a **SP** and execution is set off when turning power on!

But `yield` should be able to deal with it as any other thread!

We introduce a thread block without initialization to be the first current thread. The first dispatch will set the **PC** and **SP** before enqueuing it in the ready queue!

Tracing an example

We want to explore what goes on when the infrastructure we introduced is used help run a program with 2 threads!

The example

Two threads calculating prime numbers starting from different start values.

The **Program Counter** and **Stack Pointer** are denoted by thick arrows.

Global variables in the middle, stacks to the right.

While the code in the example is not complete, you will work with the complete version in laboration 2 (next week)!

Tracing an example

We want to explore what goes on when the infrastructure we introduced is used help run a program with 2 threads!

The example

Two threads calculating prime numbers starting from different start values.

The **Program Counter** and **Stack Pointer** are denoted by thick arrows.

Global variables in the middle, stacks to the right.

While the code in the example is not complete, you will work with the complete version in laboration 2 (next week)!

Tracing an example

We want to explore what goes on when the infrastructure we introduced is used help run a program with 2 threads!

The example

Two threads calculating prime numbers starting from different start values.

The **Program Counter** and **Stack Pointer** are denoted by thick arrows.

Global variables in the middle, stacks to the right.

While the code in the example is not complete, you will work with the complete version in laboration 2 (next week)!

Tracing an example

We want to explore what goes on when the infrastructure we introduced is used help run a program with 2 threads!

The example

Two threads calculating prime numbers starting from different start values.

The **Program Counter** and **Stack Pointer** are denoted by thick arrows.

Global variables in the middle, stacks to the right.

While the code in the example is not complete, you will work with the complete version in laboration 2 (next week)!

Tracing an example

We want to explore what goes on when the infrastructure we introduced is used help run a program with 2 threads!

The example

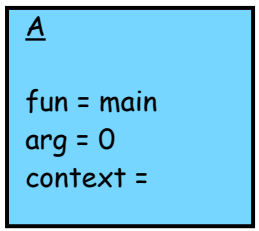
Two threads calculating prime numbers starting from different start values.

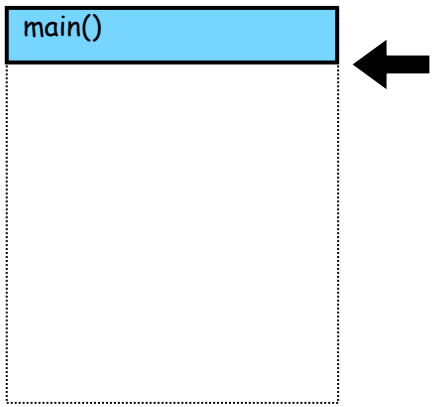
The **Program Counter** and **Stack Pointer** are denoted by thick arrows.

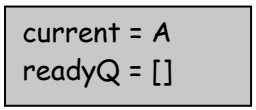
Global variables in the middle, stacks to the right.

While the code in the example is not complete, you will work with the complete version in laboration 2 (next week)!

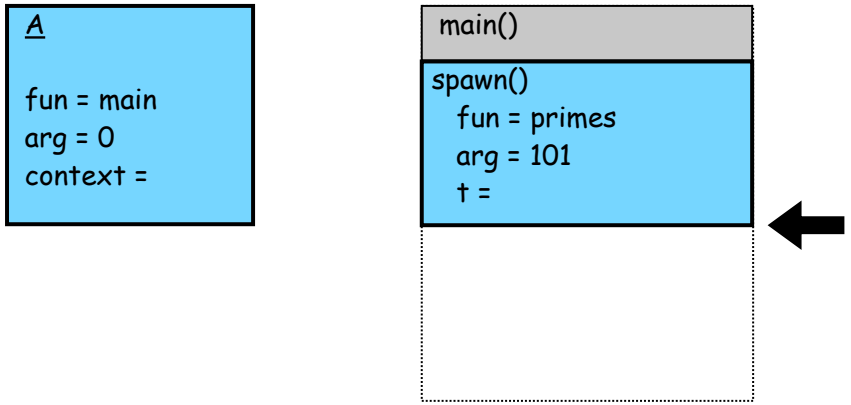
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```





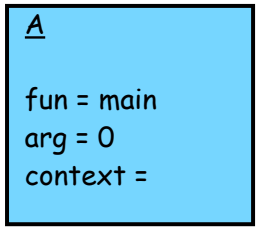


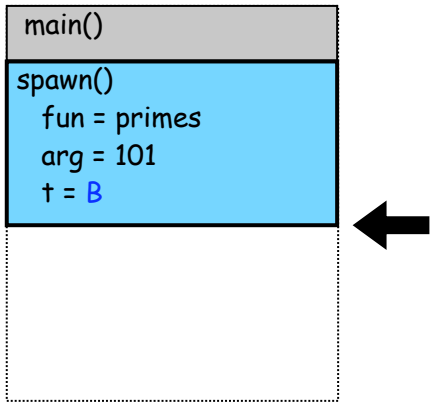
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

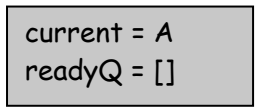


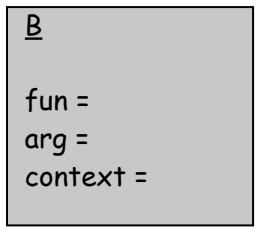
current = A
readyQ = []

```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
}
```

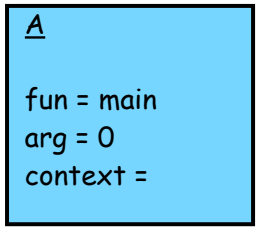


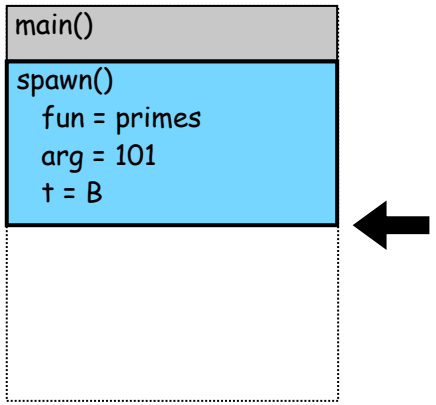


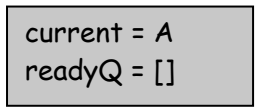


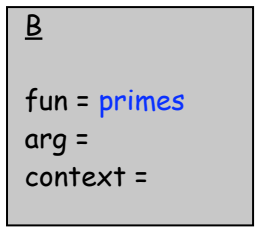


```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

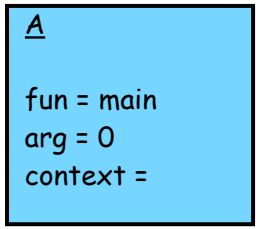


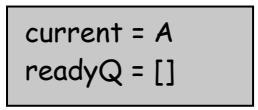


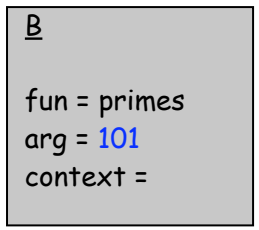


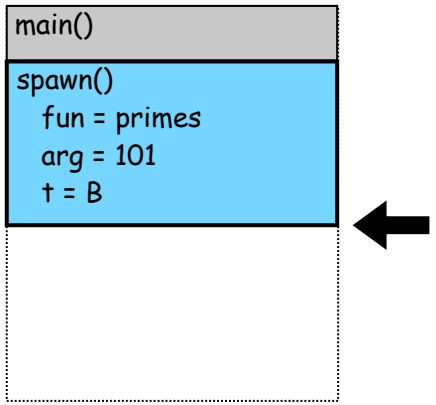


```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

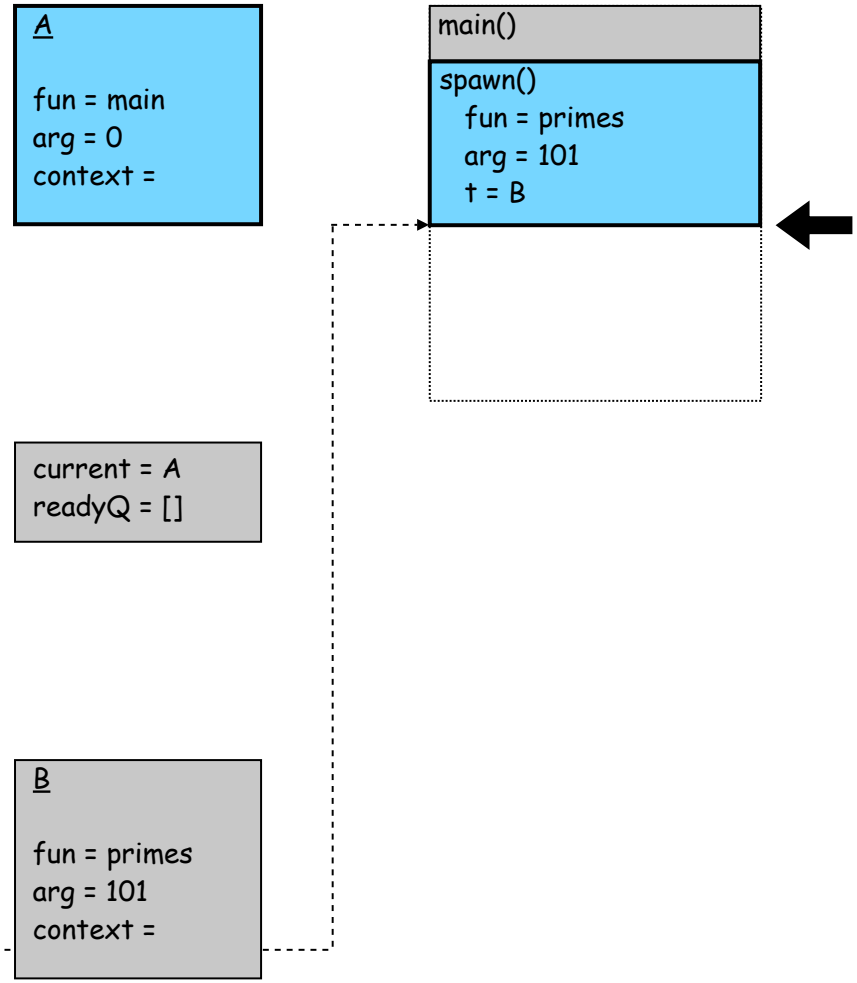




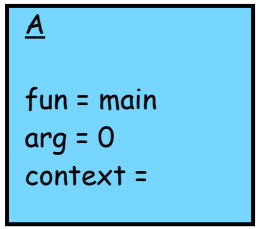


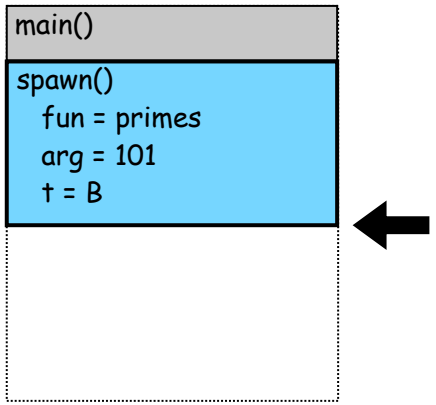


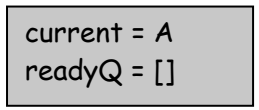
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

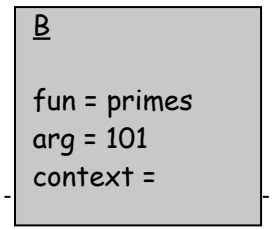


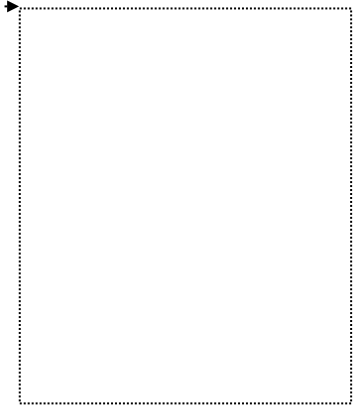
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



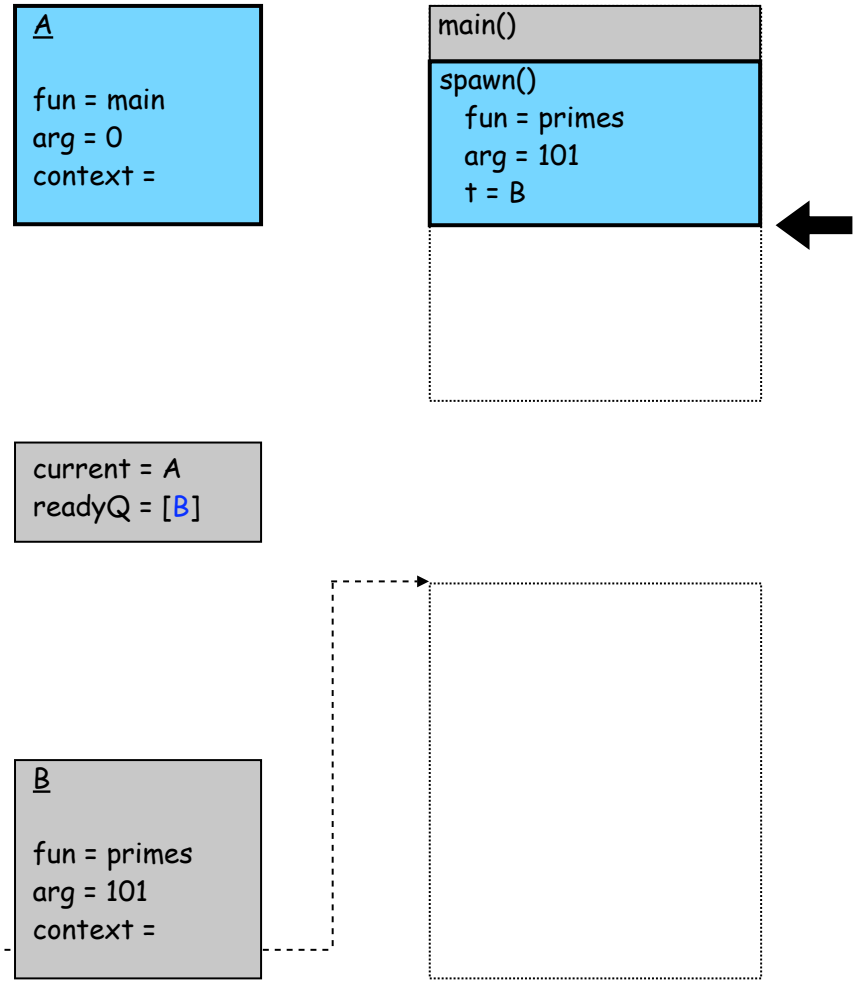




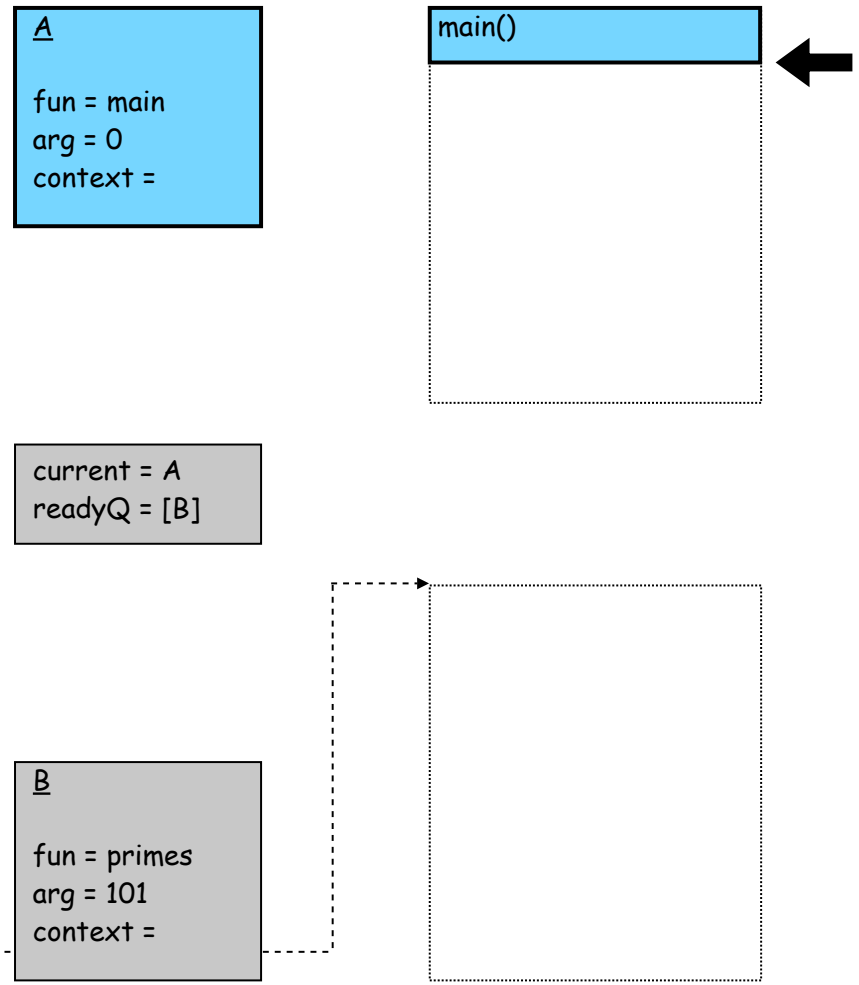




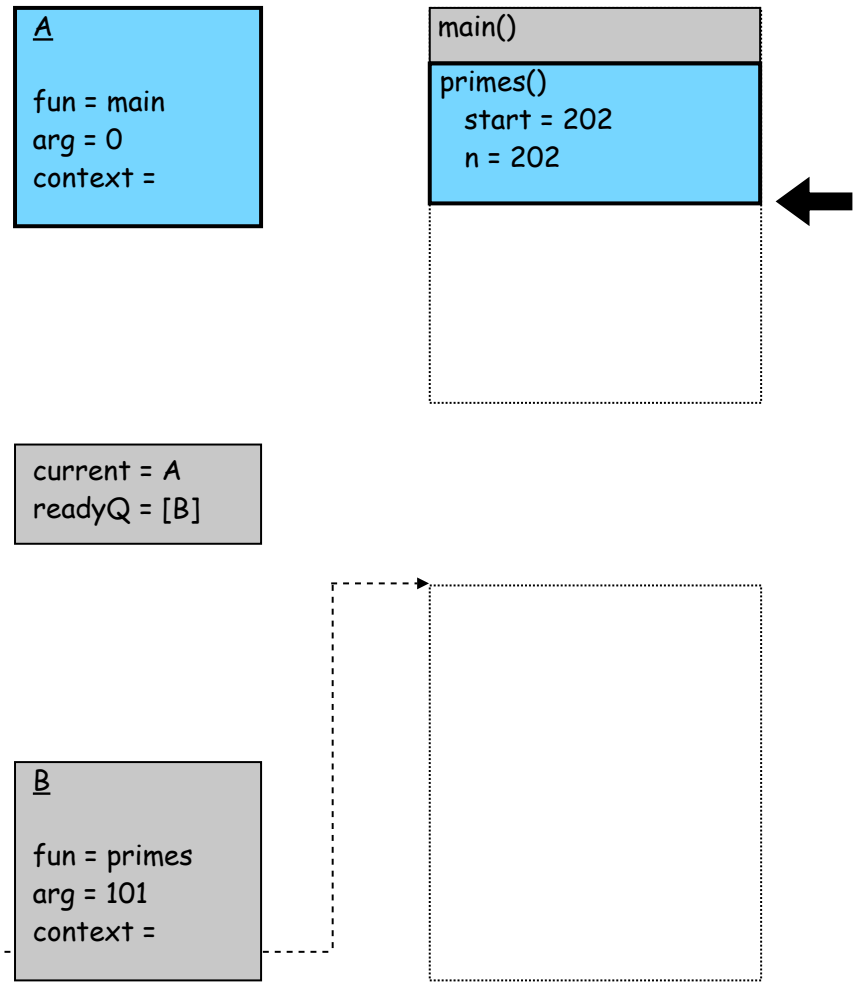
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



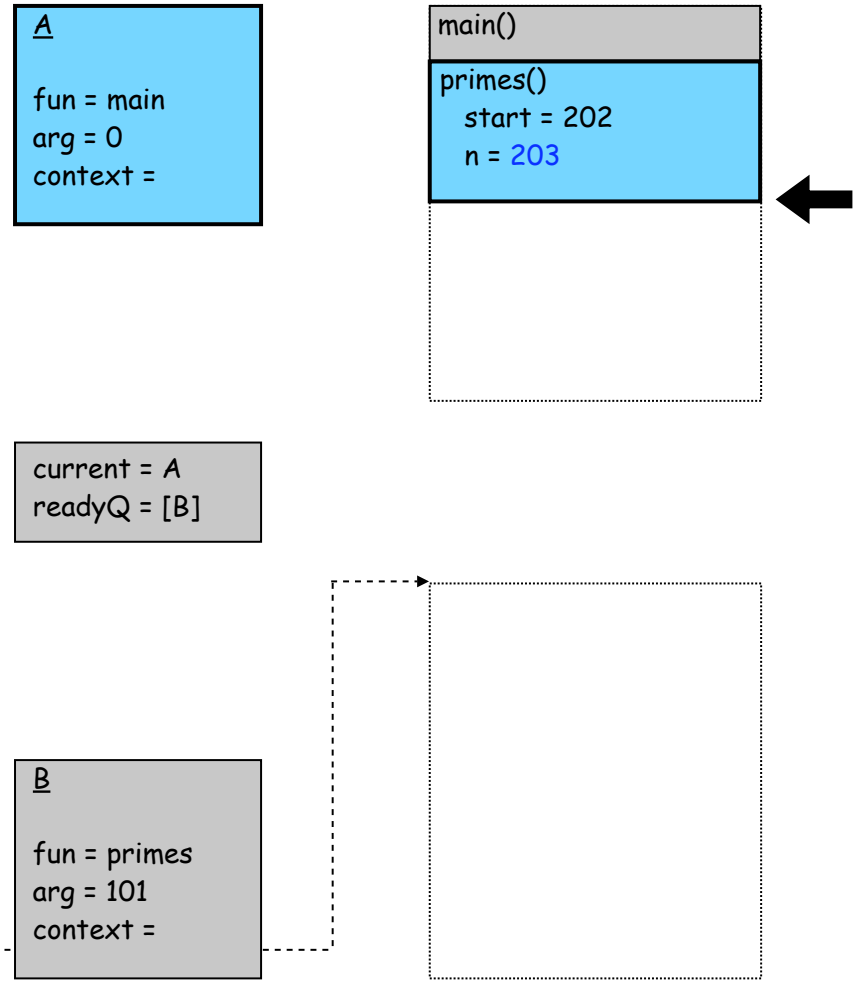
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



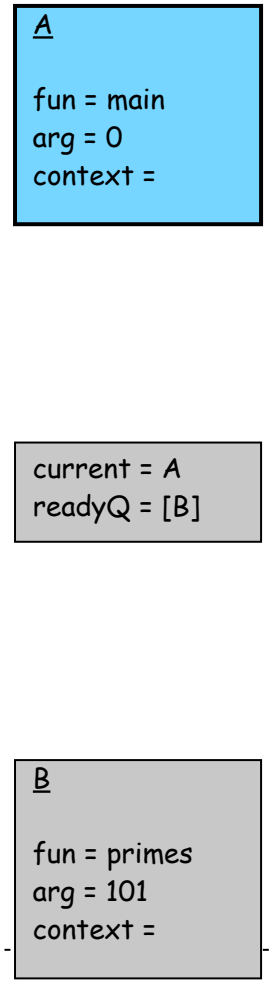
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
}
```



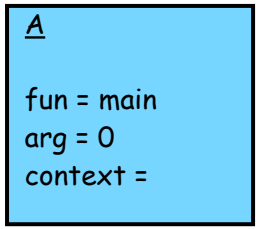
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

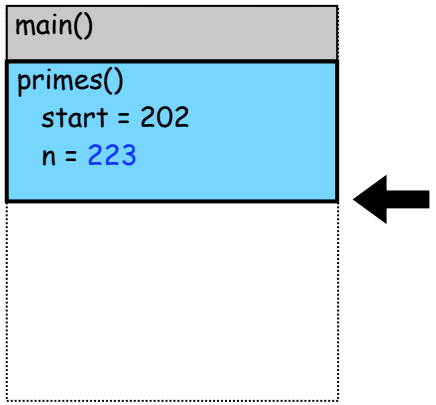


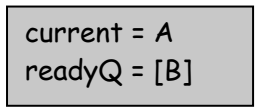
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

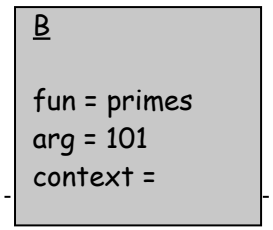


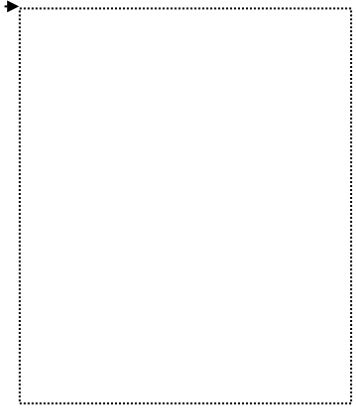
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



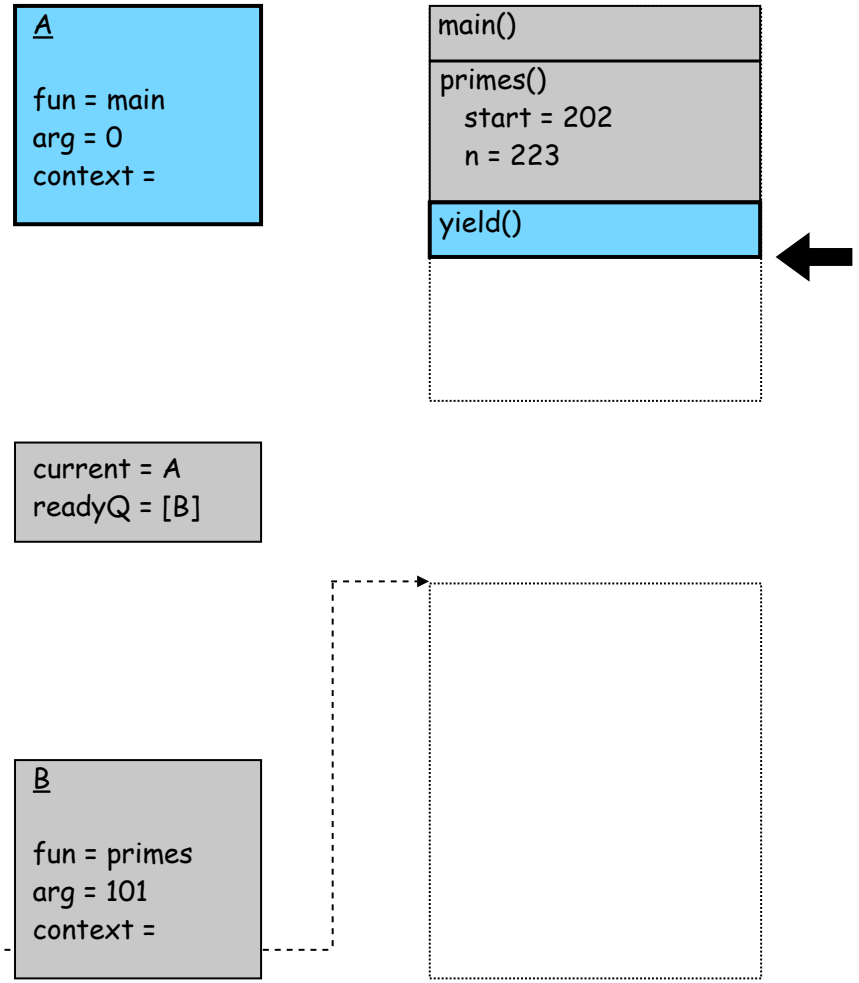




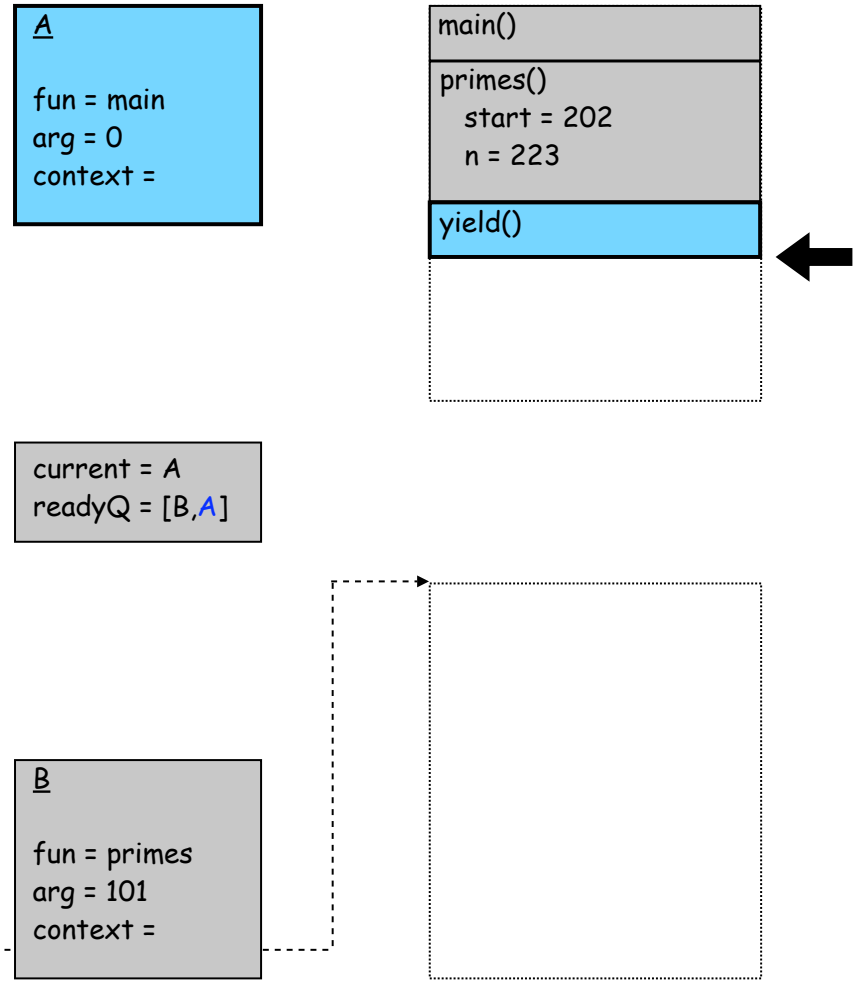




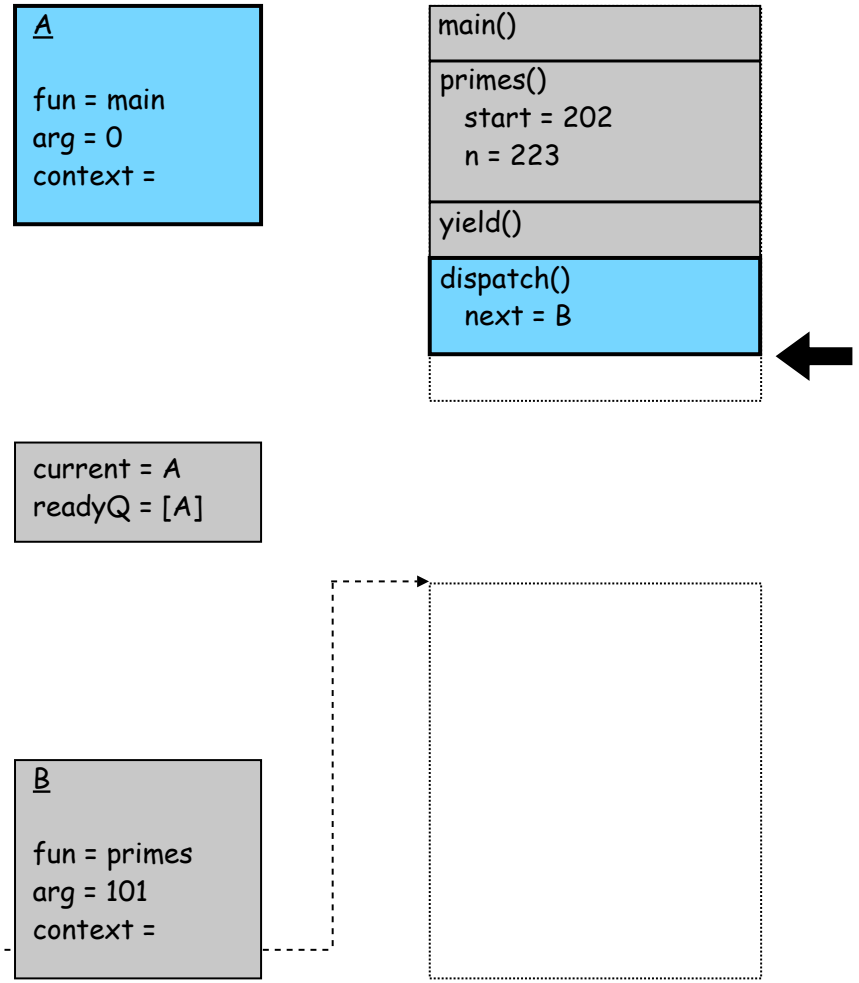
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



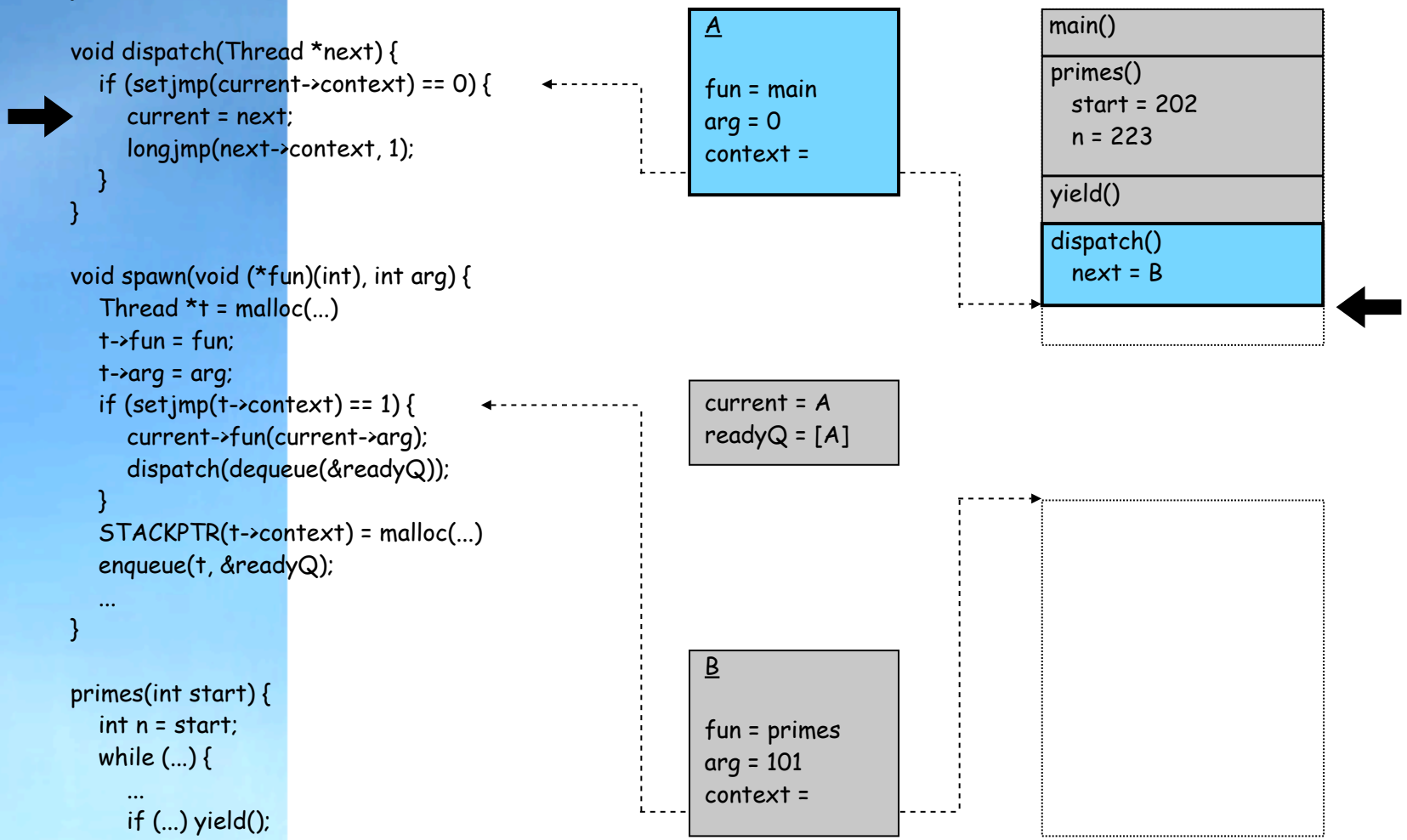
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



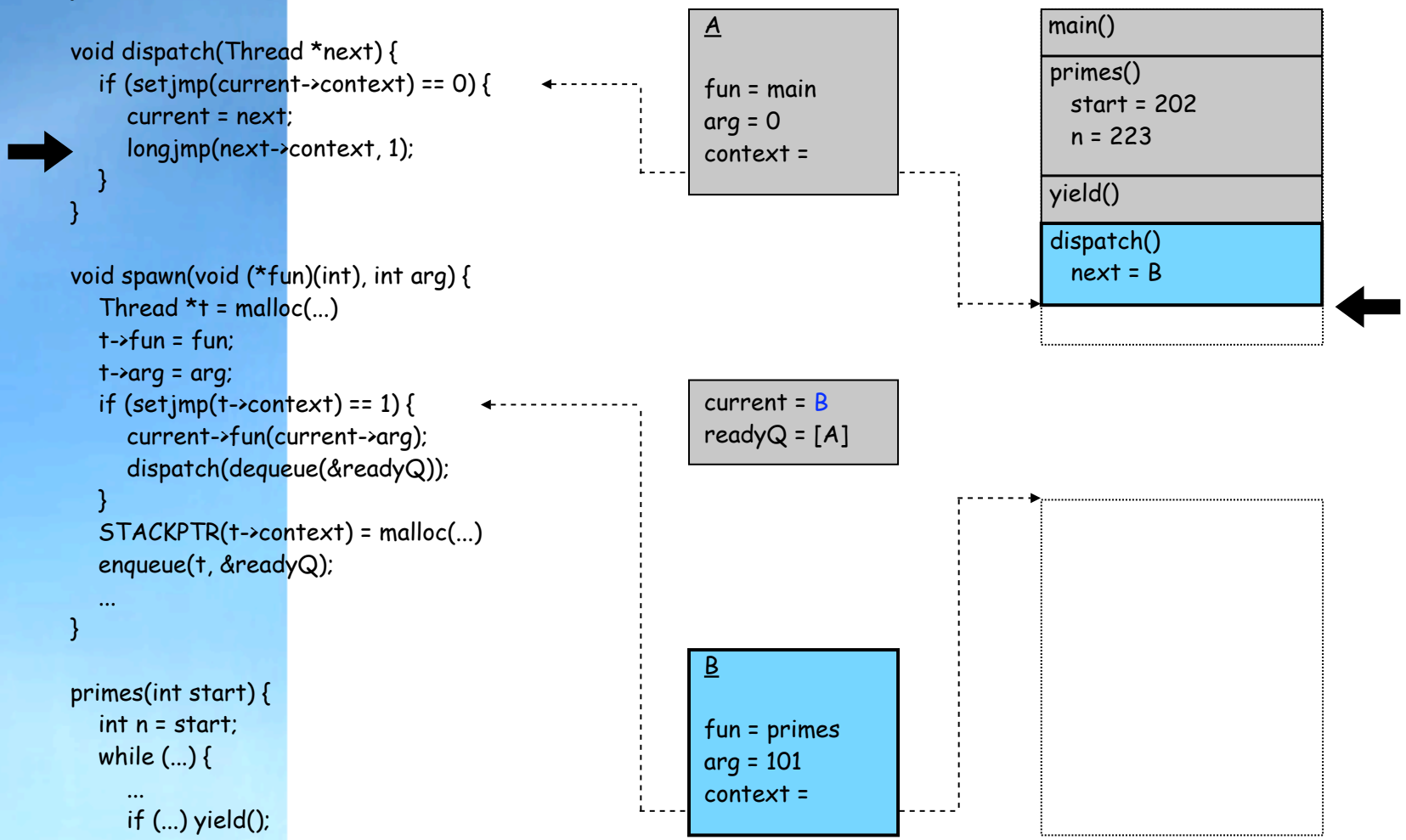
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



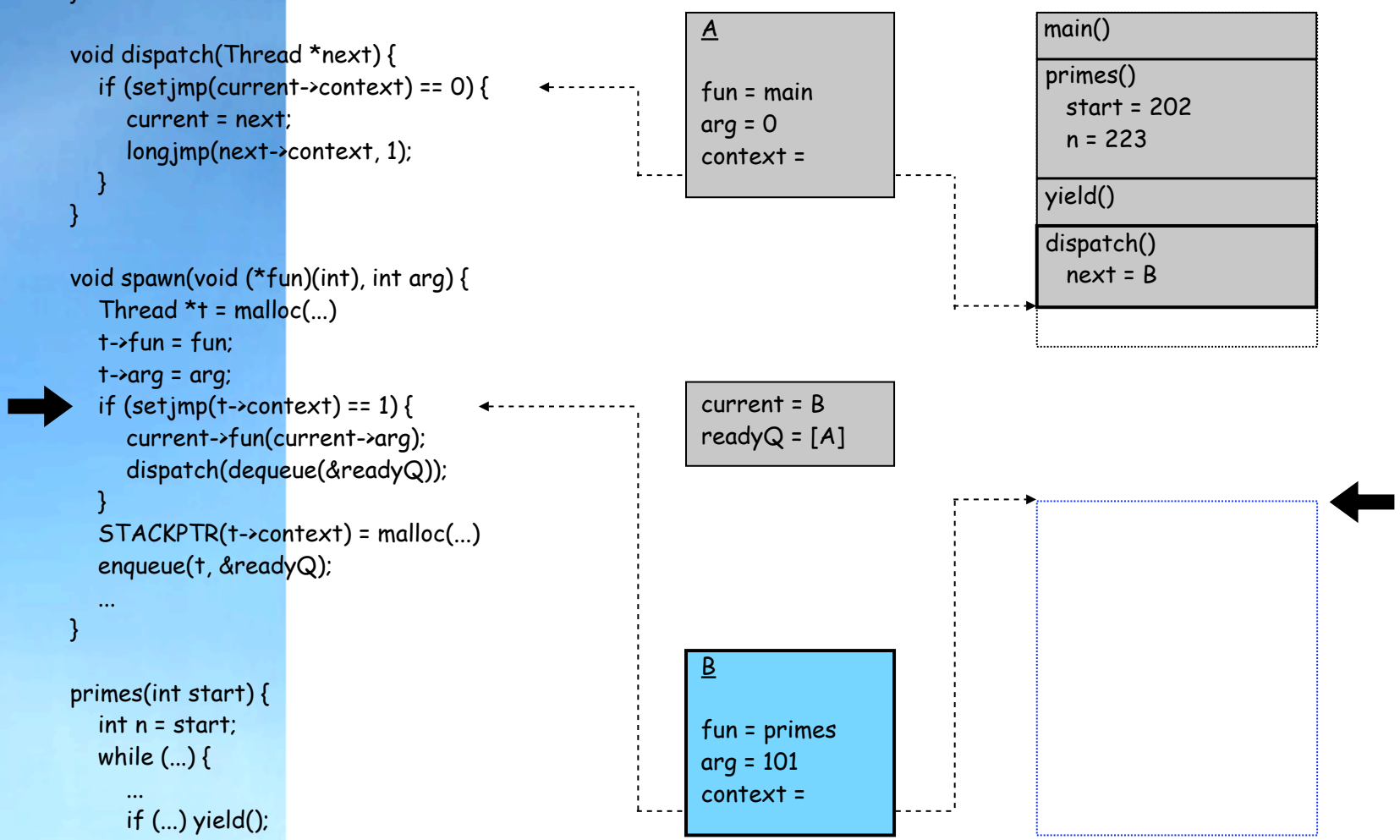
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



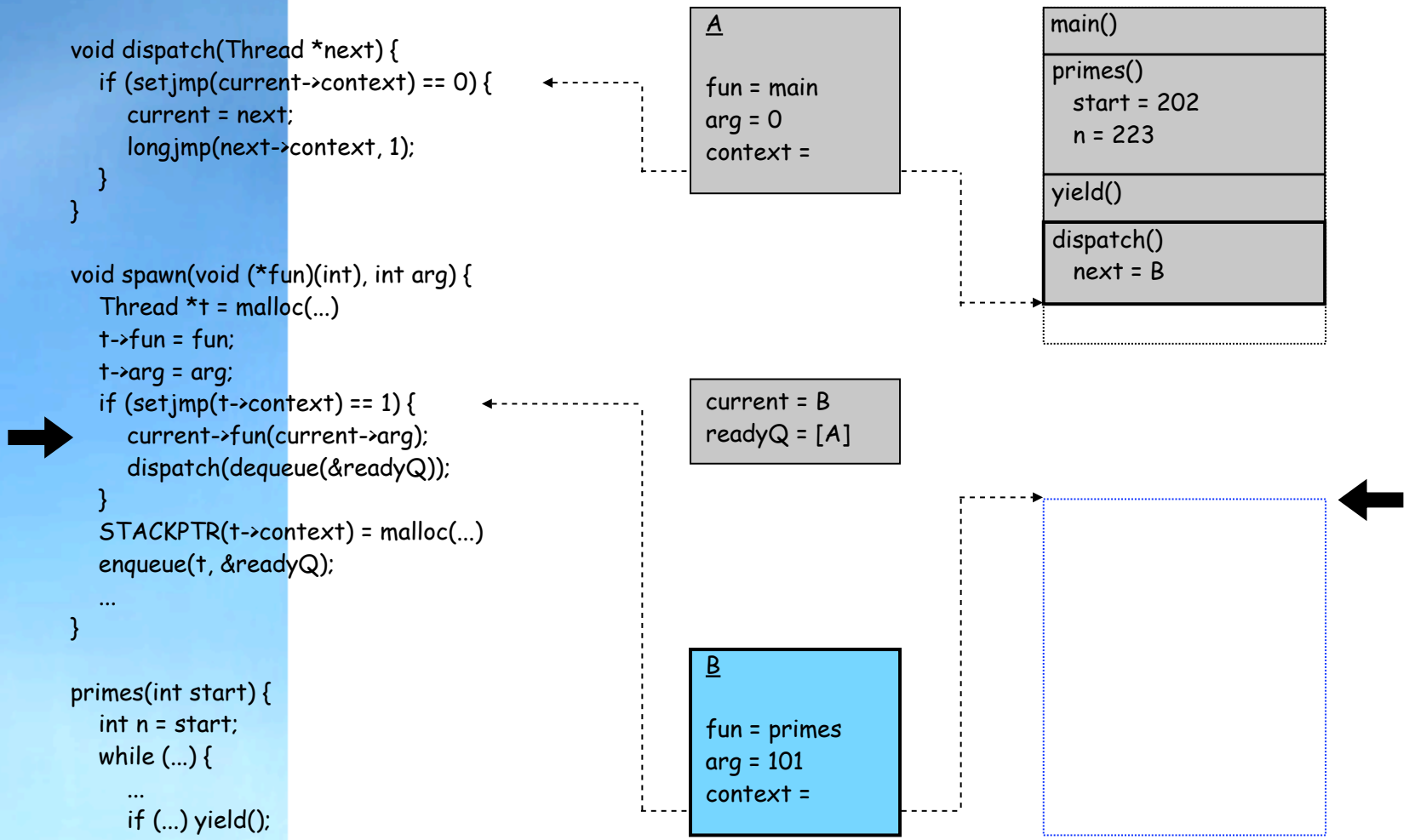
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

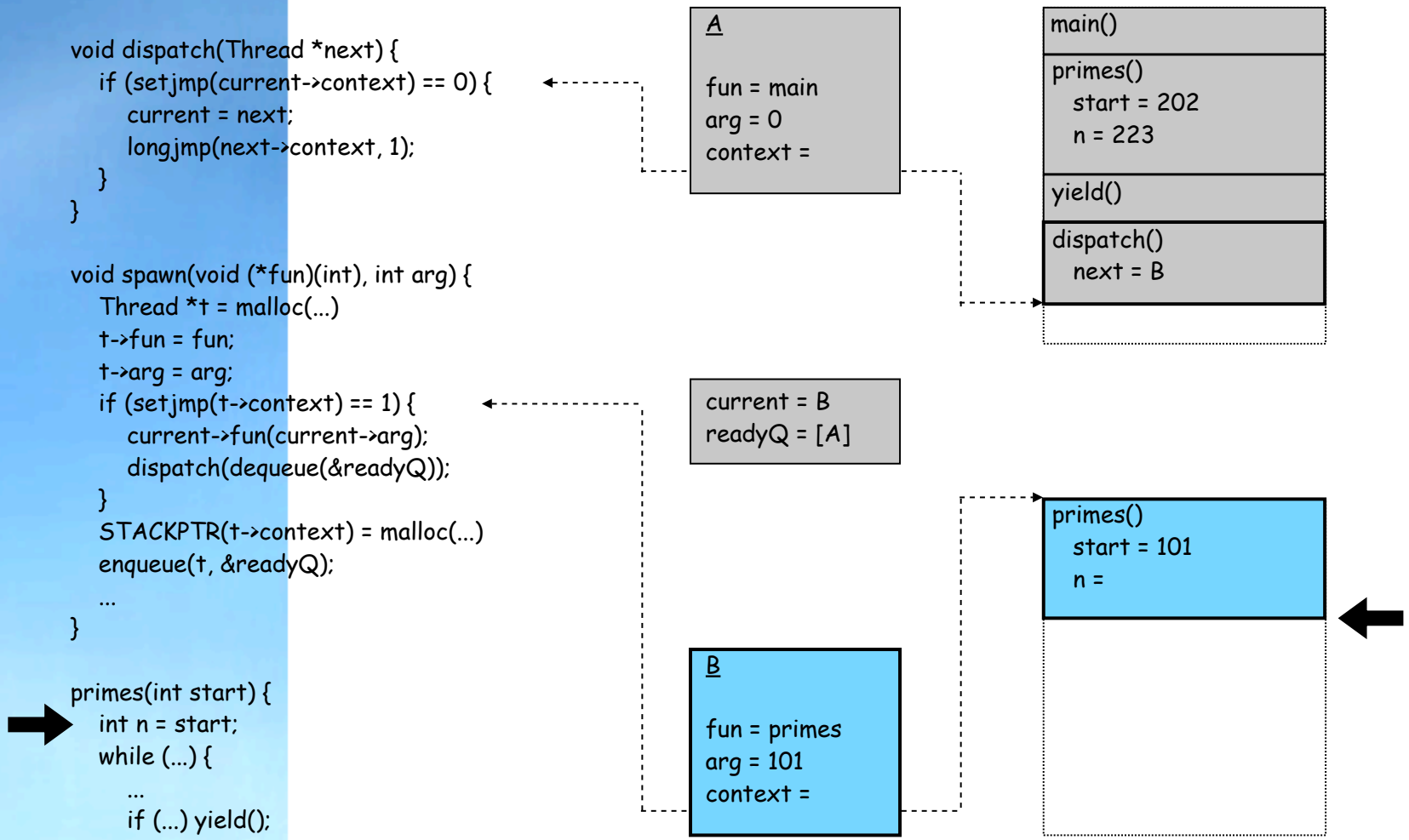


```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```

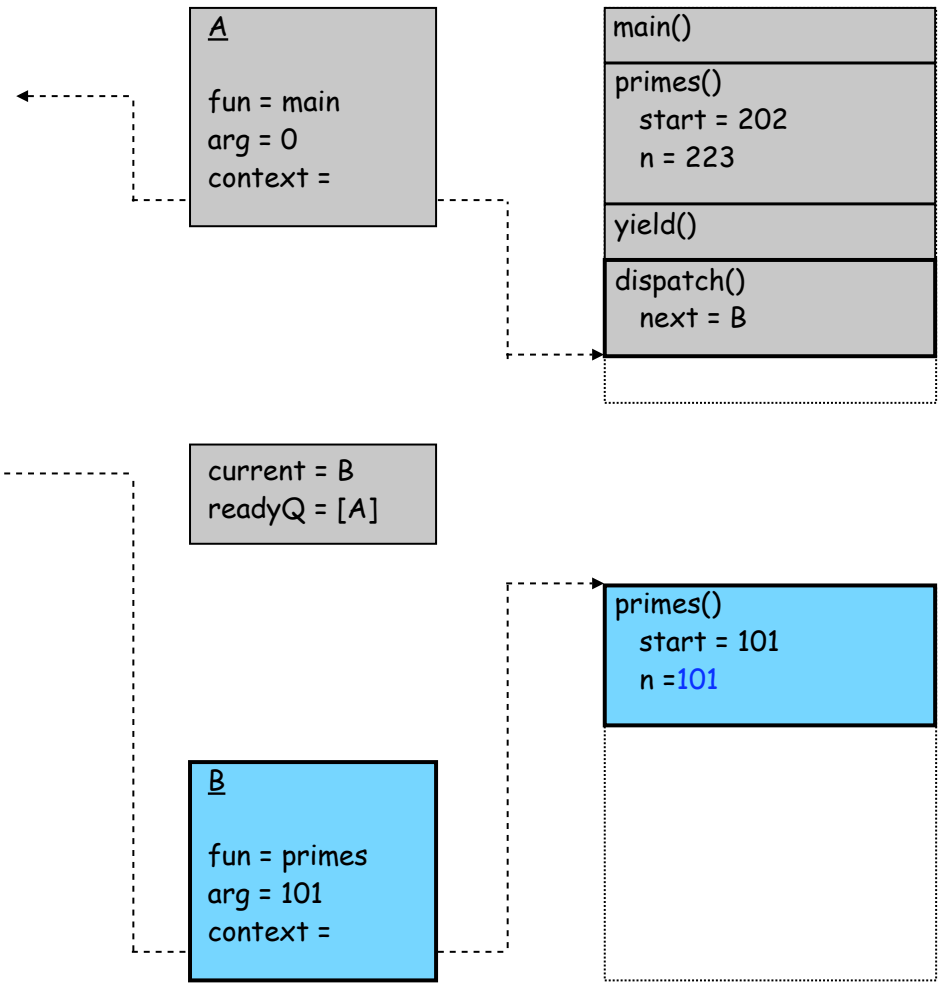


Note that fun and arg can't be found on the stack anymore!

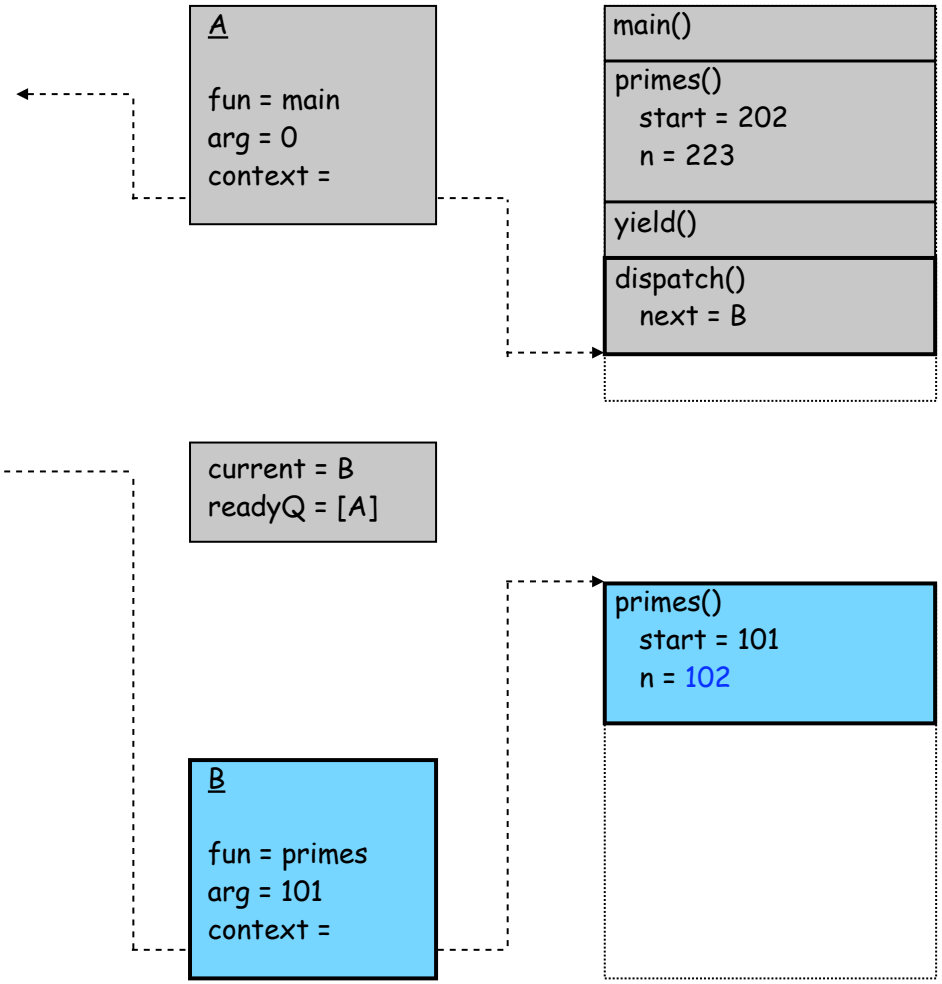
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



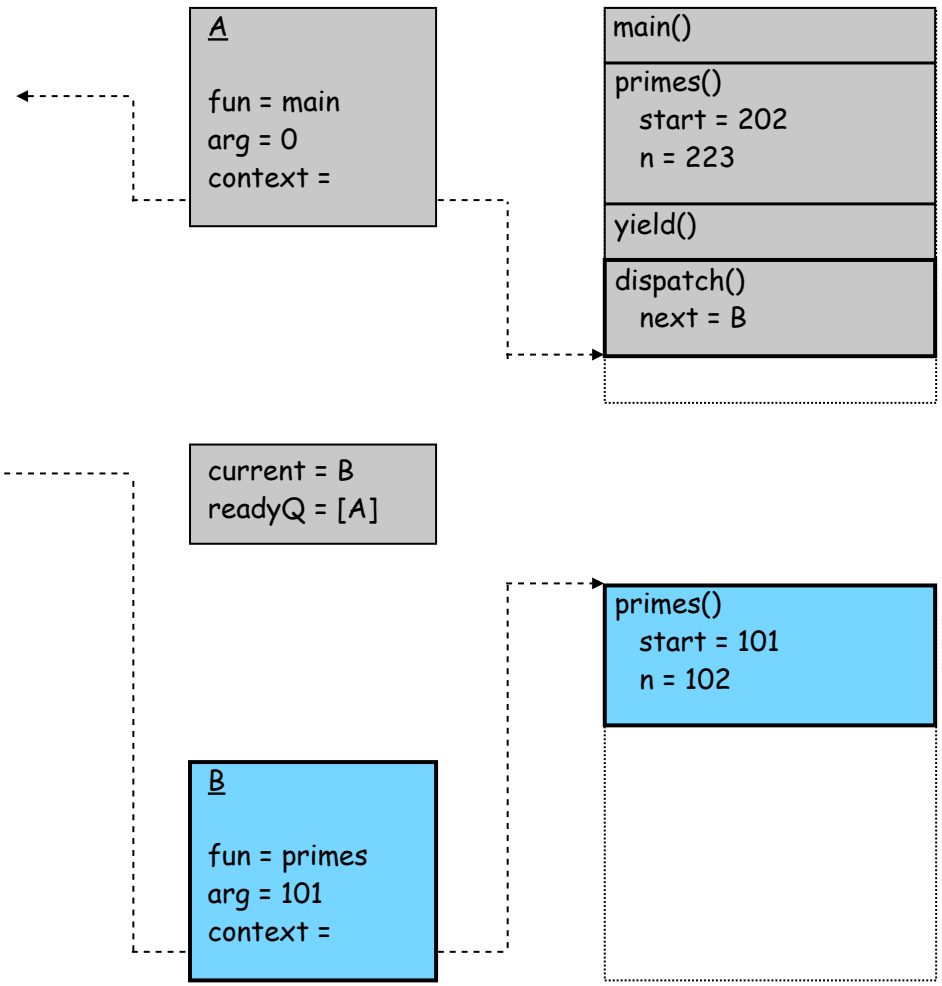
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



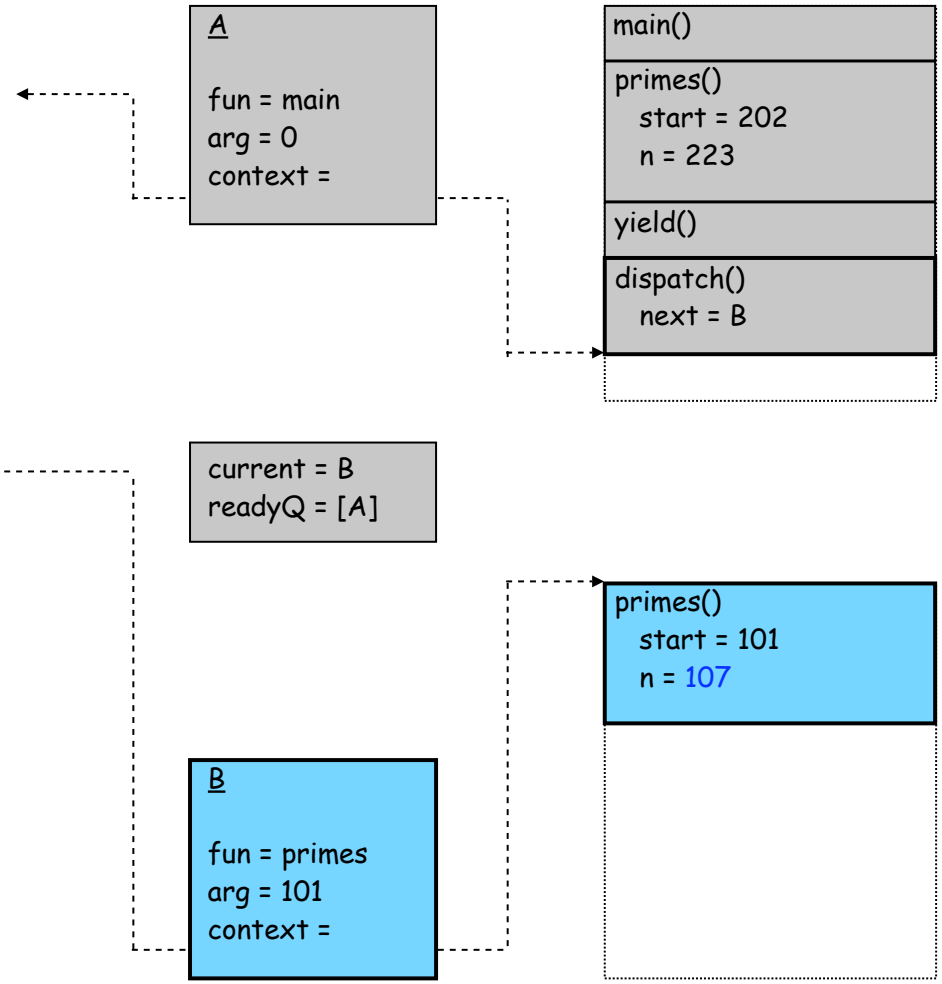
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



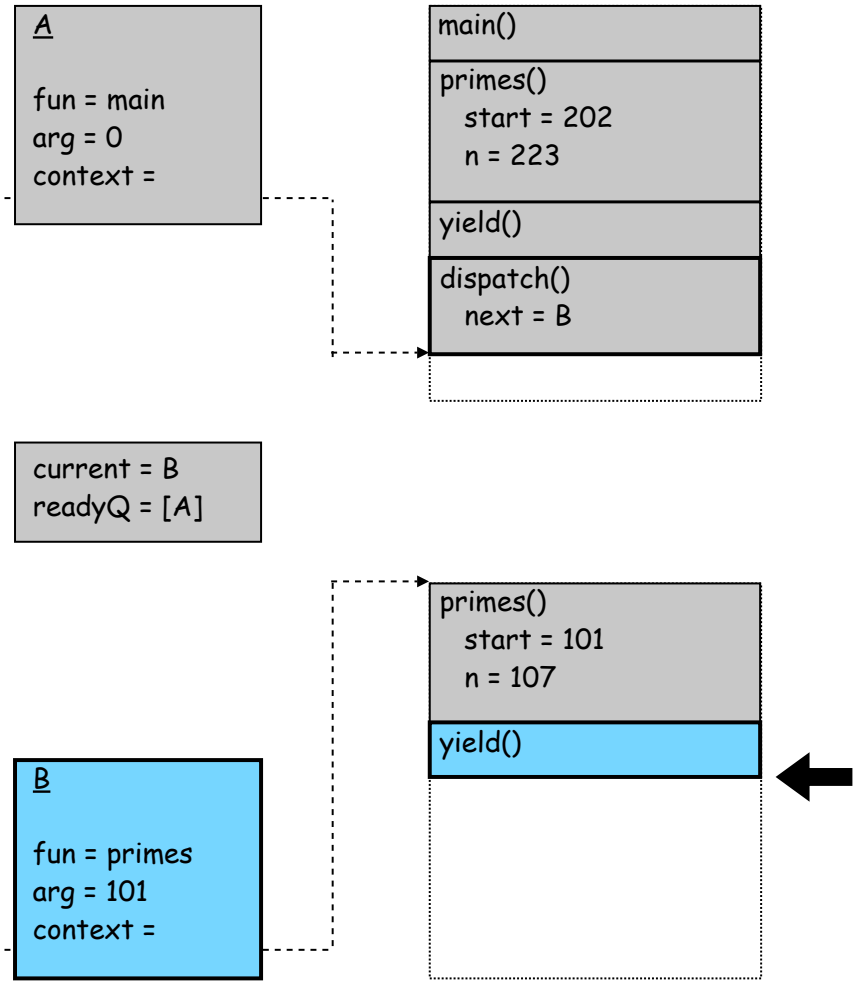
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



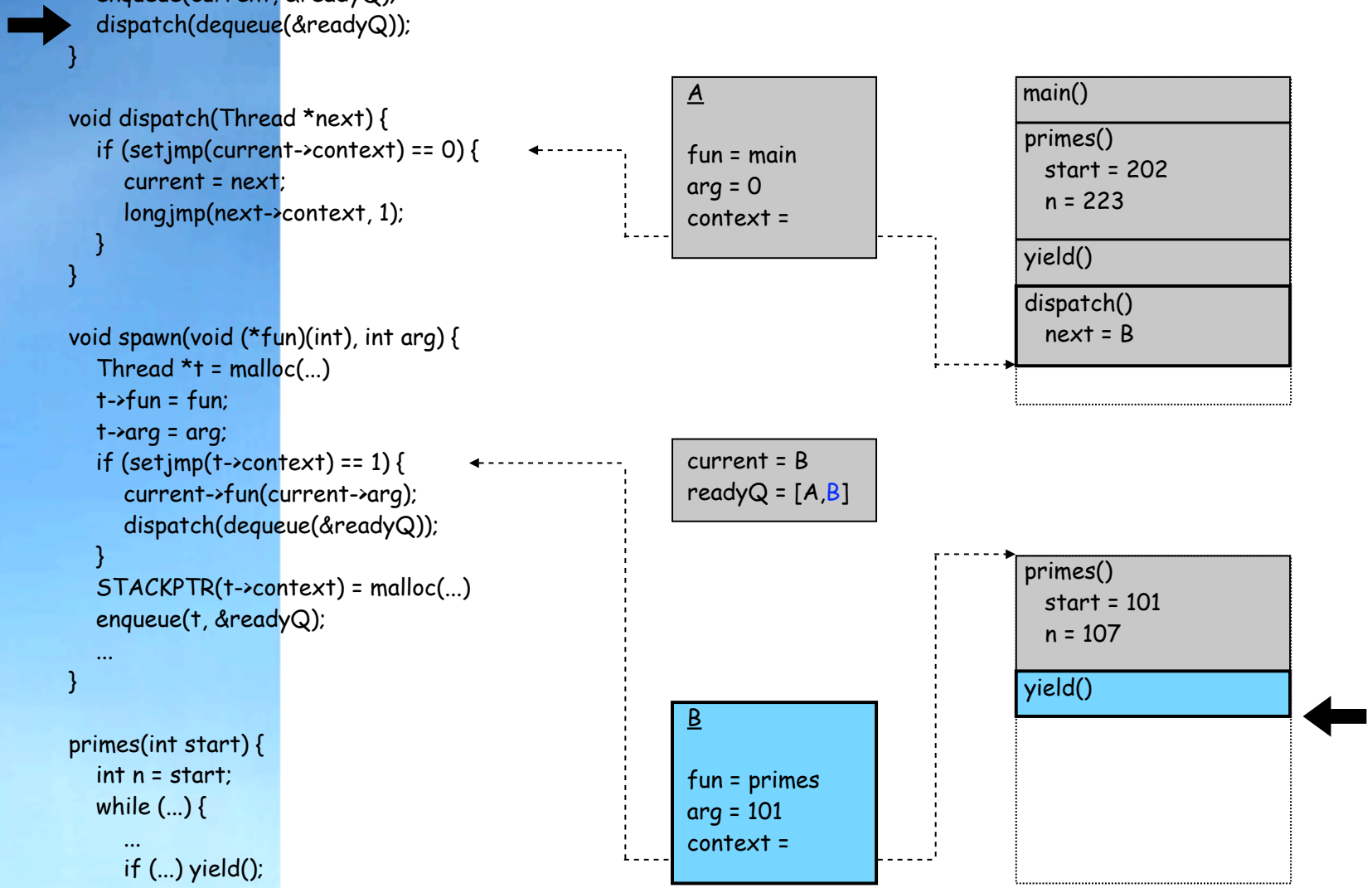
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



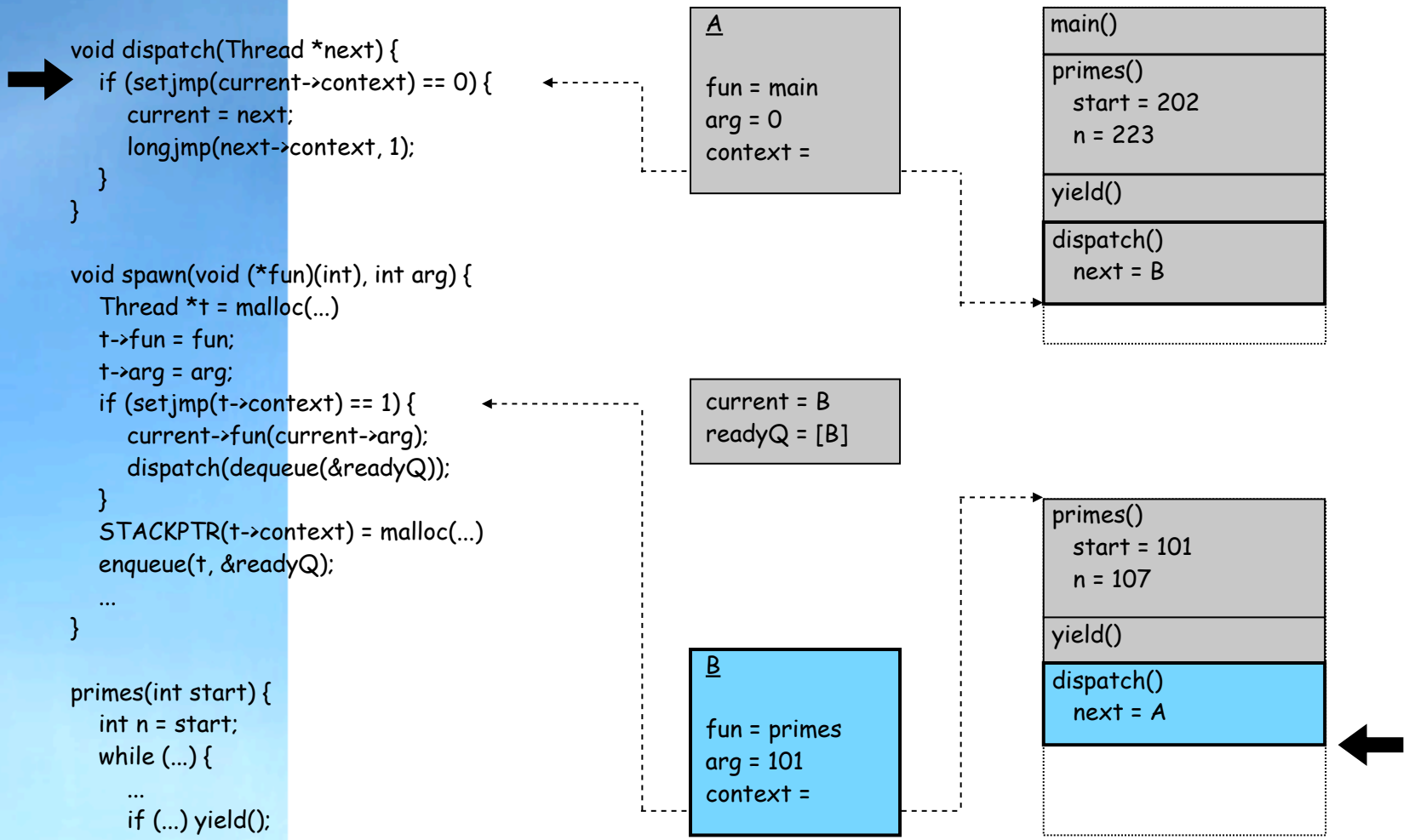
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



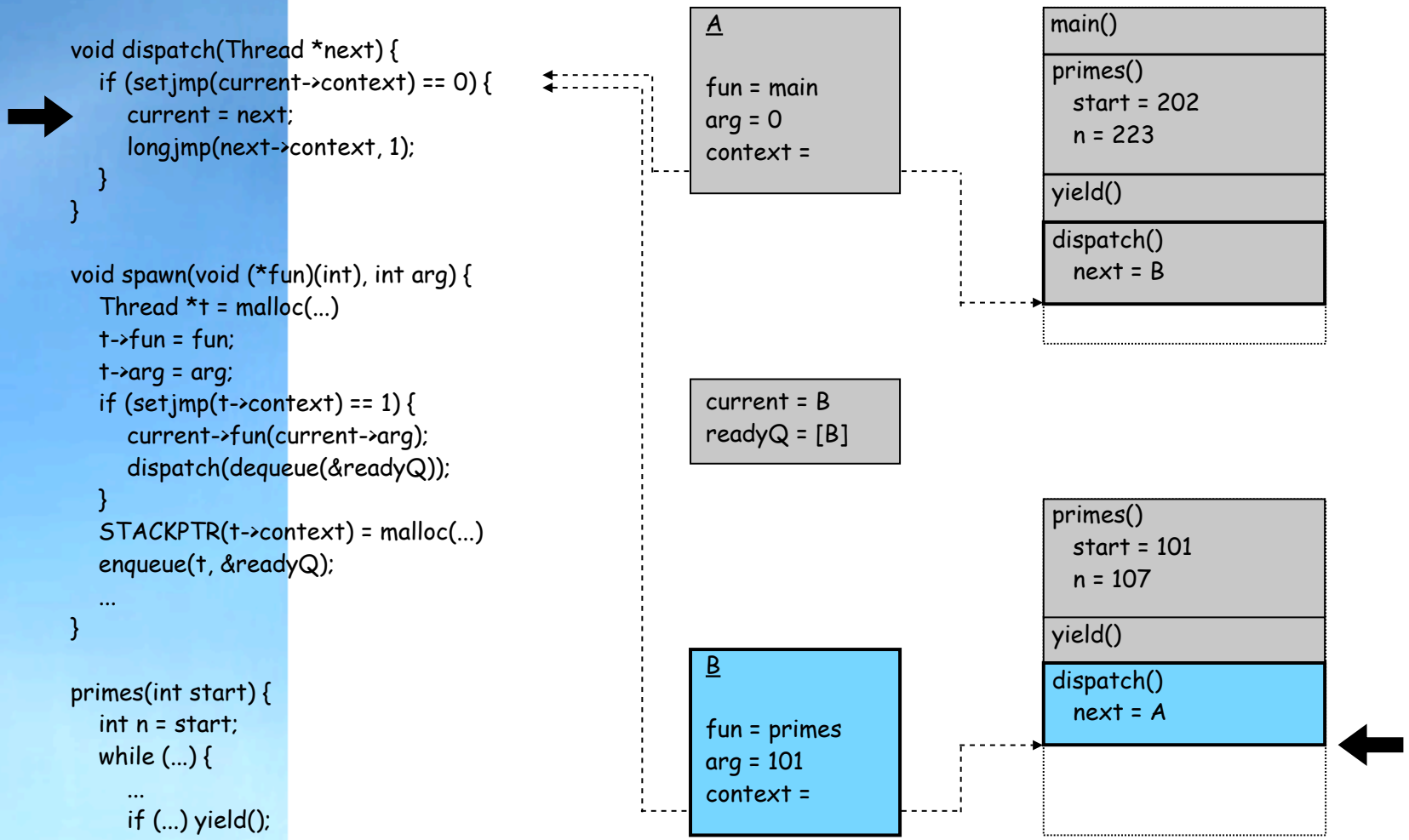
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



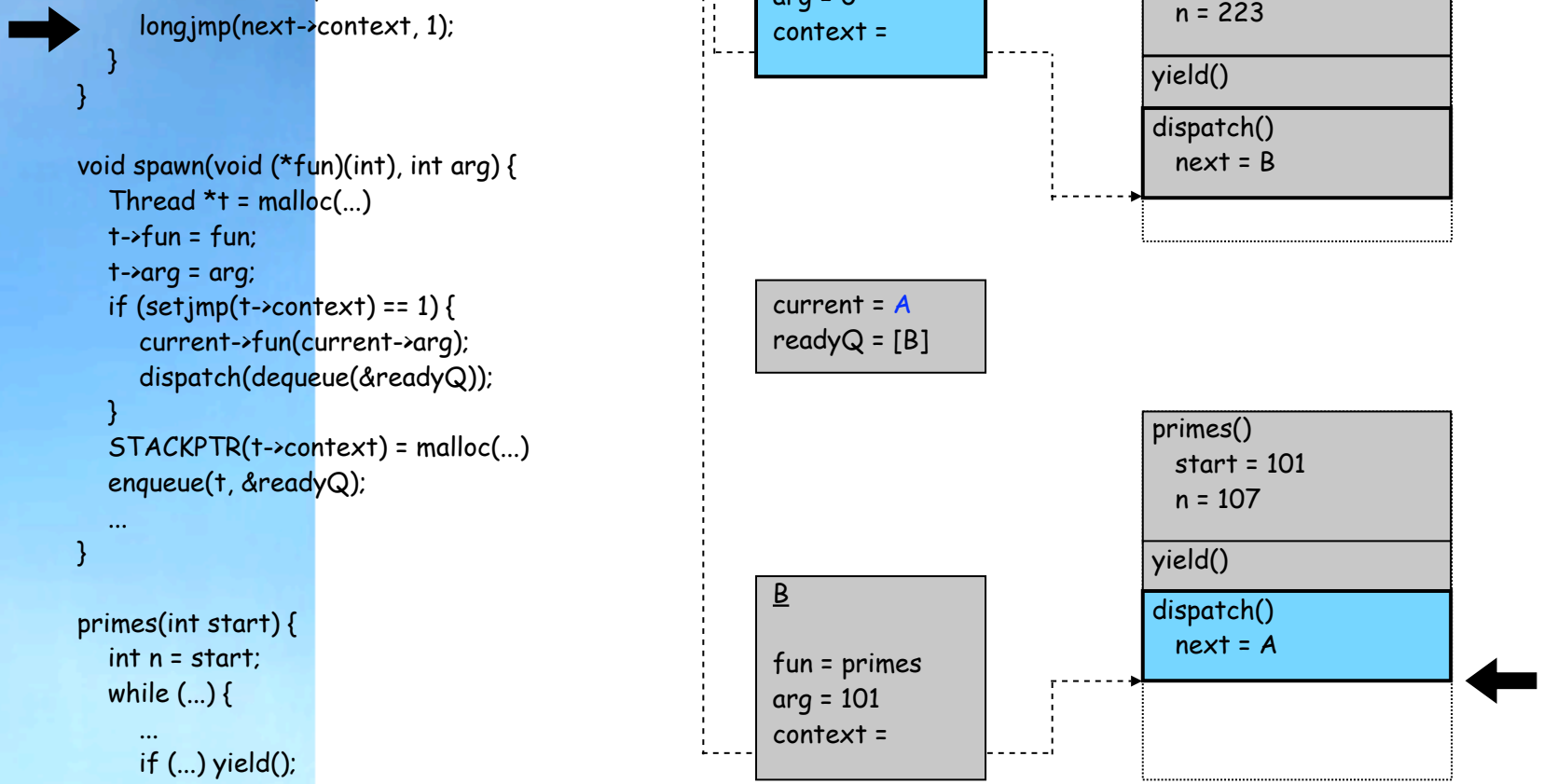
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



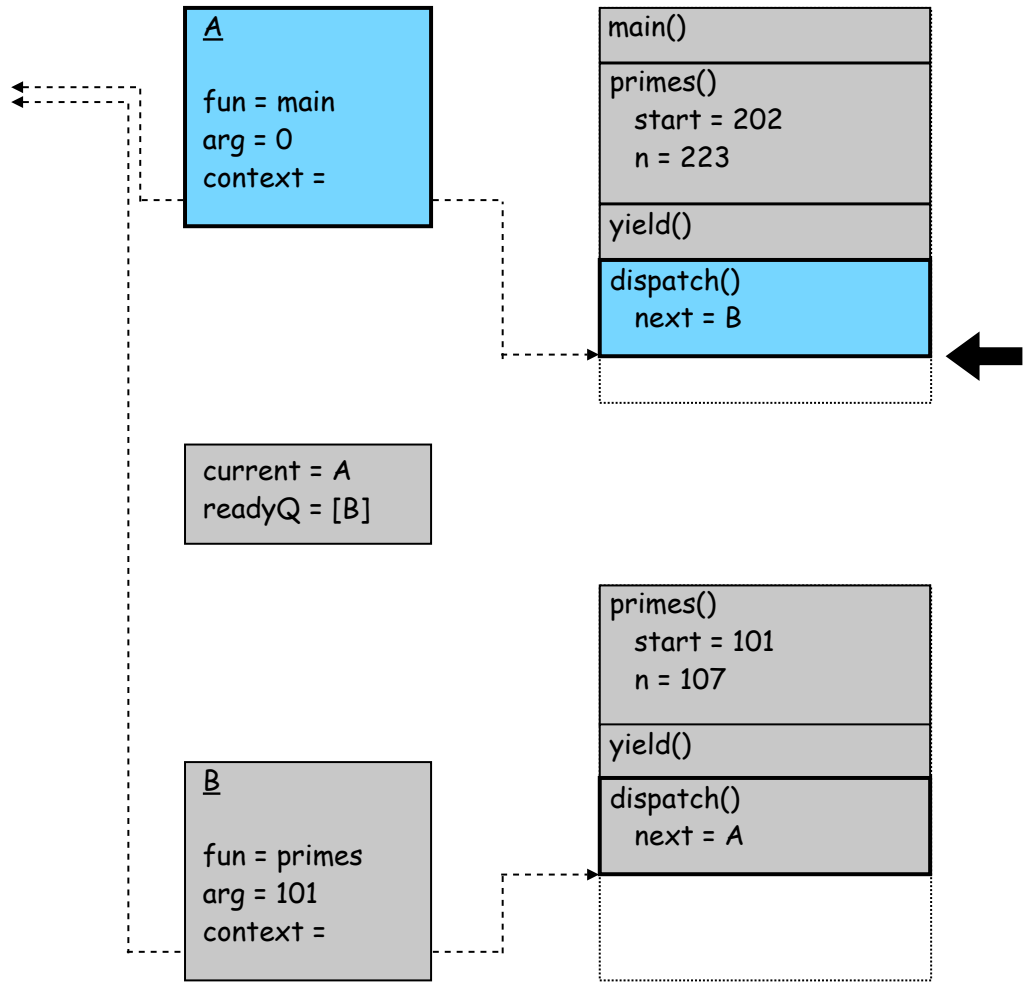
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



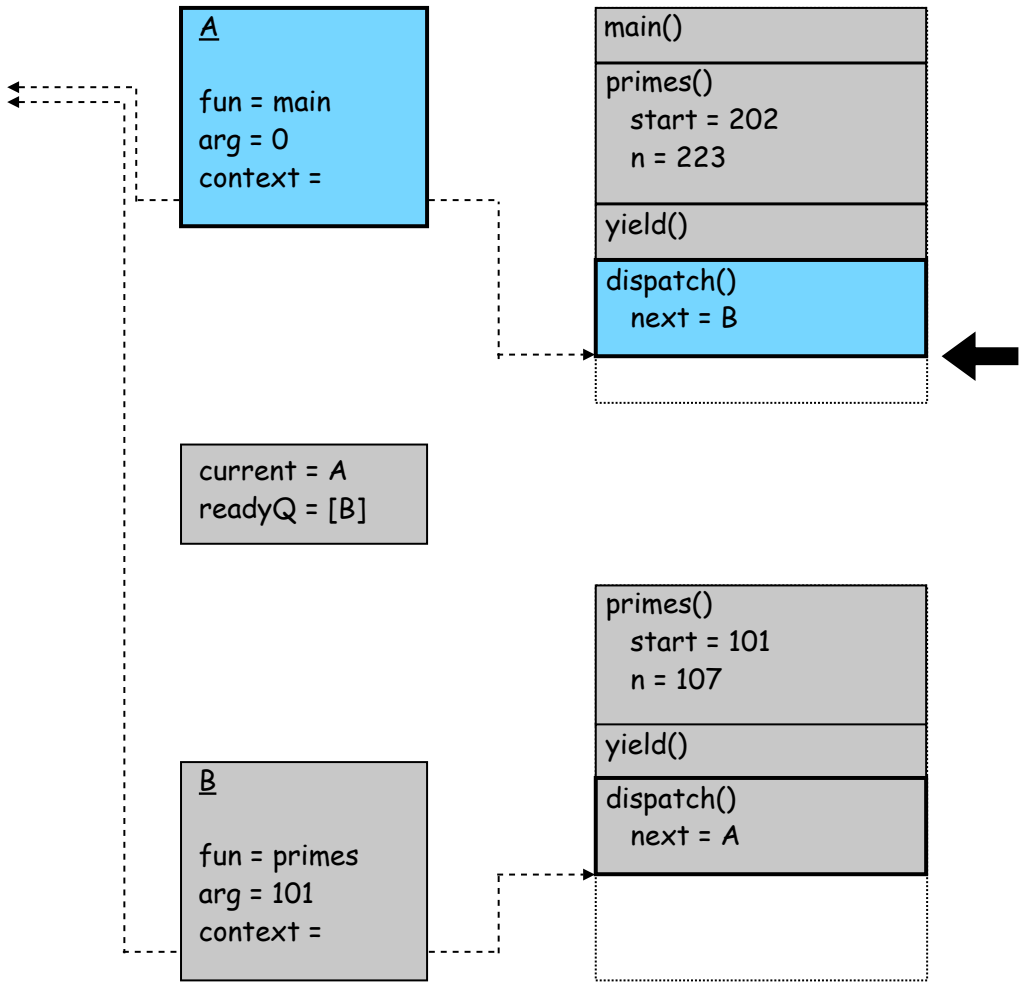
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



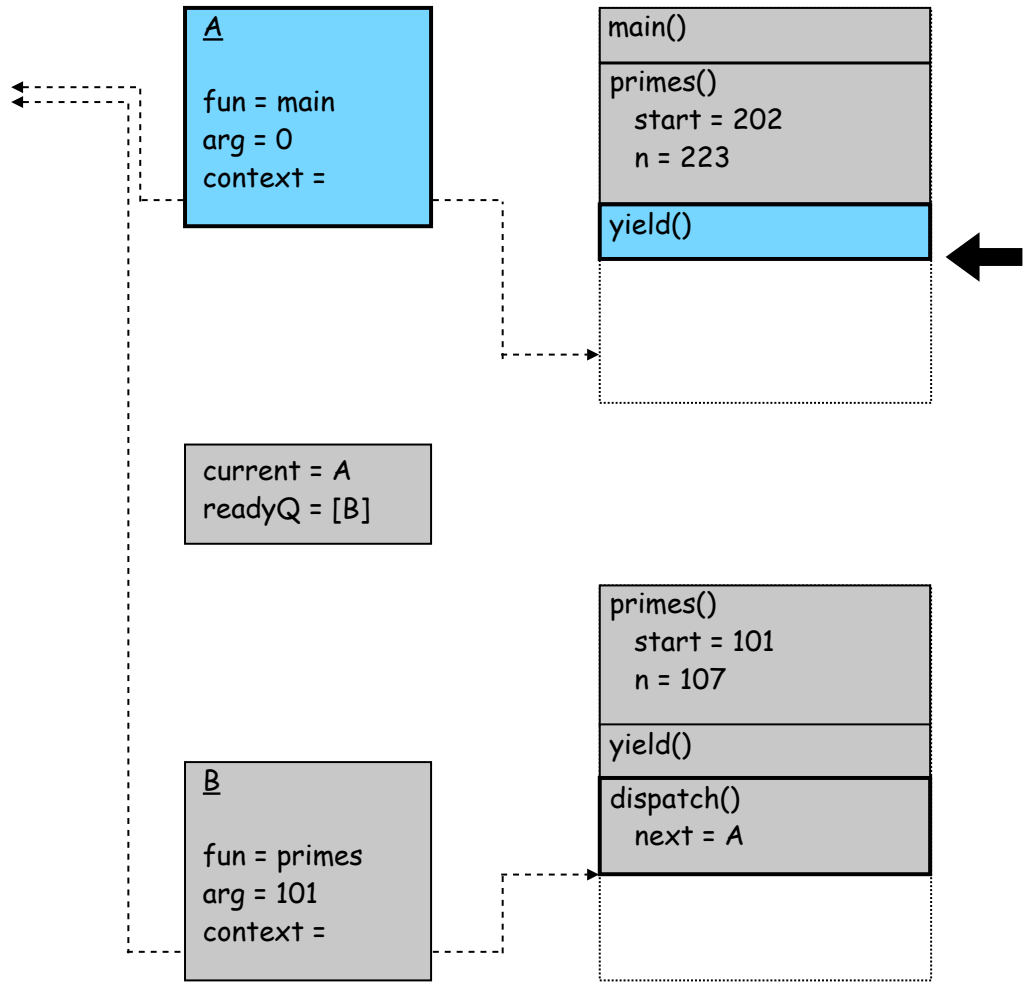
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



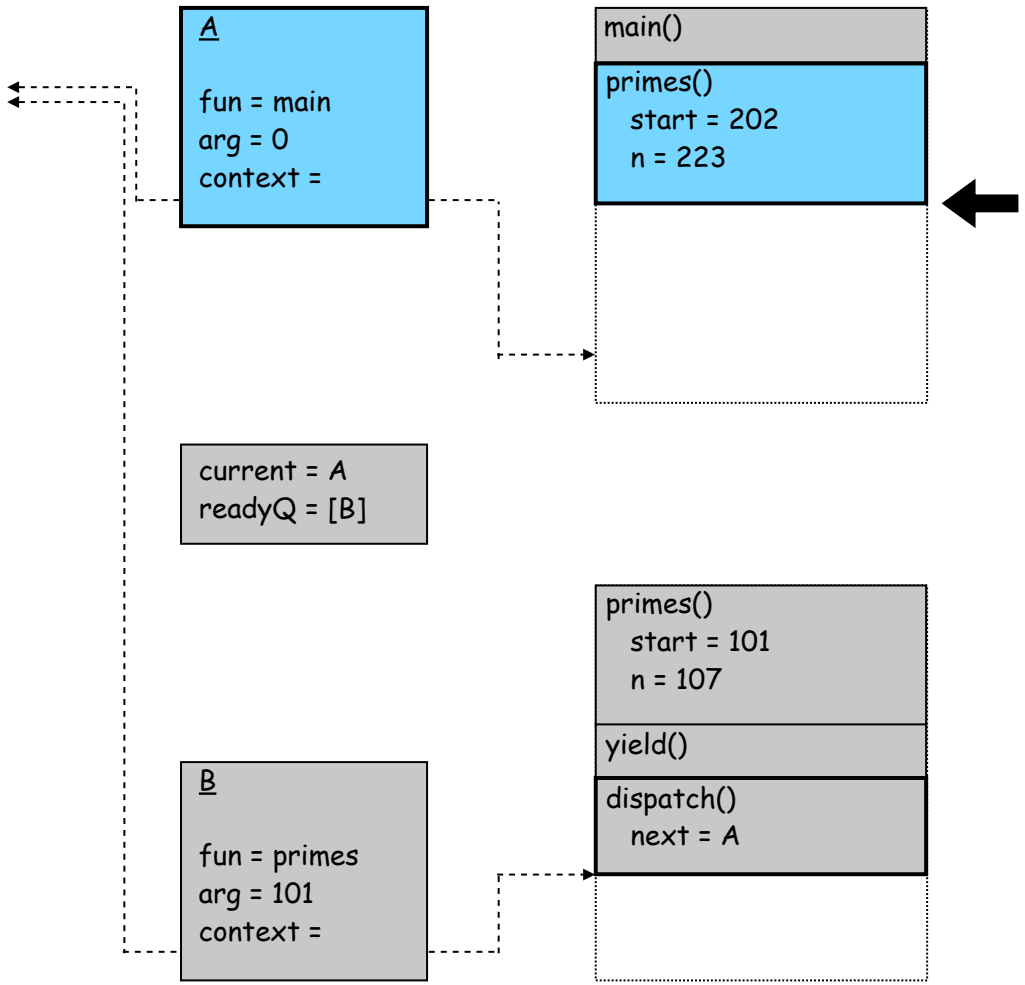
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



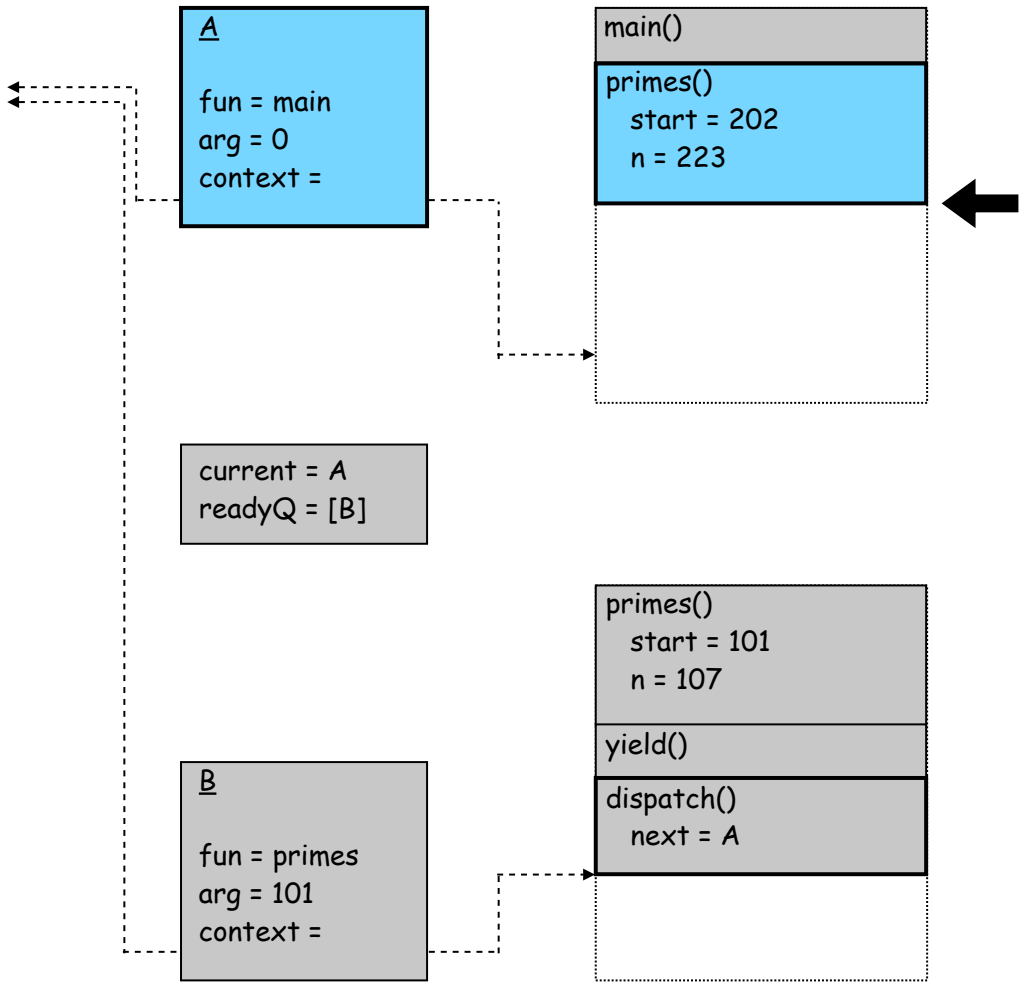
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



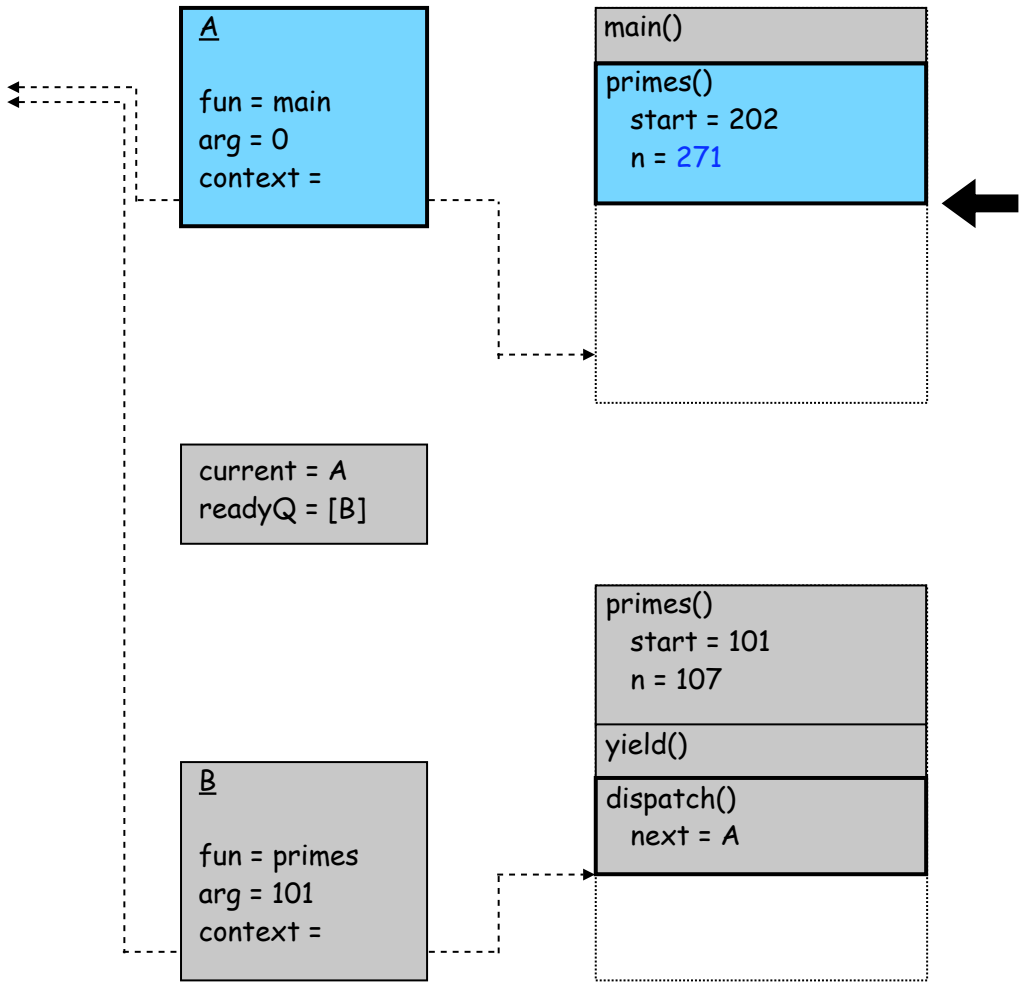
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



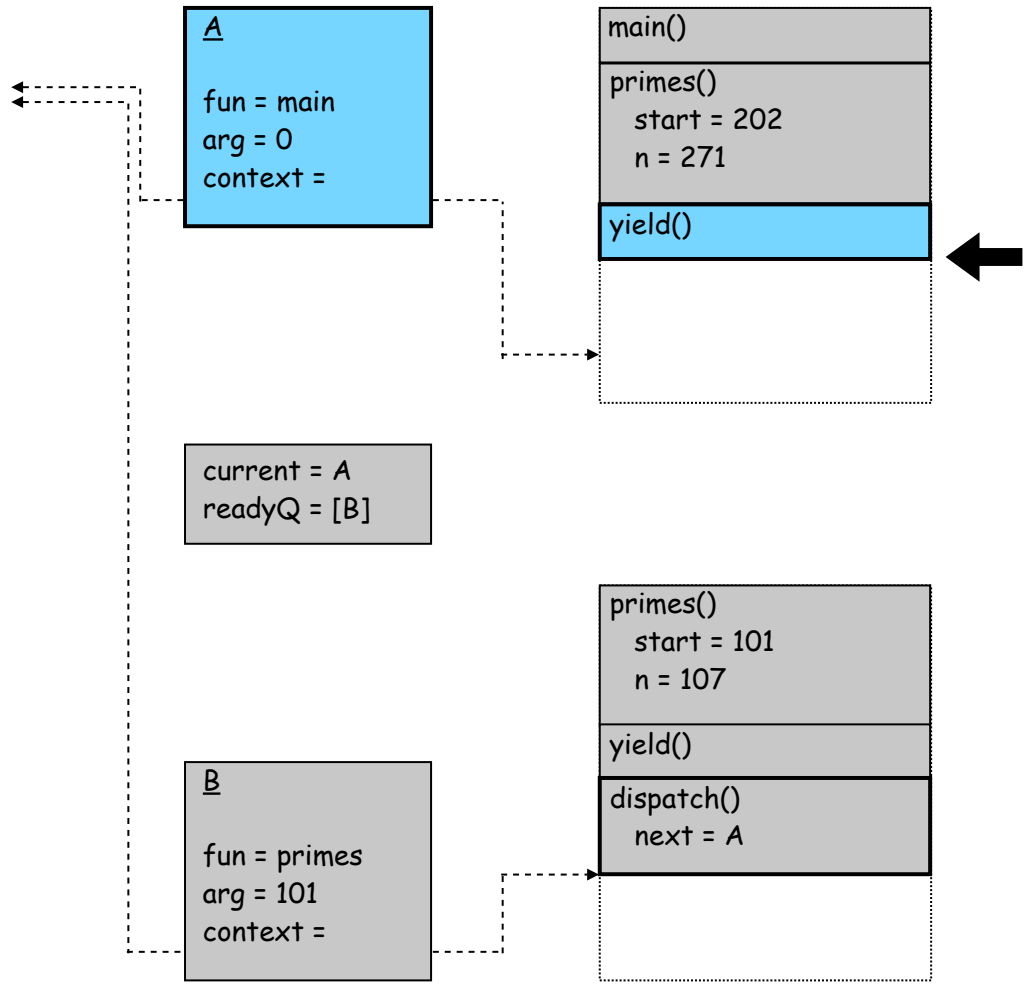
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



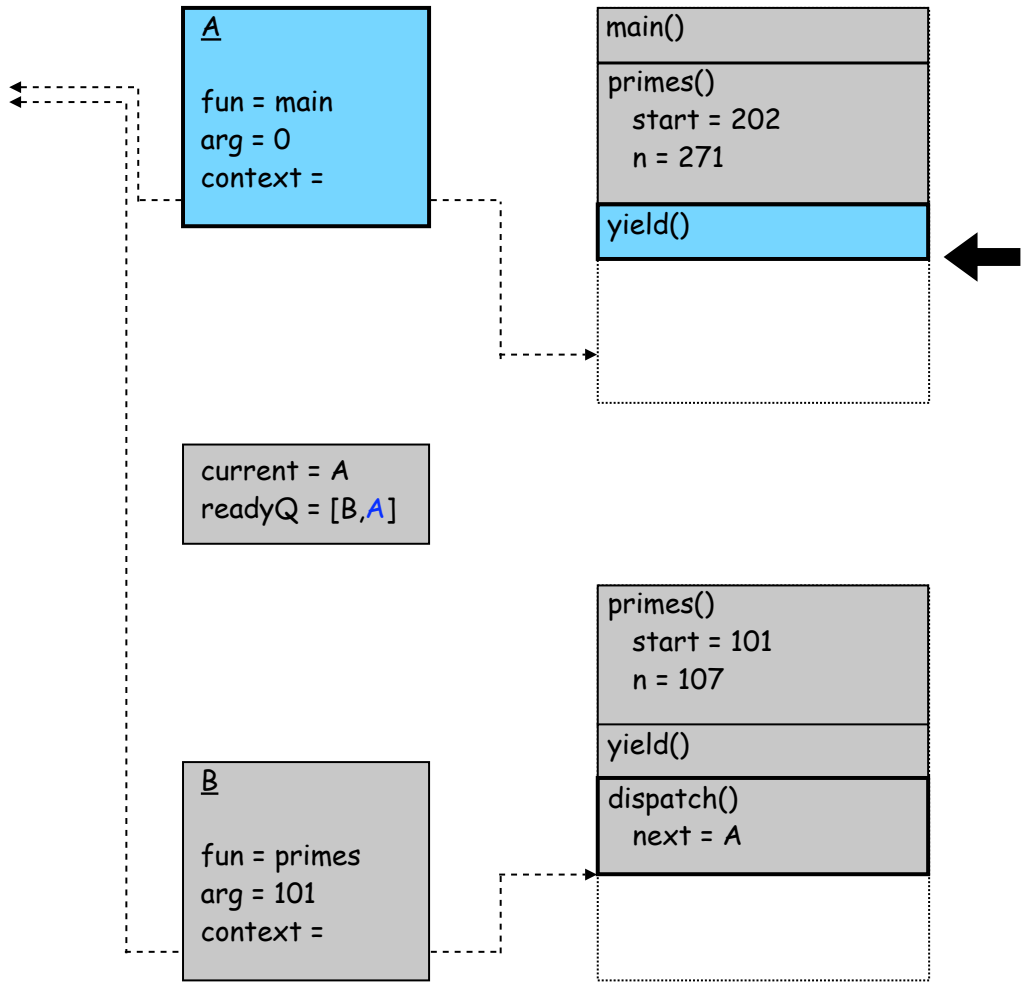
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



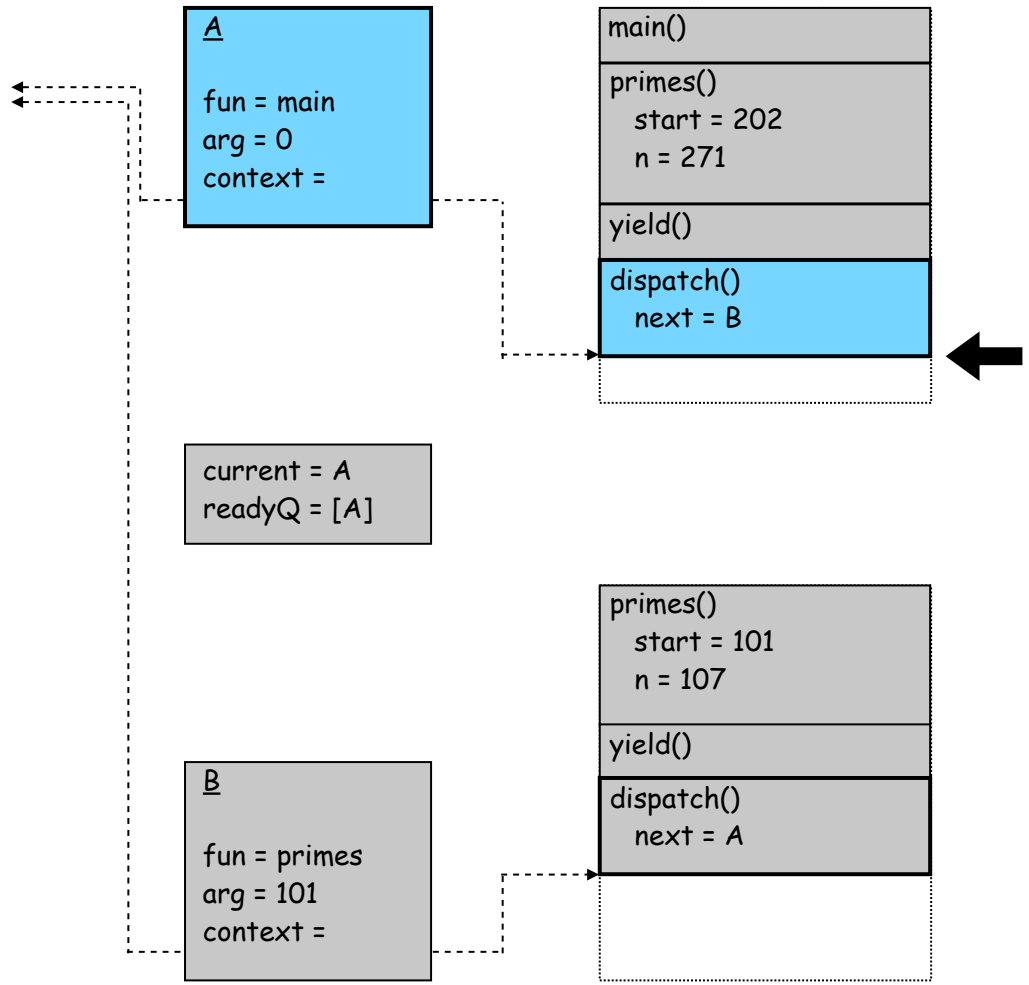
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



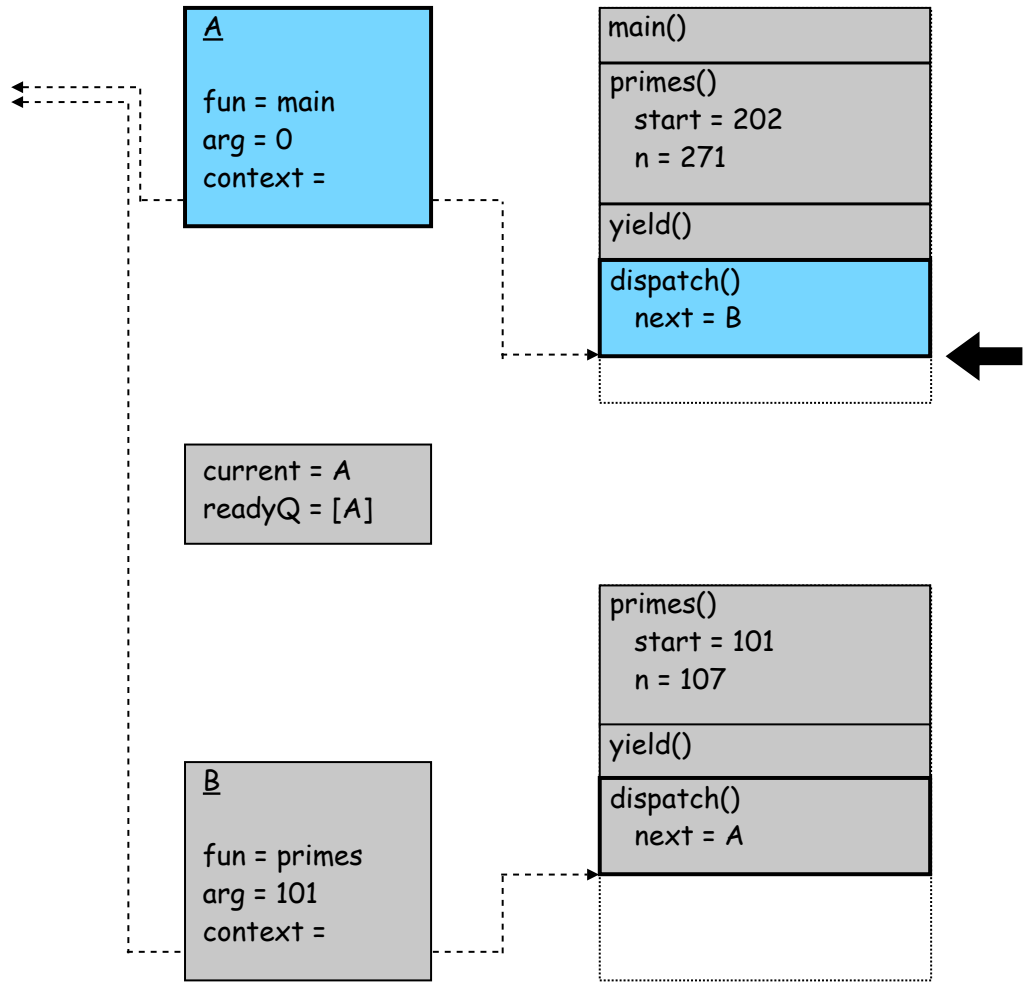
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



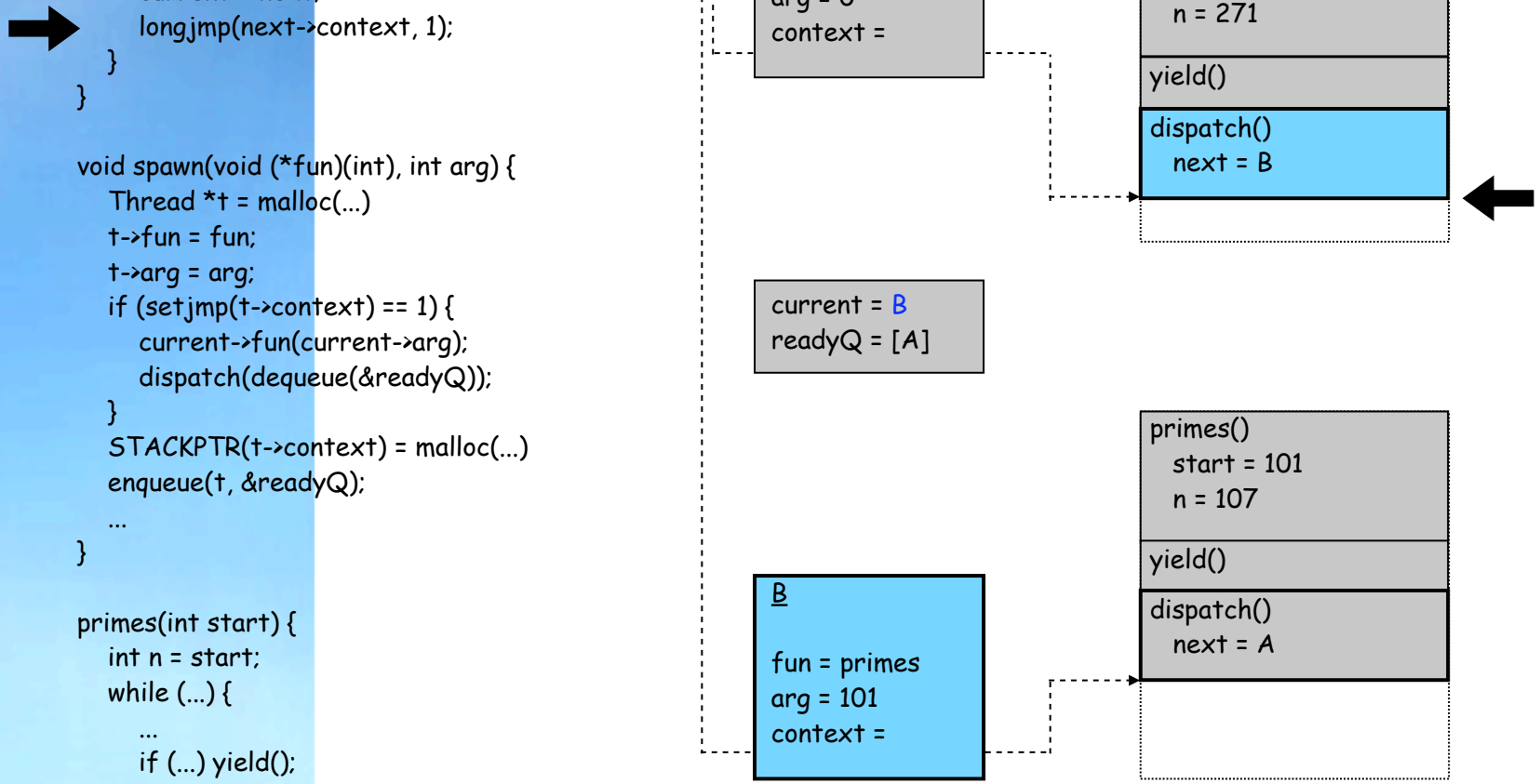
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



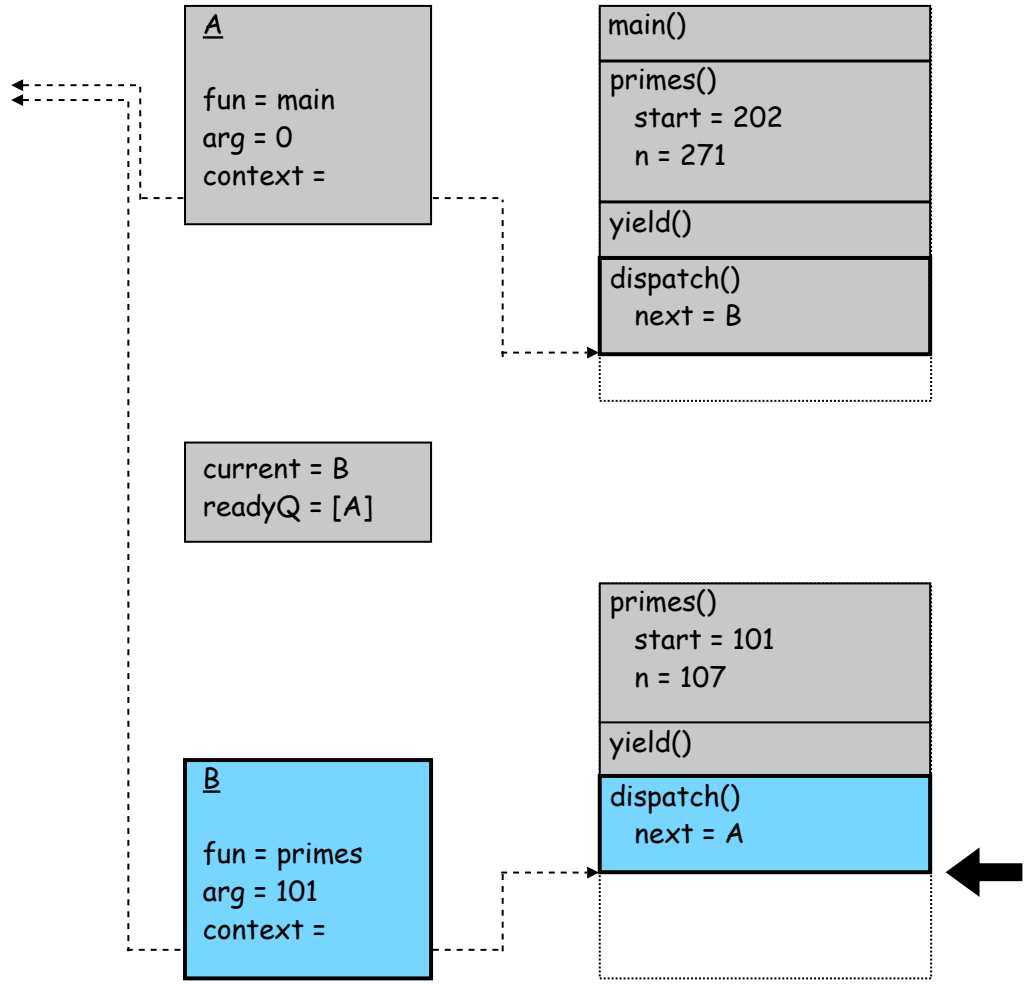
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



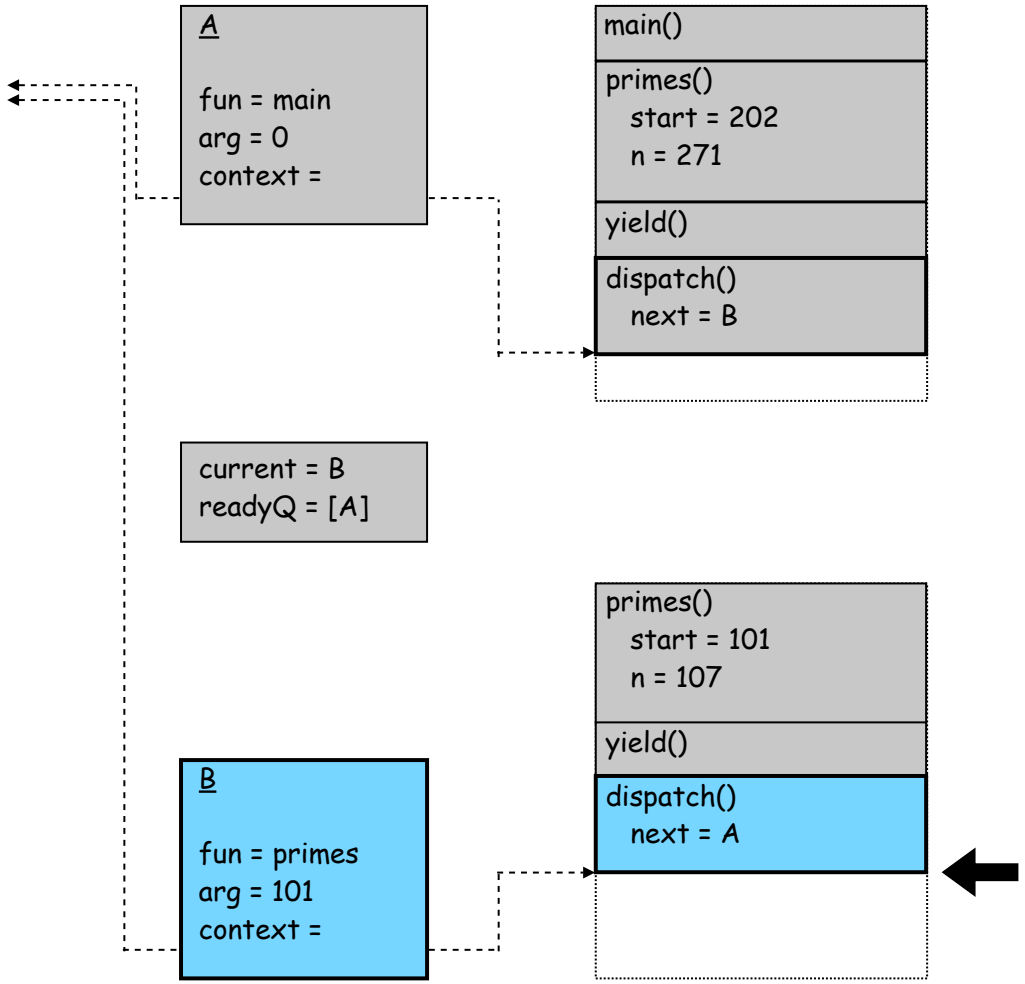
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



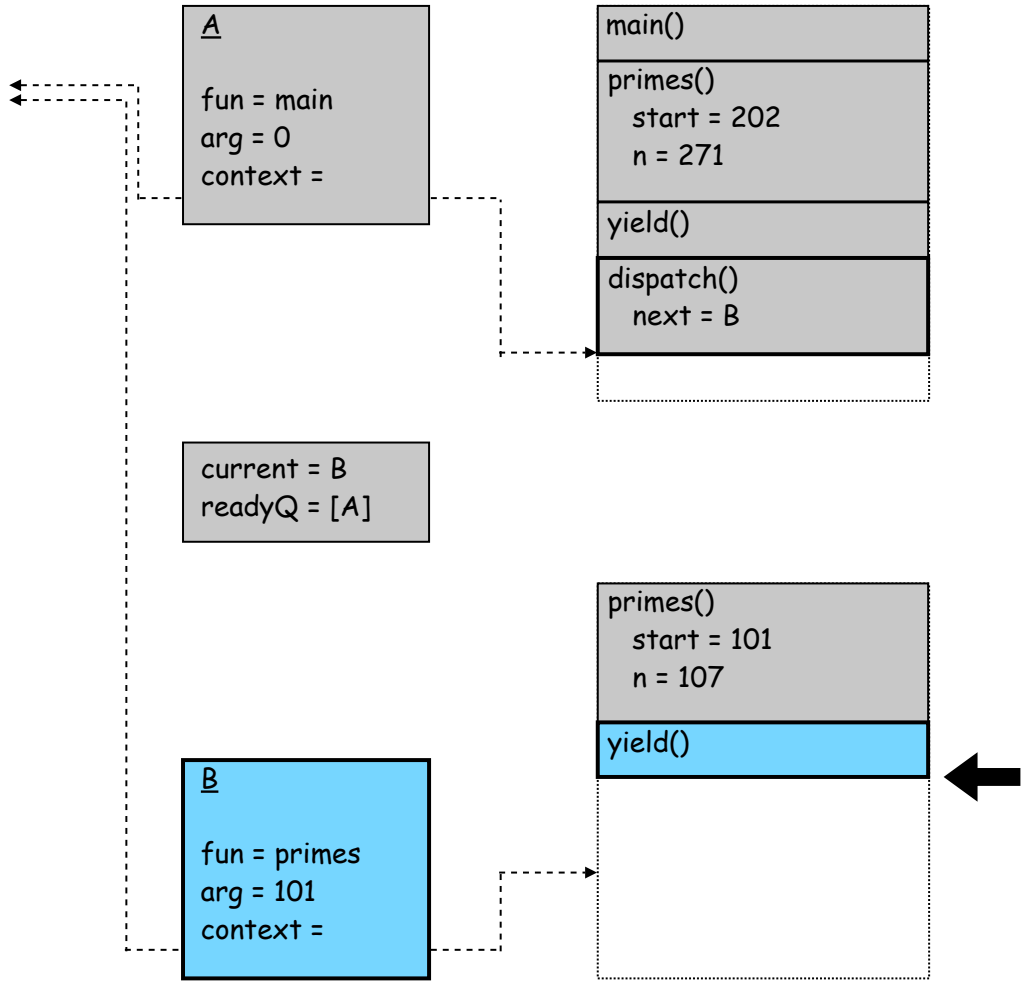
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



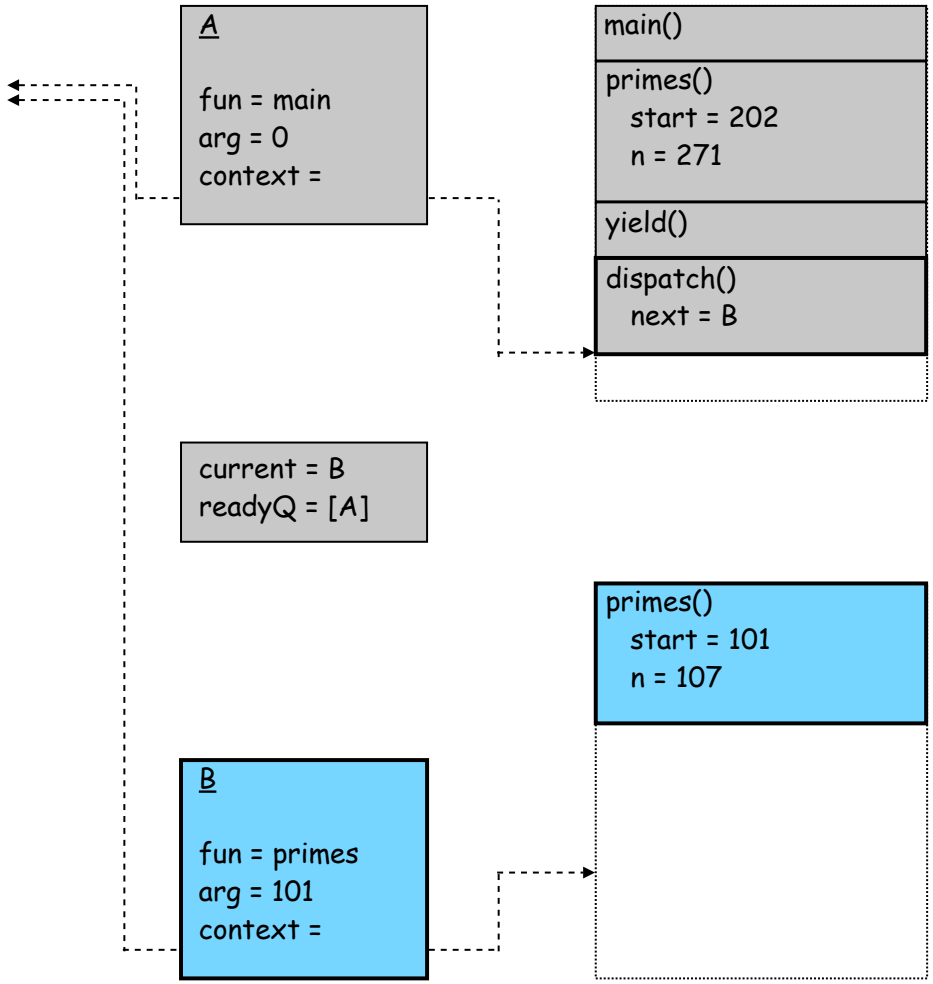
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



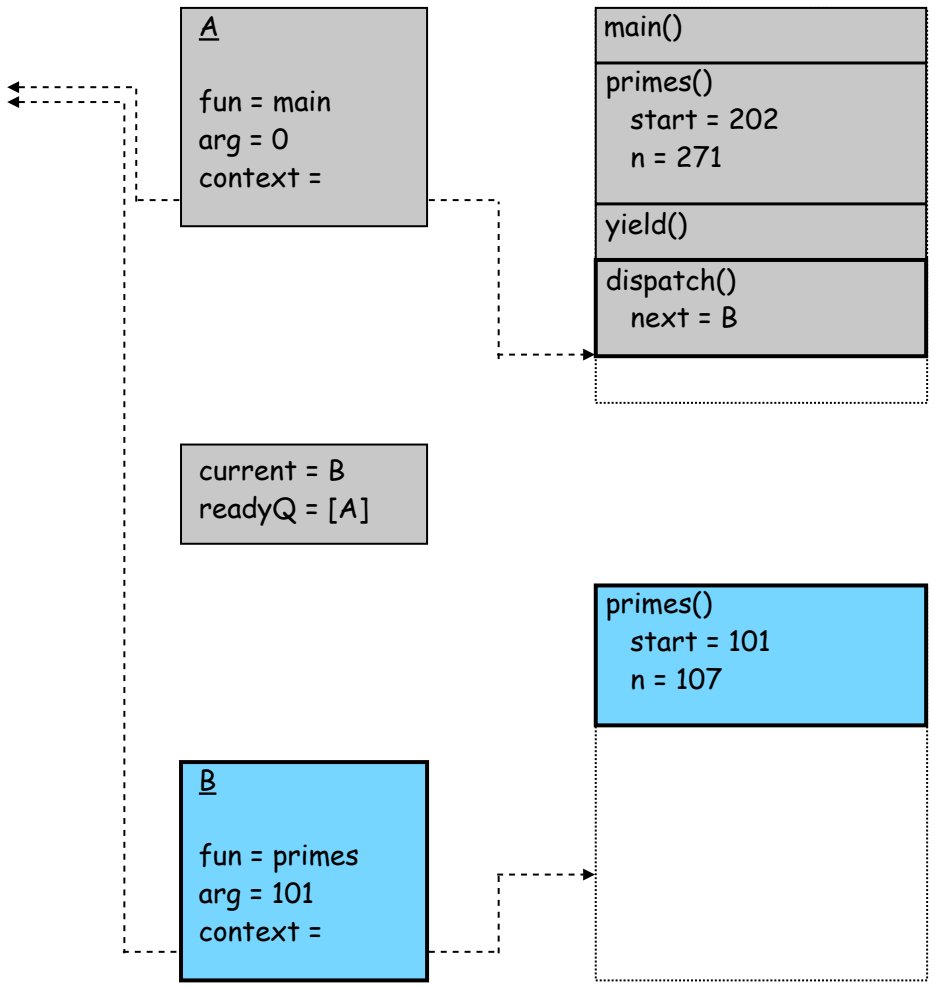
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



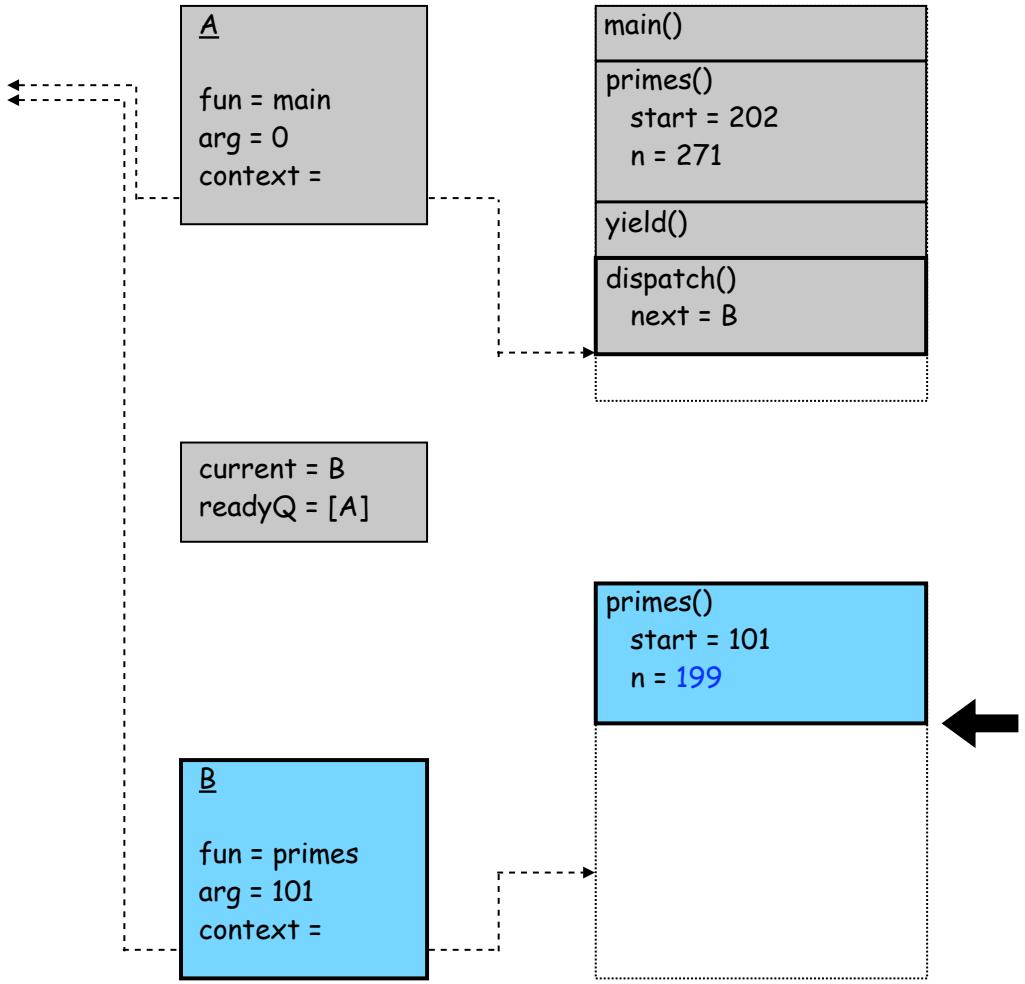
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



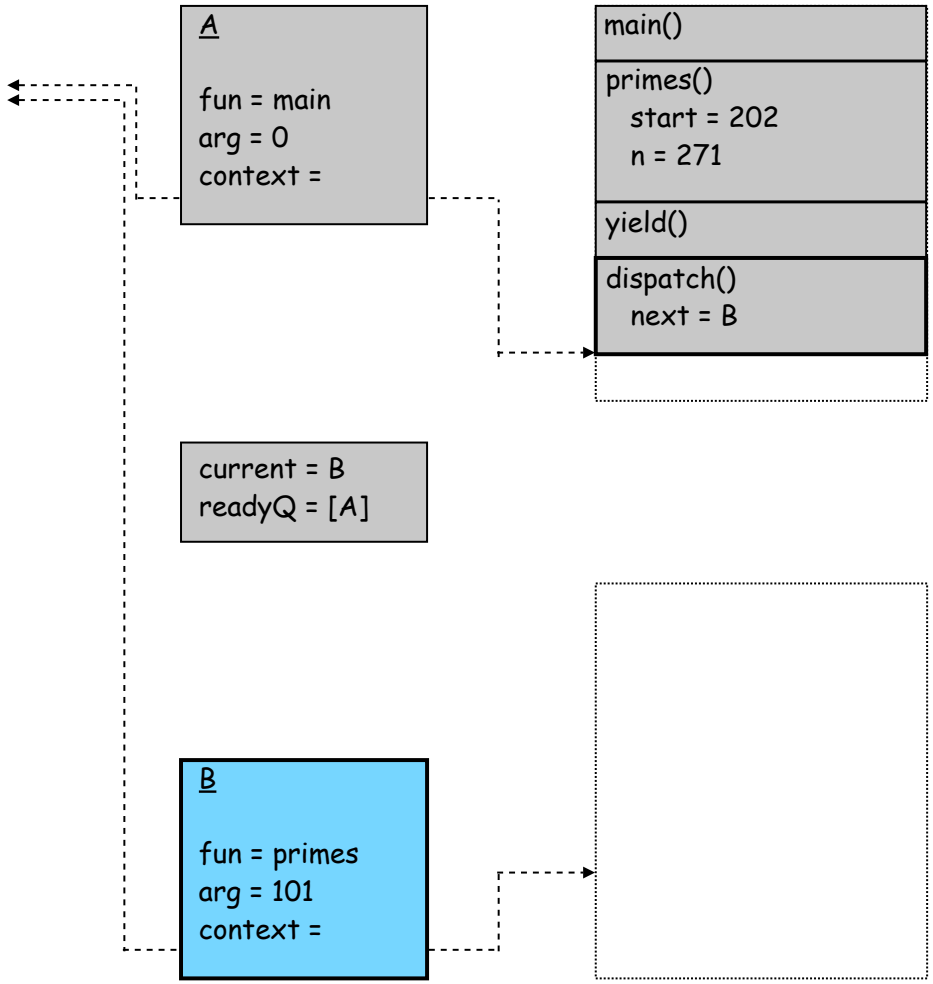
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



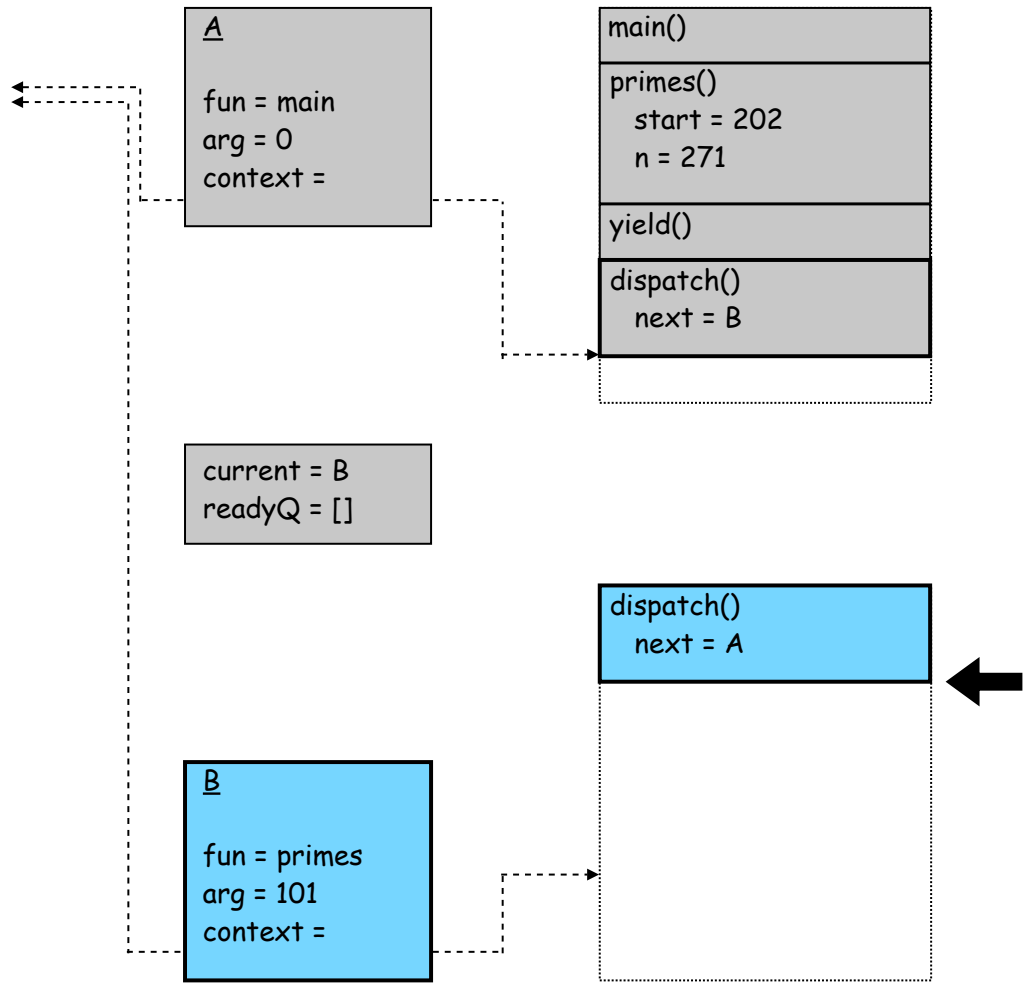
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
}
```



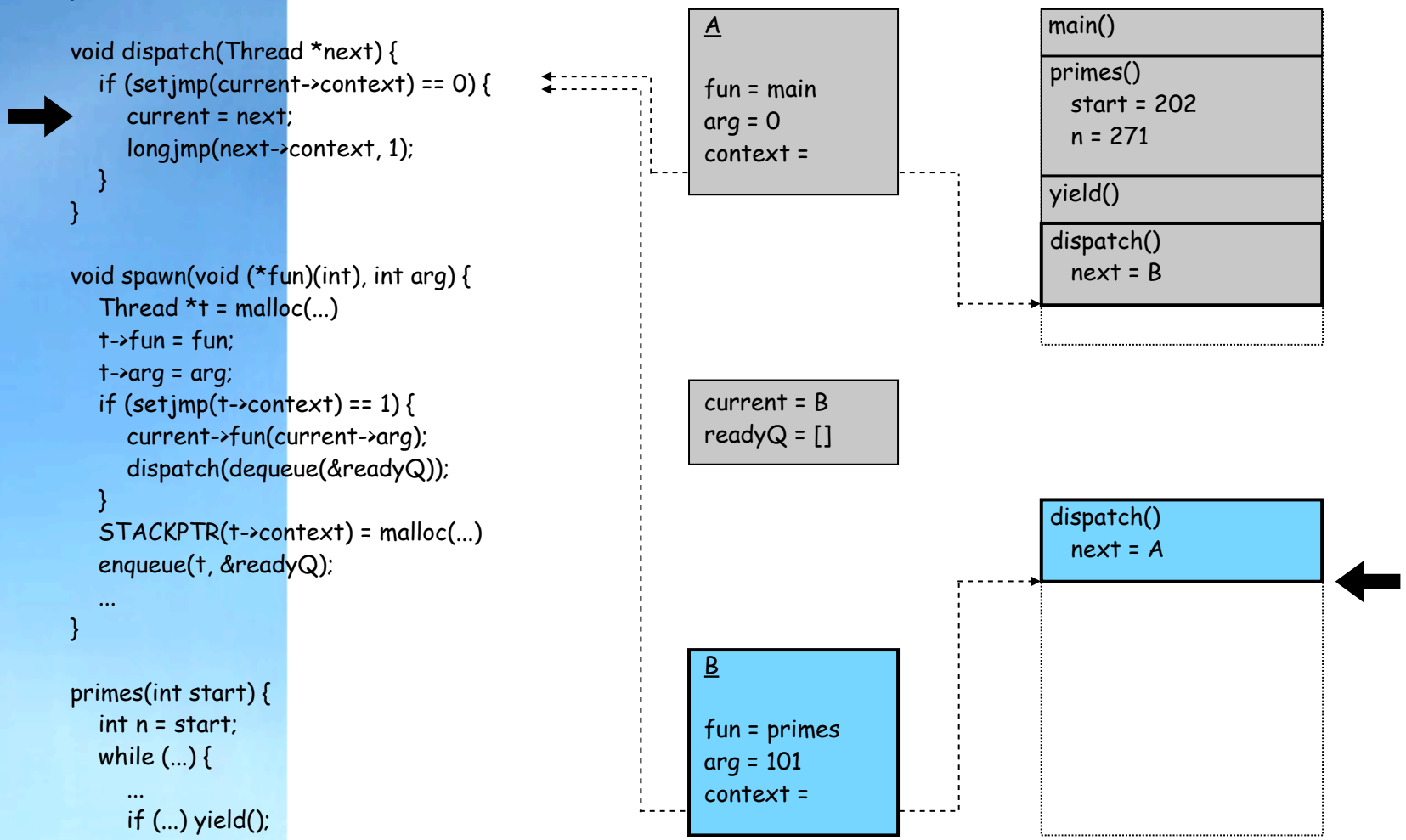
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



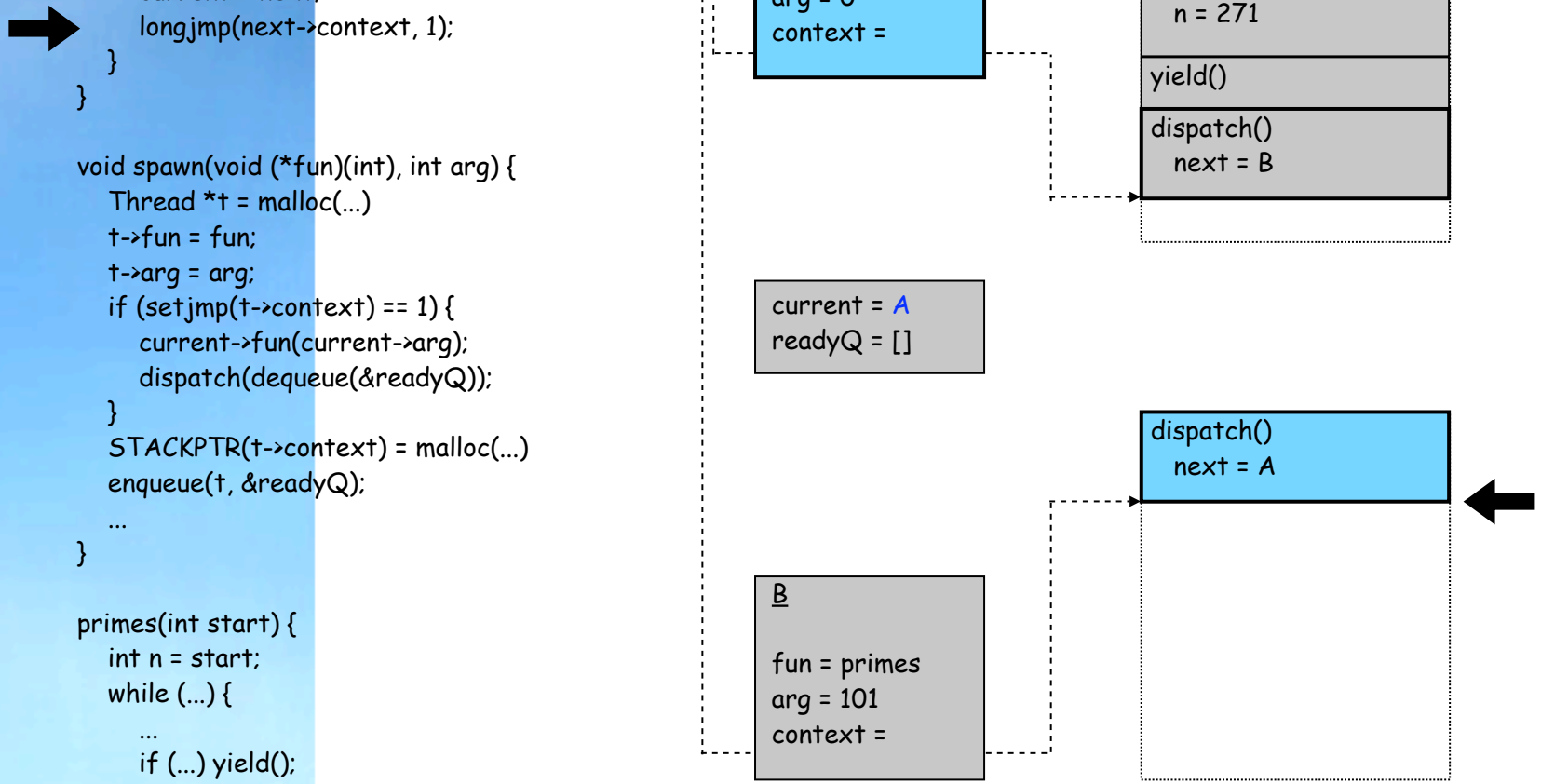
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



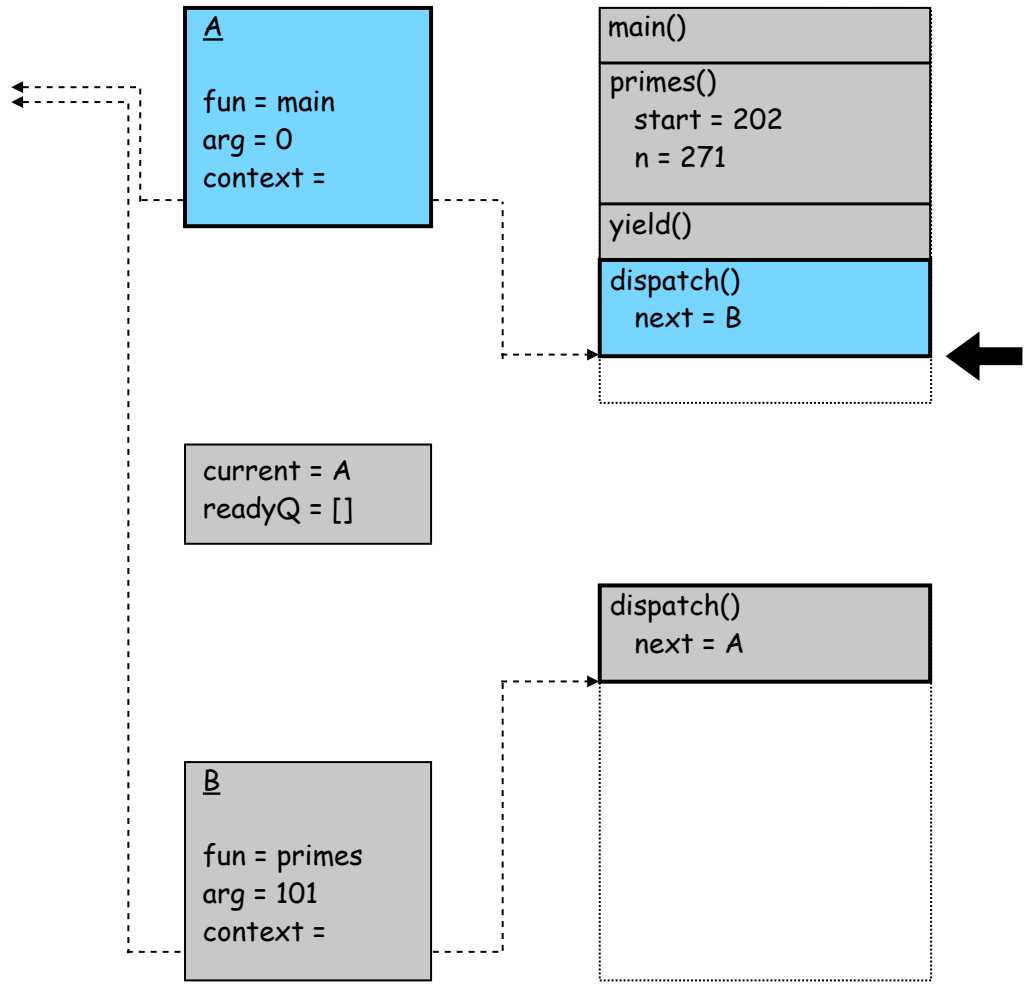
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



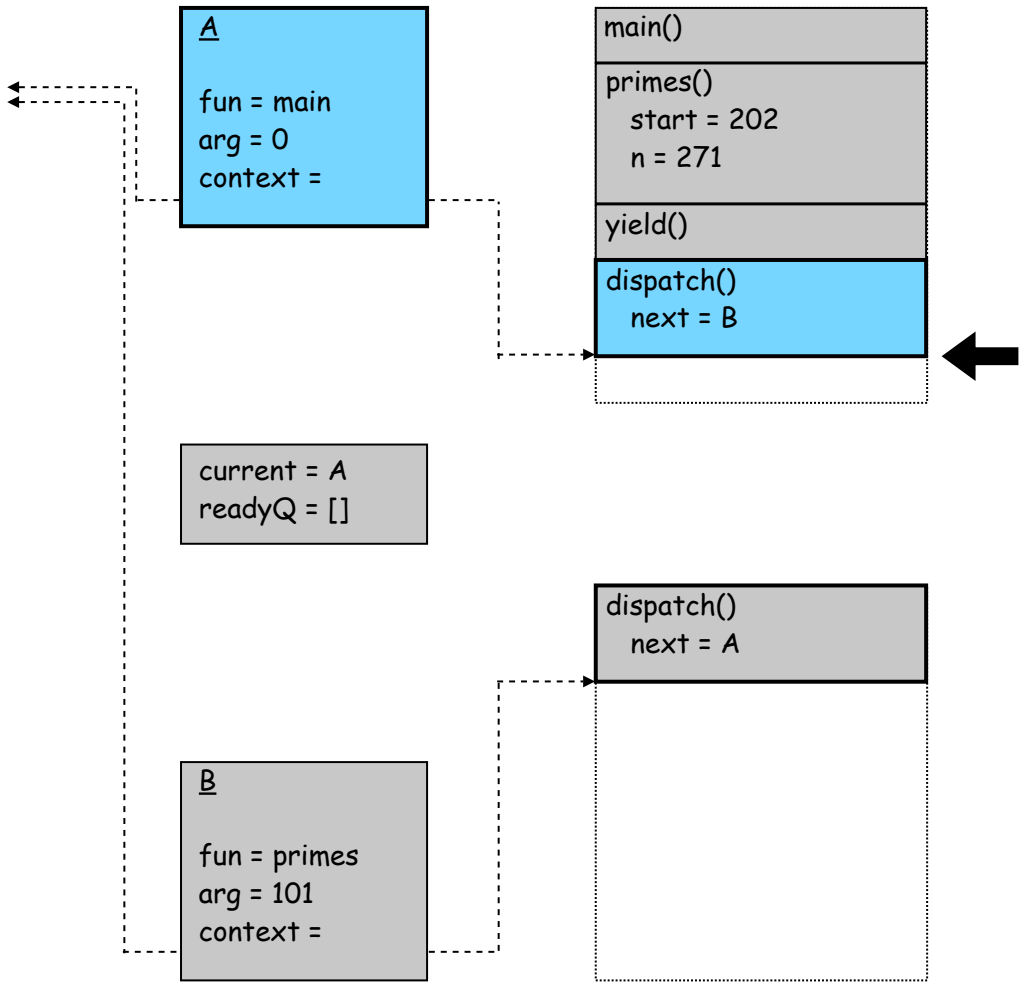
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



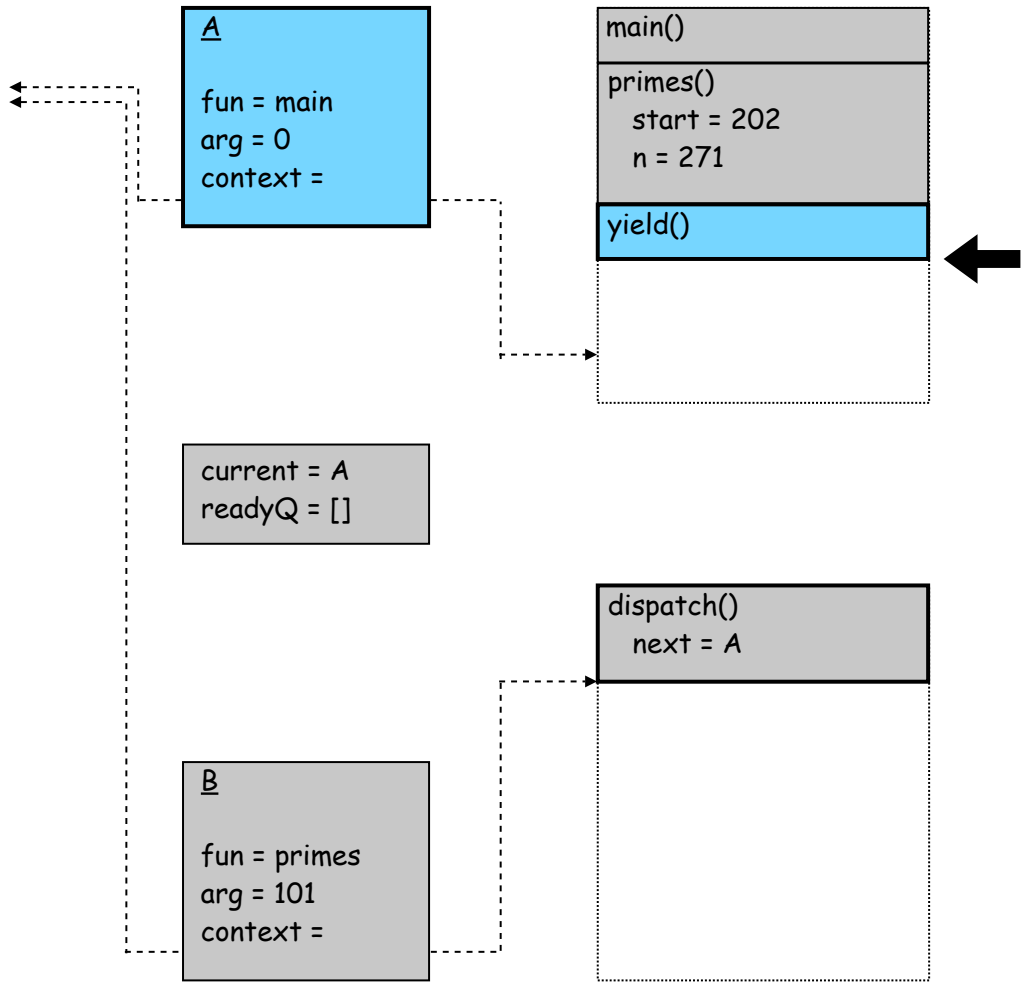
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



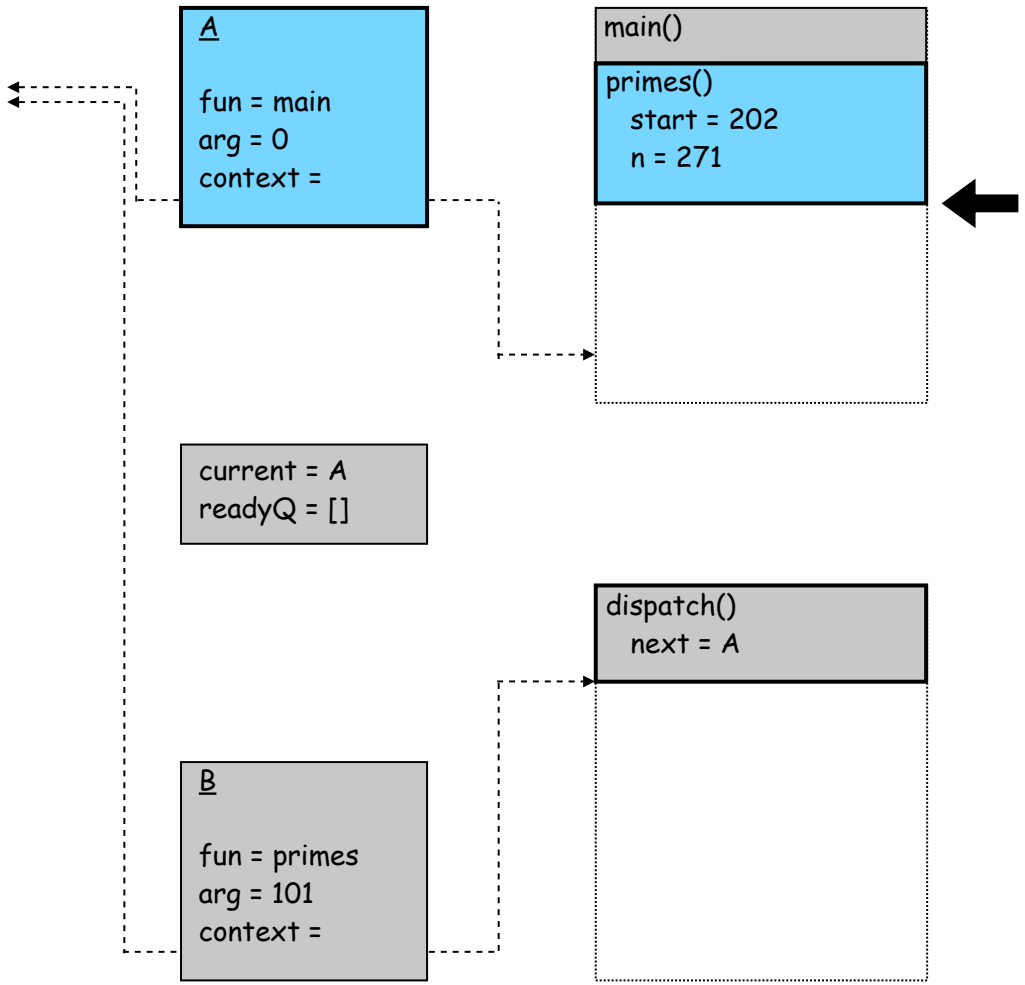
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



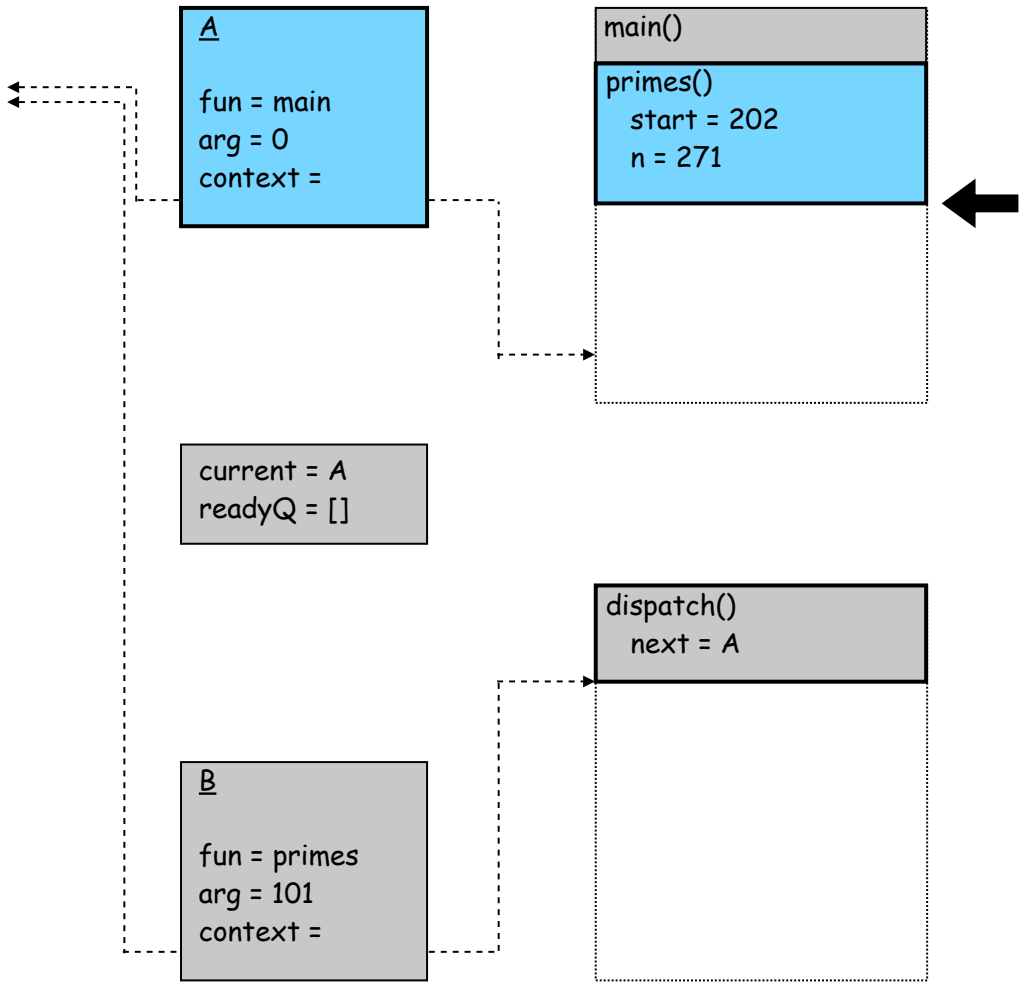
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



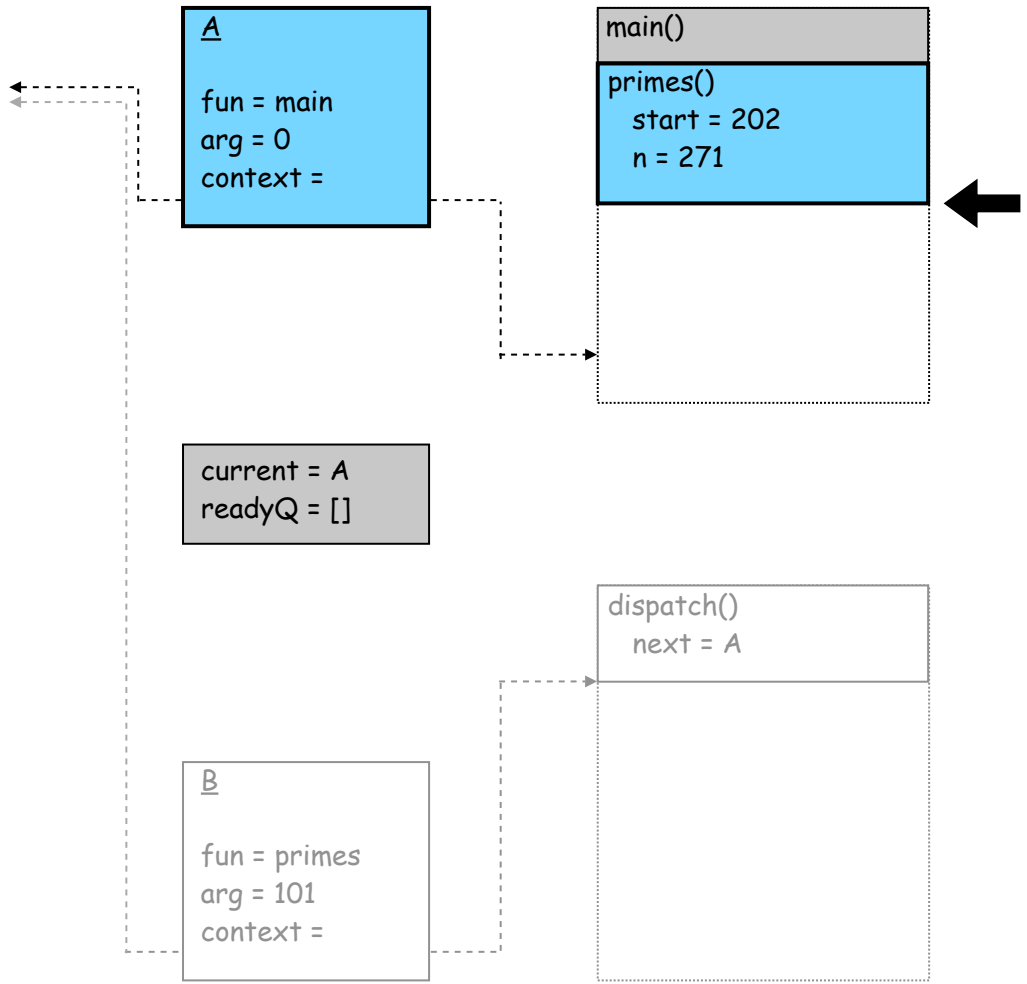
```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
}
```



```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



```
void yield() {
    enqueue(current, &readyQ);
    dispatch(dequeue(&readyQ));
}

void dispatch(Thread *next) {
    if (setjmp(current->context) == 0) {
        current = next;
        longjmp(next->context, 1);
    }
}

void spawn(void (*fun)(int), int arg) {
    Thread *t = malloc(...)
    t->fun = fun;
    t->arg = arg;
    if (setjmp(t->context) == 1) {
        current->fun(current->arg);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(t->context) = malloc(...)
    enqueue(t, &readyQ);
    ...
}

primes(int start) {
    int n = start;
    while (...) {
        ...
        if (...) yield();
    }
}

main() {
    spawn(primes, 101);
    primes(202);
```



Calling yield()

Explicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c
jsr yield
ld c, r0
cmp #37, r0
ble label34
...
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling yield()

Explicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c
jsr yield
ld c, r0
cmp #37, r0
ble label34
...
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling yield()

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c

  ⟵  Interrupt on pin 3!

ld c, r0
cmp #37, r0
ble label34
...
```

Calling yield()

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c

  ⟵  Interrupt on pin 3!

ld c, r0
cmp #37, r0
ble label34
...
```

Calling yield()

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c

  ⟵  Interrupt on pin 3!

ld c, r0
cmp #37, r0
ble label34
...
```

```
vector_3:
    push r0-r2
    jsr yield
    pop r0-r2
    rti
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling yield()

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c

  ⟵  Interrupt on pin 3!

ld c, r0
cmp #37, r0
ble label34
...
```

```
vector_3:
    push r0-r2
    jsr yield
    pop r0-r2
    rti
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Installing interrupt handlers

```
#include<avr/interrupt.h>

...
ISR(interrupt_name){
...
 // code as in a function body!
...
}
```

Preventing interrupts in avr-gcc

```
cli();
// ... code that must not be interrupted ...
sei();
```

Installing interrupt handlers

```
#include<avr/interrupt.h>


...
ISR(interrupt_name){
...
 // code as in a function body!
...
}
```

Preventing interrupts in avr-gcc

```
cli();
// ... code that must not be interrupted ...
sei();
```

Preventing interrupts

Why should we consider disabling interrupts? What parts of the program should be protected?