Threads in user space

↳ create (TCB + stack + Q)

↳ yield (to swith threads

↳ not a goto

f1() { ∞ loop }

f2() { ∞ loop }

main

{ fork → call f1

f2

}

separate process runs f1

How to we implement threads?

→ need thread control blocks

→ need context switches

↓

- hardware dependent
- need assembly

library

→ ucontext_t → a type (struct) that stores a thread context

→ makecontext() } call to init TCBs
  getcontext() }

→ swapcontext (from, to) switch

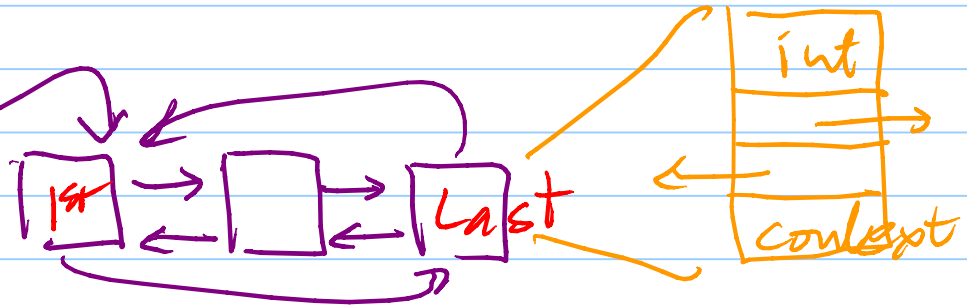
Need a TCB

struct

TCB_t → int thread_id
* next
* prev
ucontext_t context

TCB_t

Q structure →



initQ(Q)

AddQ(Q, element)

element = DelQ(Q)

Works on multiple Qs

RotateQ(Q) ≡ AddQ(Q, DelQ(Q));

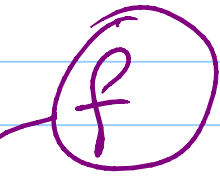
// done by moving the header

to start a thread → for function 'f'

- allocate a TCB (malloc)

- allocate a stack (malloc)
  ↳ ... size? 8k

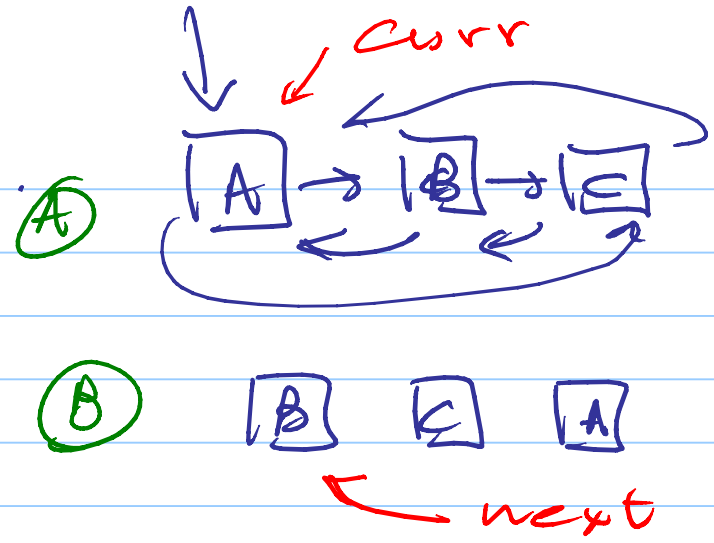
- INIT TCB → complicated
  (code provided) uses makecontext()

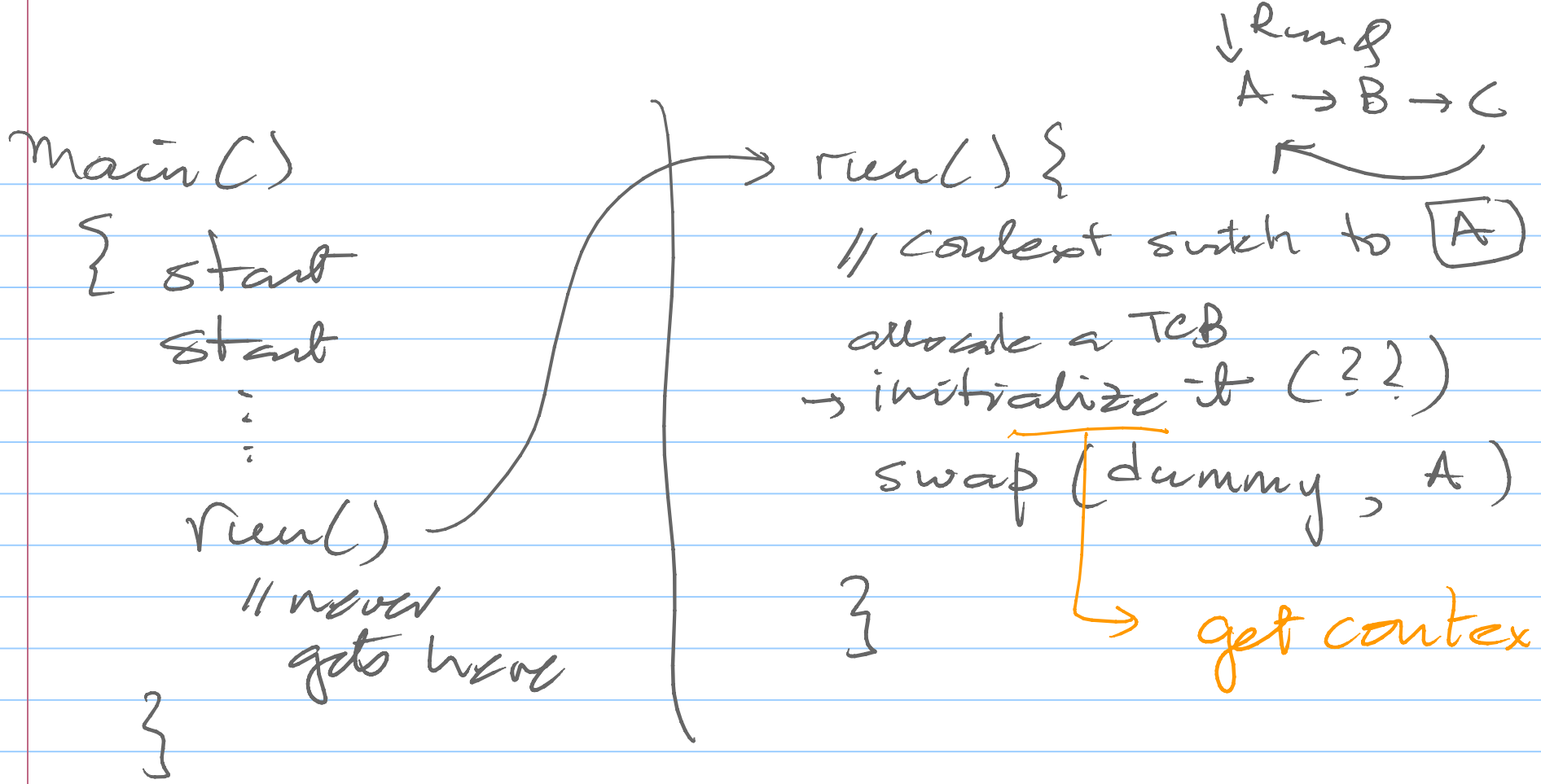


Global pointer

```
StartThread(f)
{ allocate  ---> TCB & stack
    makeContext(f)
    AddQ(RunQ, TCB)
}
```

```
yield()
{ curr = RunQ;
  RotateQ(RunQ);
  next = RunQ
  swapcontext(curr->context,
              next->context)
}
```




Rung
A → B → C
main()
{ start
start
:
:
rtn()
// never
gets here
}
rtn() {
// context switch to A
allocate a TCB
→ initialize it (??)
swap (dummy, A)
}
get context

yield()
{
=
=
=

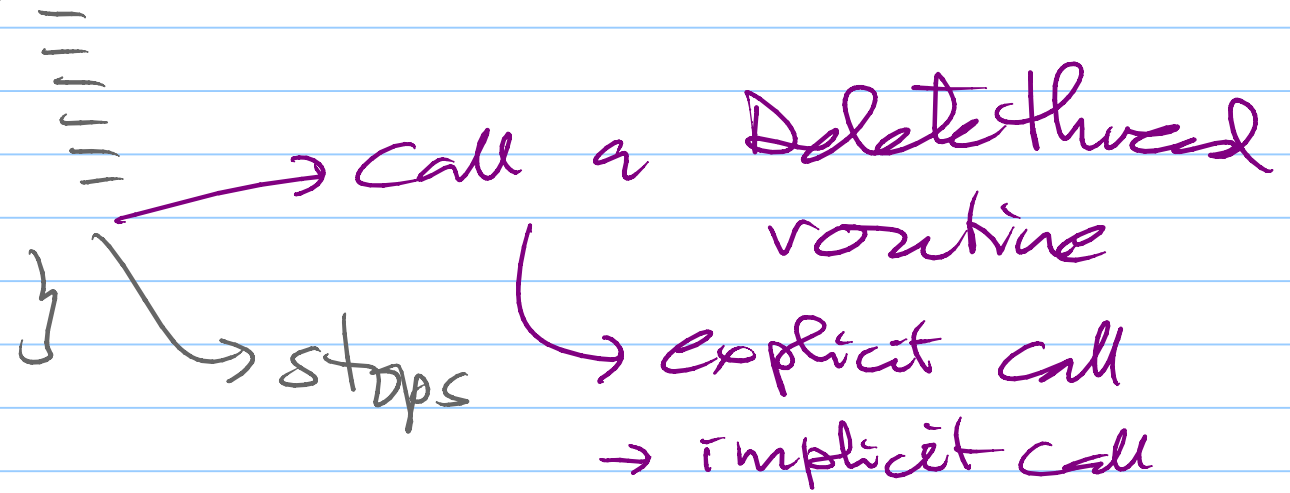
① what if Run Q has 1 TCB ?
→ thread continues

② » Run Q has 0 TCB

→ add an idle process →

Can a thread terminate?

f1() { ← start it



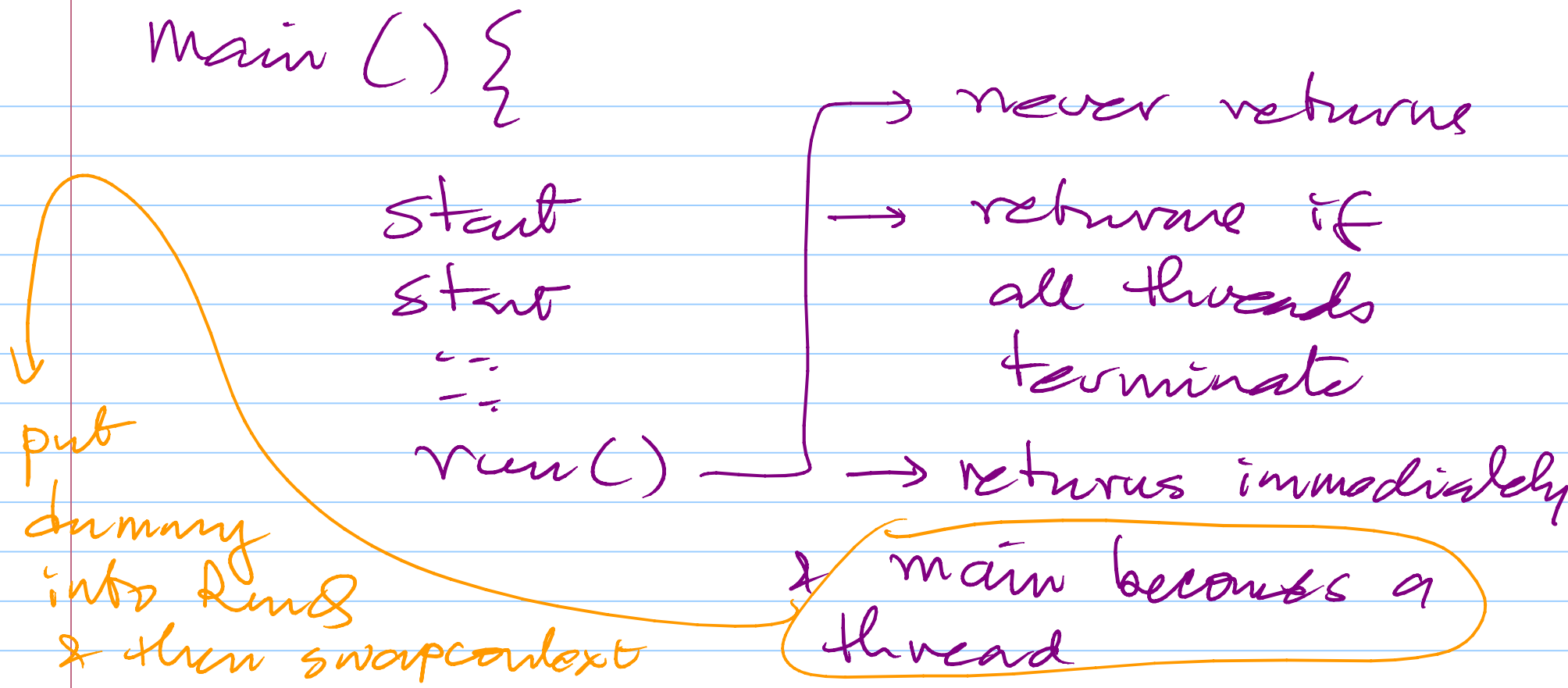
RunQ is empty

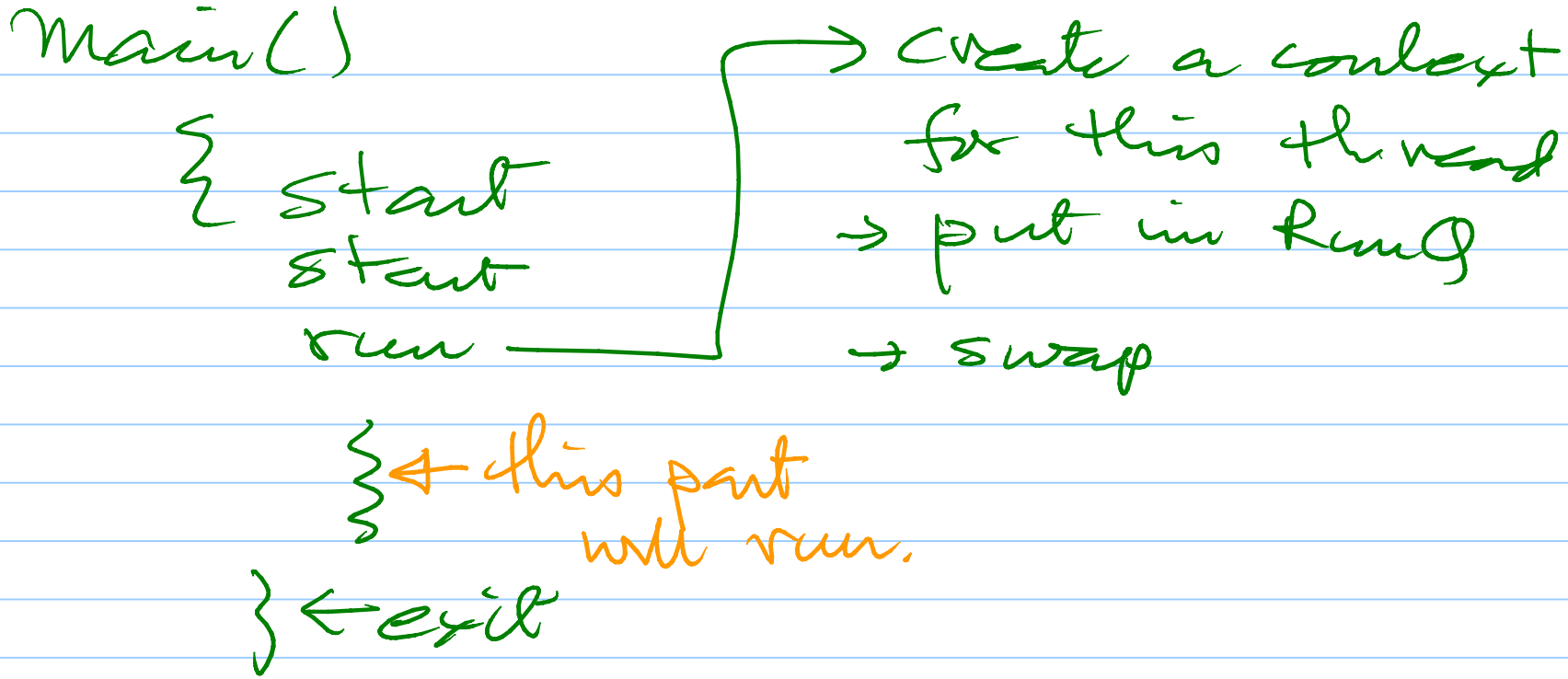
→ run idle process/thread

→ terminate

→ return to main

↳ need to have the "dummy" TCB

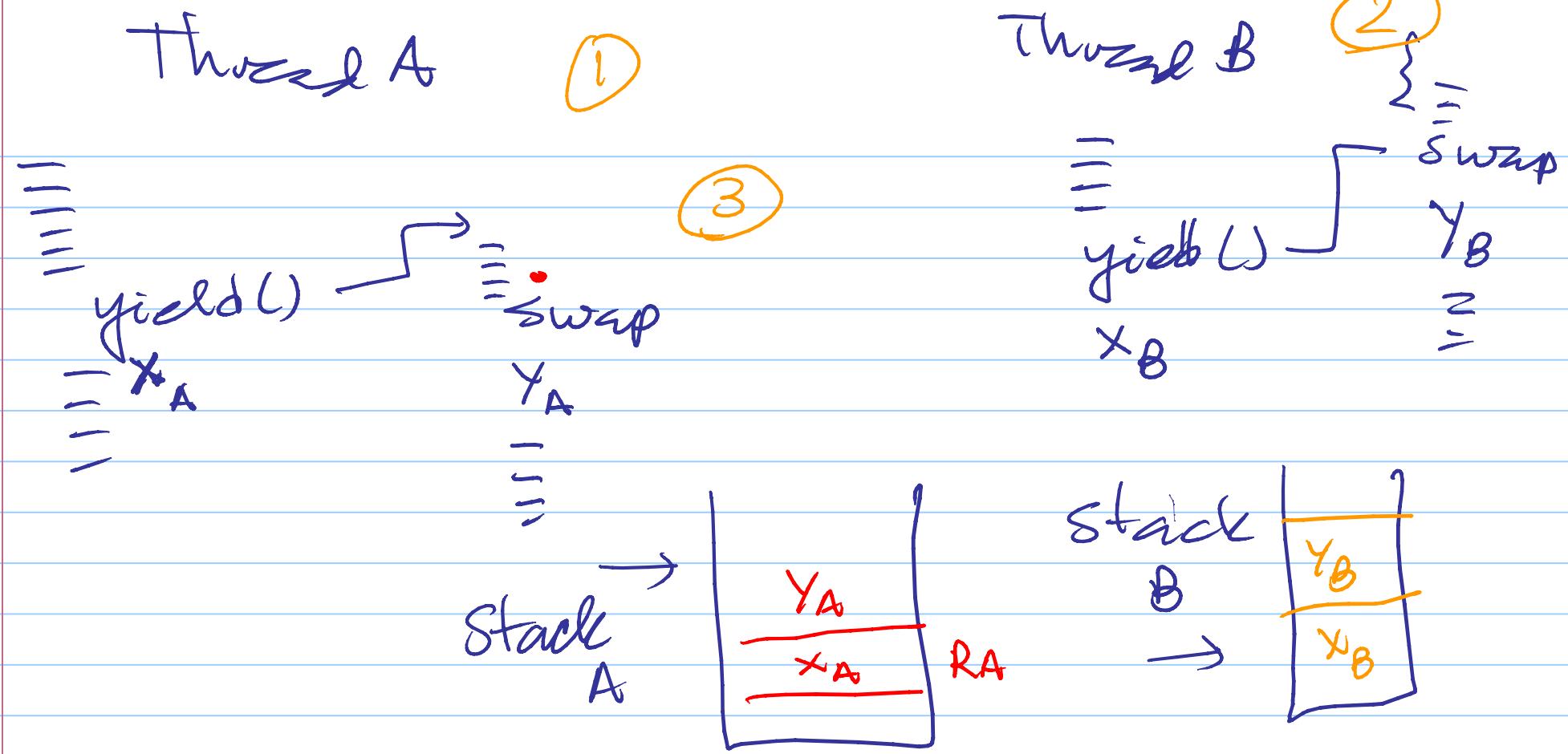

Main () {
start
start
run()
never returns
returns if all threads terminate
returns immediately
main becomes a thread
put dummy into RunS & then swapcontext


Main()
{ start
start
run
create a context
for this thread
→ put in RunQ
→ swap
this part
will run.
exit

Another look at swapcontext

```
yield()
  {
  =
    swap(f, t)
  }
```

```
swap(f, t)
  { move regs → f
    move sp → f
    move t → regs
    move t → sp
  } ← RET
```


Thread A
1
Thread B
2
swap
yield()
x_A
y_A
3
yield()
x_B
y_B
Stack A
y_A
x_A
RA
stack B
y_B
x_B

- User level scheduling
  - non preemptive (need yield)
    - execution is atomic
      - ↳ critical sections for free
    - explicit yields are NOT needed

kernel level scheduling

- Thread creation → kernel call → TCB stored in kernel
- yield → kernel call
  ↳ also called by timer int

→