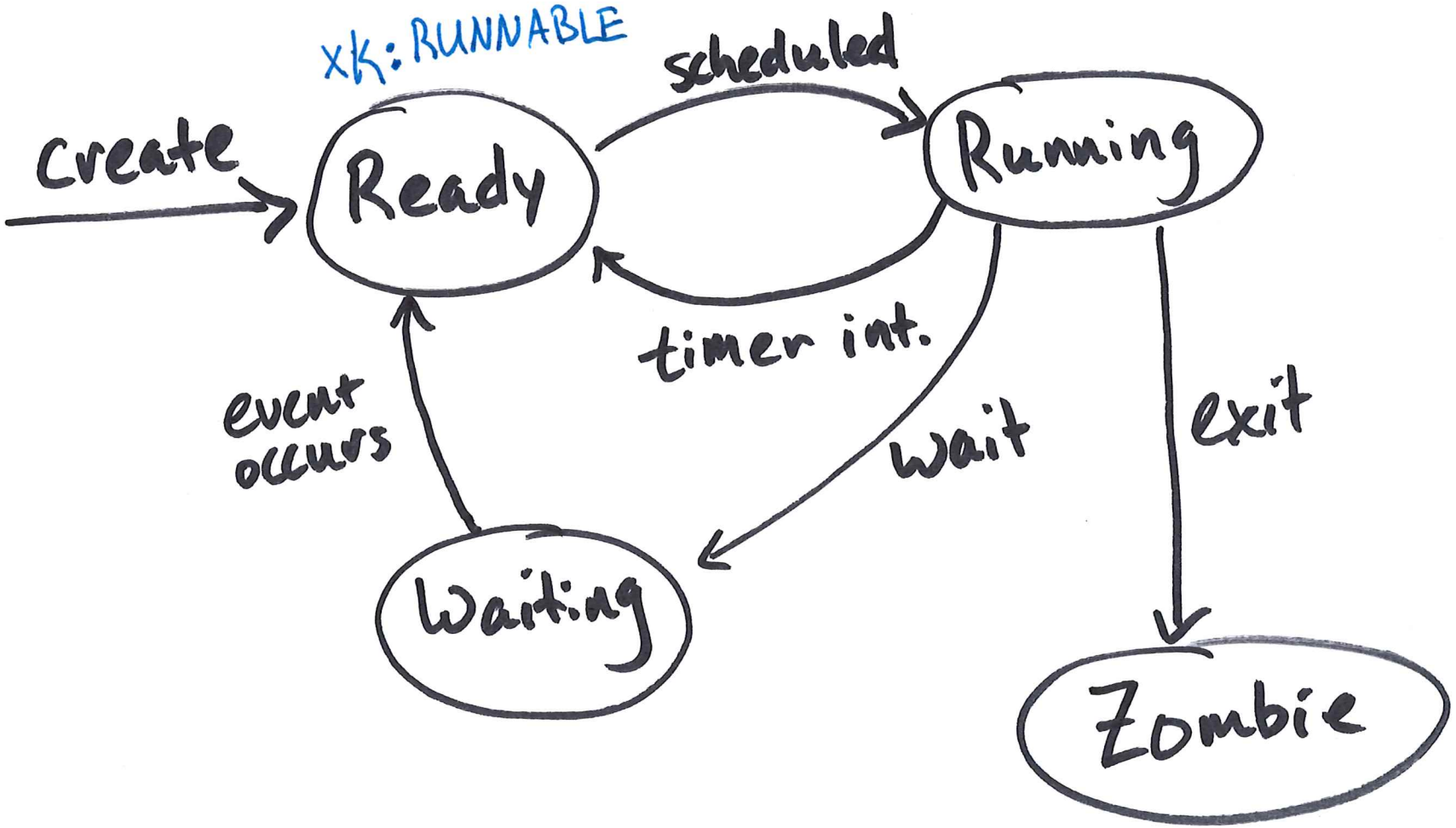
Process Lifecycle (simplified)



Lec 6

Threads

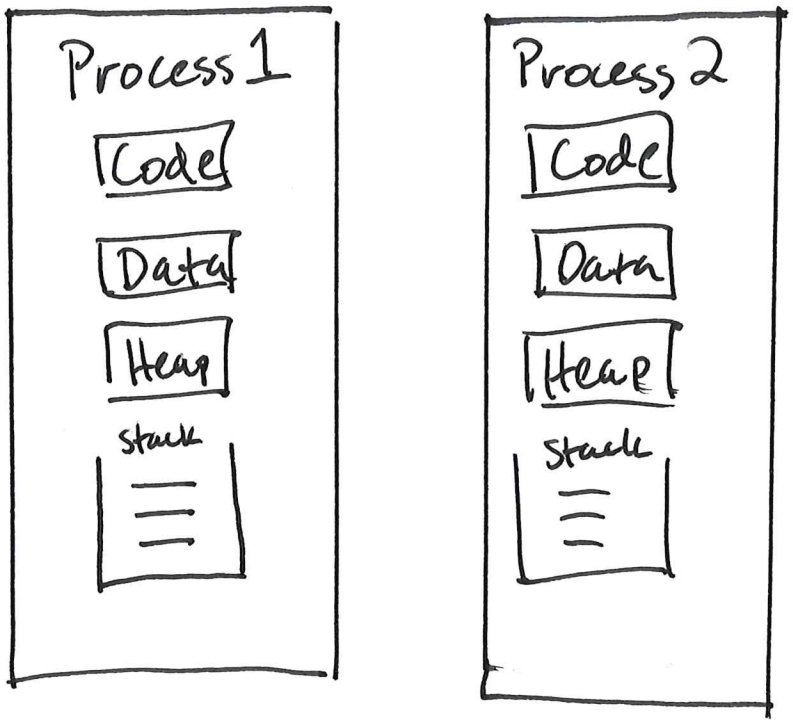
Process Concurrency

- processes share CPU via scheduler
- isolated by default
  - can send messages to each other
    "interprocess communication" IPC
    e.g. pipes (Lab 2)
- can be configured to share memory

Threads

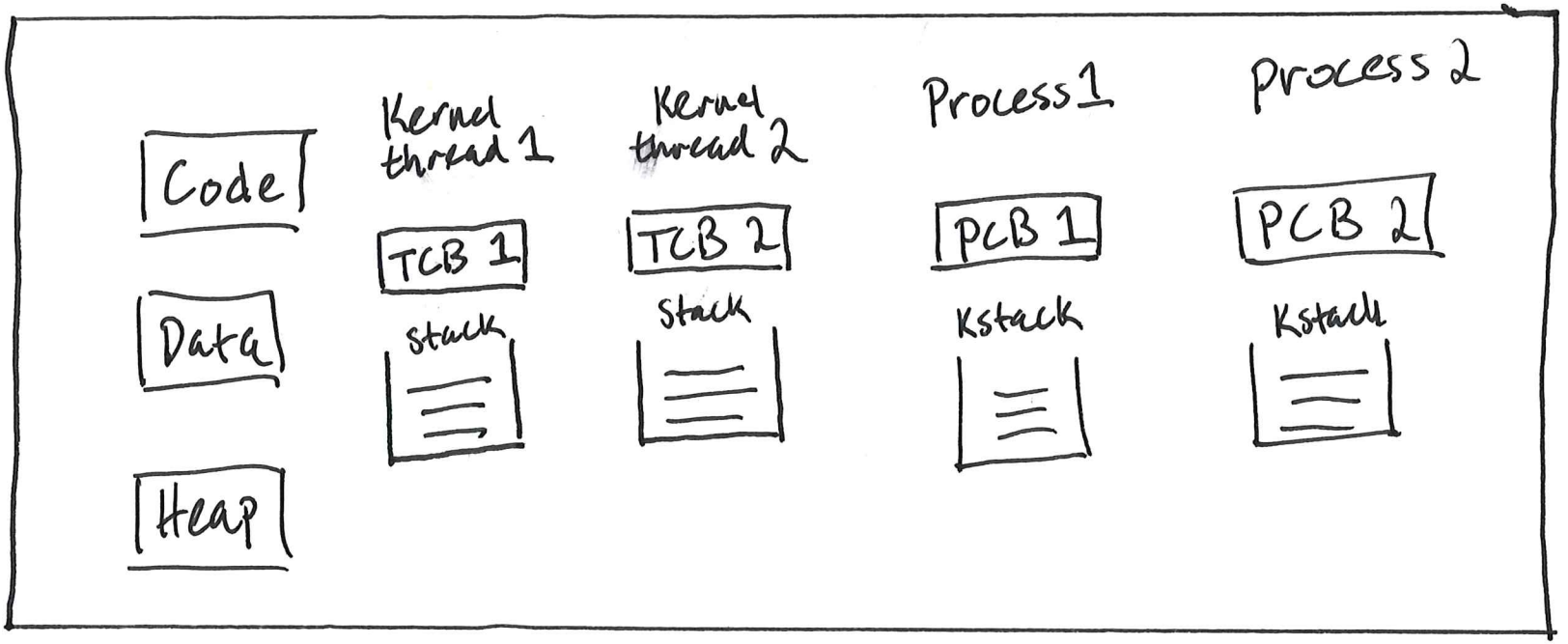
- A **thread** is a single, separately schedulable execution sequence
  - has its own stack and PC
- Can have many threads in one process
  - all share same address space
- Kernel can have its own threads

Multithreaded Kernel

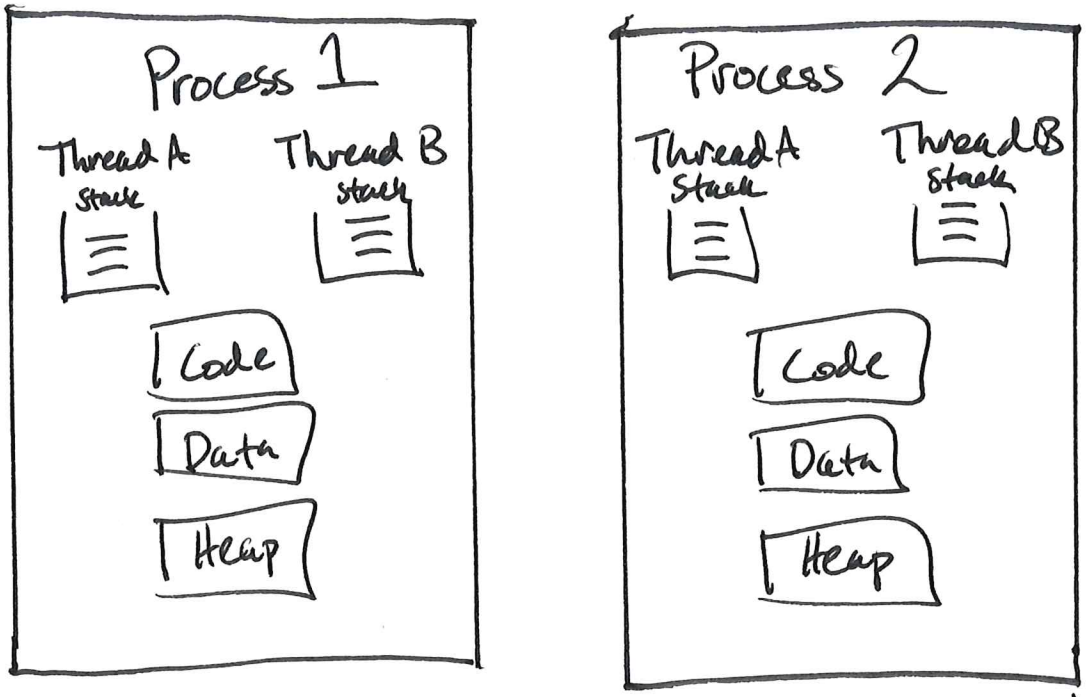


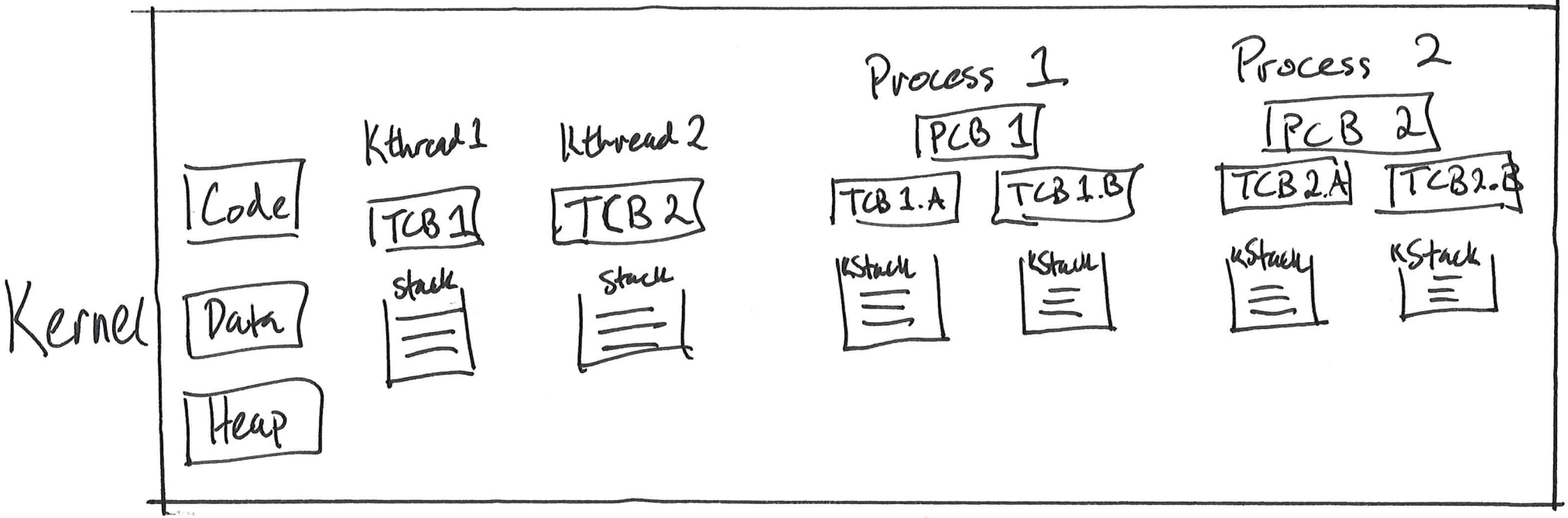
User space

Kernel



Multithreaded User Processes





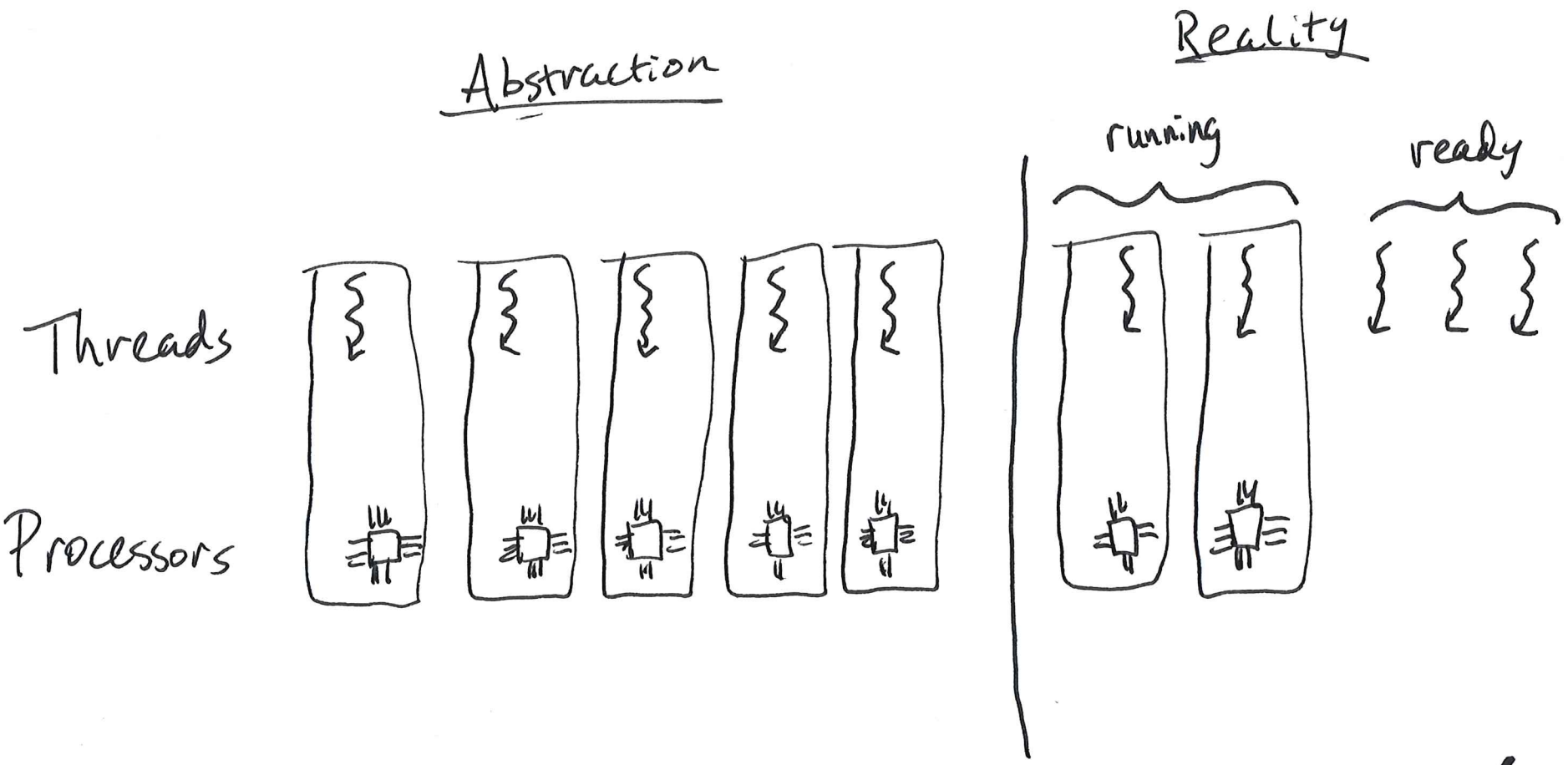
6.4

Thread API

- thread_create(func, args)
- thread_yield() voluntarily give up processor
- thread_join(tid) wait for other thread to exit
- thread_exit()

Thread Mental Model

- ∞ processors execute at variable speed
- programs must work regardless of speed



Do we need Kernel support to implement threads?

- no

- but yes

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

enum cr_stat {
  UNUSED = 0,
  RUNNABLE,
  RUNNING
};

#define STACK_SIZE 2048

char *main_rsp;

struct coroutine {
  enum cr_stat stat;
  char *rsp;
  char stack[STACK_SIZE];
};

#define NCR 10
struct coroutine coroutines[NCR];
int current_crid = -1;

extern void myswtch(char**, char*);
```

```
void coroutine_yield() {
  int old_crid = current_crid;
  int crid = current_crid;
  if (coroutines[crid].stat == RUNNING) {
    coroutines[crid].stat = RUNNABLE;
  }
  crid++;

  if (crid < 0 || crid >= NCR) {
    crid = 0;
  }
  while (crid < NCR &&
         coroutines[crid].stat != RUNNABLE) {
    crid++;
  }

  if (crid == NCR) {
    crid = 0;
    while (crid < NCR &&
           coroutines[crid].stat != RUNNABLE) {
      crid++;
    }
    if (crid == NCR) {
      myswtch(&coroutines[0].rsp, main_rsp);
      printf("coroutine_yield: unreachable\n");
      exit(1);
    }
  }

  coroutines[crid].stat = RUNNING;
  current_crid = crid;
  myswtch(&coroutines[old_crid].rsp,
          coroutines[crid].rsp);
}
```

6.8

```
void coroutine_exit(int crid) {
  coroutines[crid].stat = UNUSED;
  coroutine_yield();
}

void coroutine_preface() {
  coroutine_exit(current_crid);
}

#define PUSH(sp, x) \
  (sp) -= 8; *(void**)(sp) = (x);

int coroutine_create(void (*start)()) {
  int crid = 0;
  while (crid < NCR &&
         coroutines[crid].stat != UNUSED) {
    crid++;
  }
  if (crid == NCR) {
    return -1;
  }
  struct coroutine *cr = &coroutines[crid];

  cr->stat = RUNNABLE;
  bzero(cr->stack, STACK_SIZE);
  cr->rsp = cr->stack + STACK_SIZE;

  PUSH(cr->rsp, coroutine_preface);
  PUSH(cr->rsp, start);

  // push values for callee saved registers
  PUSH(cr->rsp, 0);
  PUSH(cr->rsp, 0);
  PUSH(cr->rsp, 0);
  PUSH(cr->rsp, 0);
  PUSH(cr->rsp, 0);
  PUSH(cr->rsp, 0);

  return crid;
}
```

```
void coroutine_start() {
  int crid = 0;
  while (crid < NCR &&
         coroutines[crid].stat != RUNNABLE) {
    crid++;
  }

  if (crid == NCR) {
    printf("no runnable coroutines\n");
    exit(1);
  }

  printf("going to run cr %d\n", crid);

  coroutines[crid].stat = RUNNING;
  current_crid = crid;
  myswtch(&main_rsp, coroutines[crid].rsp);
}
```

6.9

```
void testA() {
  printf("A with crid %d\n", current_crid);
  coroutine_yield();
  printf("hello from A again\n");
  coroutine_yield();
  printf("goodbye from A\n");
}

void testB() {
  printf("B with crid %d\n", current_crid);
  coroutine_yield();
  printf("hello from B again\n");
  coroutine_yield();
  printf("goodbye from B\n");
}

int main() {
  printf("starting in main\n");
  int crid_A = coroutine_create(testA);
  printf("main: created A with crid = %d\n",
         crid_A);
  int crid_B = coroutine_create(testB);
  printf("main: created B with crid = %d\n",
         crid_B);
  coroutine_start();
  printf("back in main\n");
}
```

6.10