Project 1
Non-Preemptive Multitasking

Kai Mast

Department of Computer Science
Cornell University

February 3rd, 2017

Excited about writing your own Operating System?

- Project 1 is already released!
- It is due February 17th
- Currently only complies with GCC <= 5
- BUT, let us talk about C first!

Enumerated Types and Constants

- Enums are consecutive integers starting from 0
- unless you say otherwise…
- Not “advanced” just really important
- ***Do not use magic numbers in your code!***

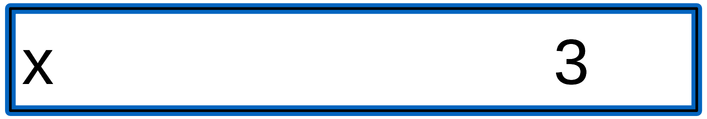
```
enum month_t { JANUARY,
               FEBRUARY,
                MARCH
};
```

Constants should be all in caps

```
const int MAX_PLAYERS = 10;
```

Pointer example

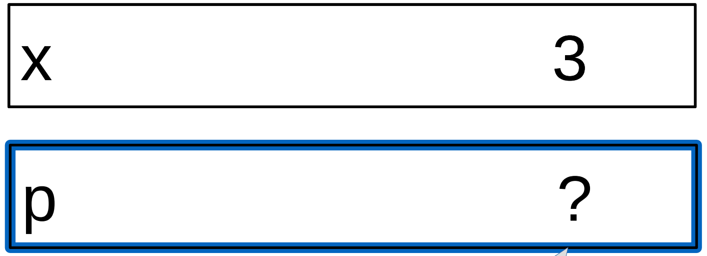
```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```



x 3

Pointer example

```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```

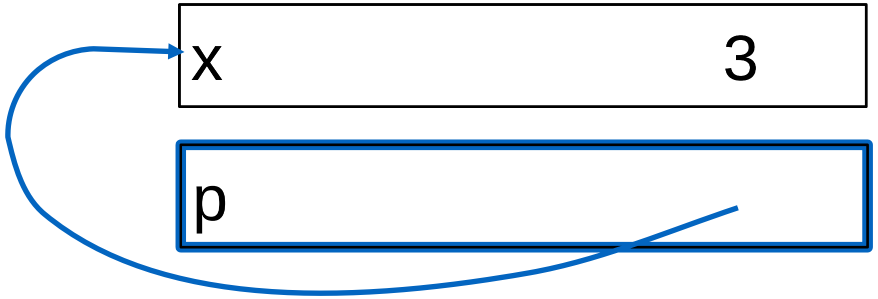


x	3
p	?

Be careful!
p has a random value at this point.

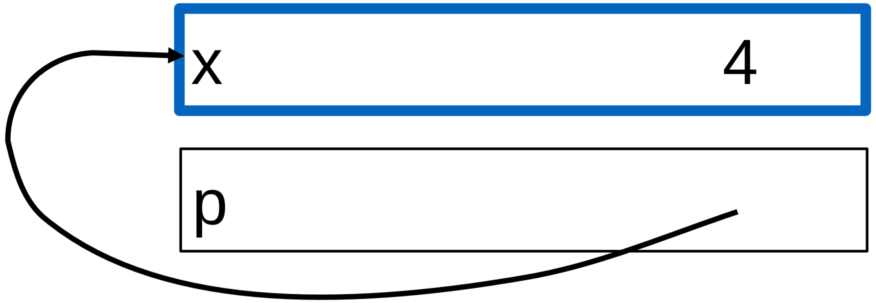
Pointer example

```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```



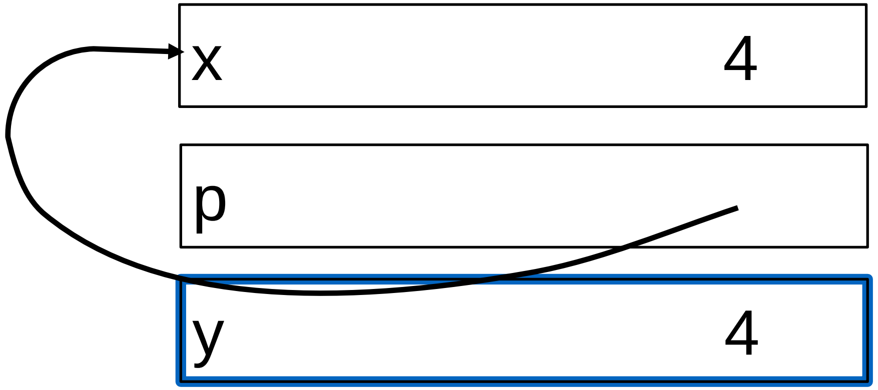
Pointer example

```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```



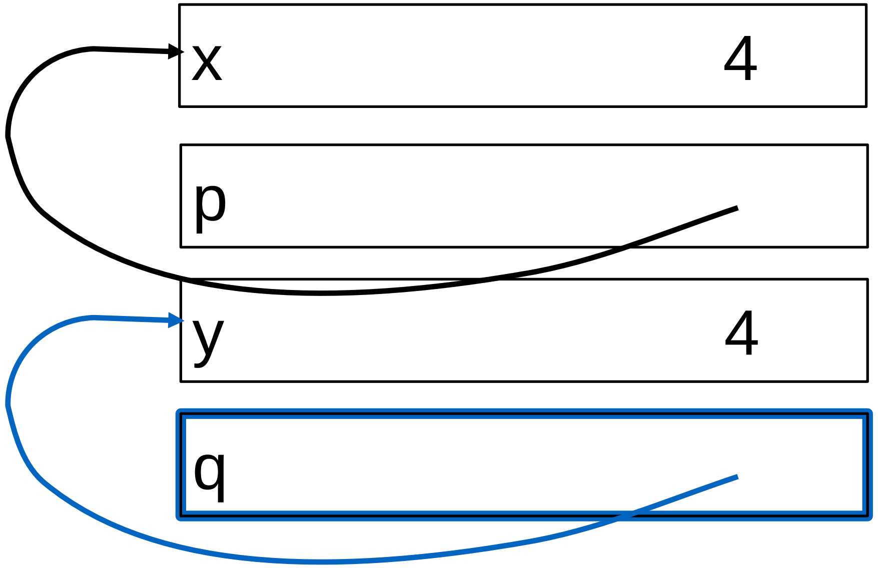
Pointer example

```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```



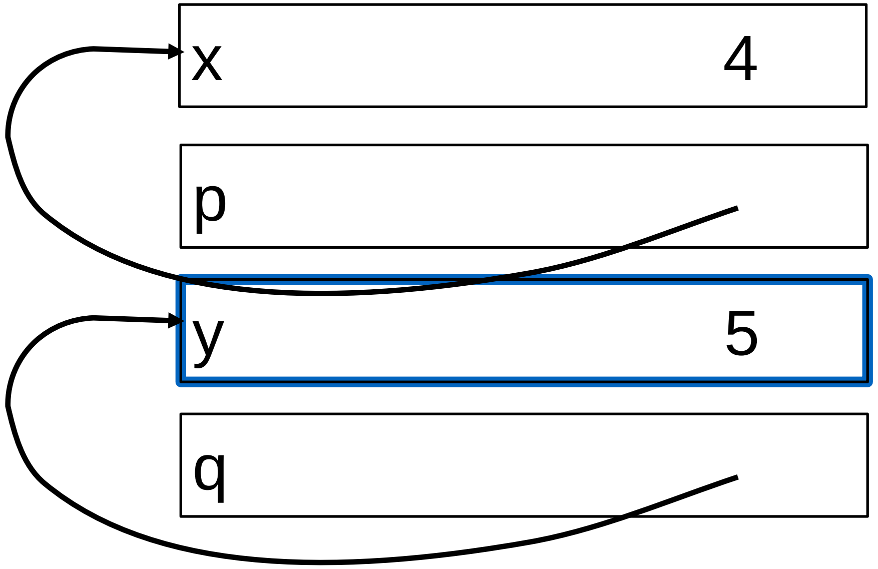
Pointer example

```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```



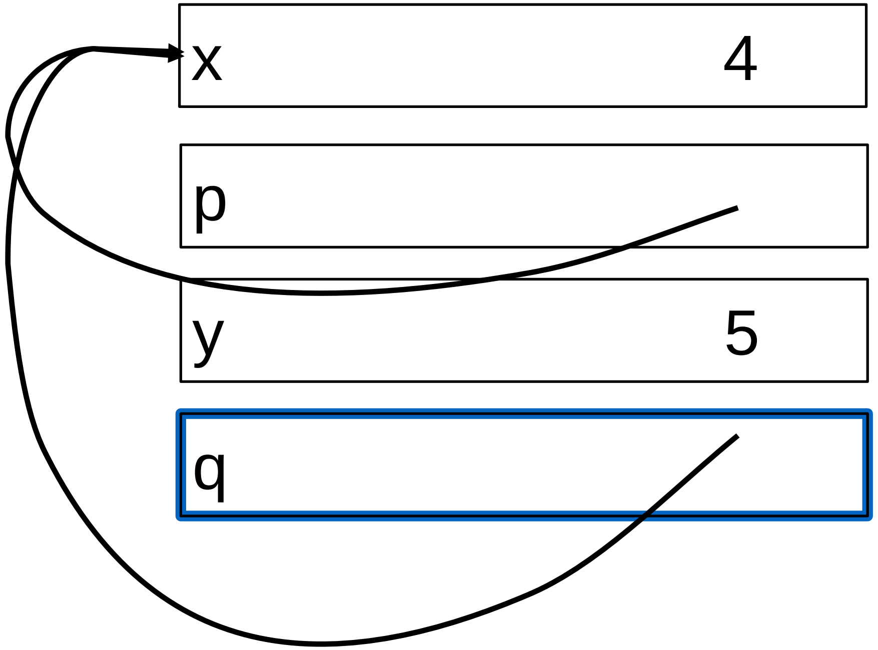
Pointer example

```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```



Pointer example

```
int x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
int *q = &y;
*q = *p + 1;
q = p;
```



q and p now point to the same address, but the value only exists once.

Dynamic Memory Allocation

malloc(len)	“Give me a buffer of `len` bytes.”
free(ptr)	“I don’t need what `ptr` points to anymore.”
realloc(ptr, len)	“Change the size of what `ptr` points to to `len`.”
sizeof(x)	“Give me the size of the **type** of x.”

What is the size of a type?

`sizeof(x)` is your friend!

Once again, don’t use magic numbers.

- int is not 4 bytes on every system
- You might change the type of a variable at some point in the future!

Don’t use sizeof on pointers.

- Sizeof will give you the size of the pointer
- Not what the pointer points to

Allocating arrays

Pointers may point to single elements or arrays.

```
const size_t NUM_ELEMENTS = 42;

e1l33t_type_t *ptr = NULL;

ptr = (e1l33t_type_t*) malloc(NUM_ELEMENTS * sizeof(*ptr));
```

Casting from generic void* to our custom type

This is the same as `sizeof(e1l33t_type_t)`

What is wrong with this example?

```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int *p = malloc(i * sizeof(*p));
    x = do_some_computation(x, i, p);
  }
  printf(“Answer %d\n”, x);
  return EXIT_SUCCESS;
}
```

Much Better ☺

```
int main (void) {
  int x = 0;
  for (int i = 10; i < 100; i++) {
    int *p = malloc(i * sizeof(*p));
    x = do_some_computation(x, i, p);
    free(p);
  }
  printf(“Answer %d\n”, x);
  return EXIT_SUCCESS;
}
```

Don’t do this.

```
char *str1 = malloc(1024 * sizeof(char));
char *str2 = str1;

free(str1);
free(str2);
```

Don’t do this either.

```
char *str = malloc(1024 * sizeof(char);
char *substr = str1[5];

free(substr);
```

Definitely don’t do this

```
char *str = “I love 4410”;
free(str);
```

str is not dynamically allocated but on your stack!

Passing values by pointers

```
void set_to_three(int *i_ptr) {
    *i_ptr = 3;
}

int main() {
    int i = -1;
    set_to_three(&i);
    printf(“i is 3 now!”);
    return 0;
}
```

Passing values by pointers
…to pointers?

```
void my_alloc_function(void **p) {
   *p = malloc(14853);
}

int main() {
    void *p = NULL;
    my_alloc_function(&p);
    printf(“p is not NULL anymore!”);
    free(p);
    return 0;
}
```

Unless malloc returns NULL,
which can happen ☺

Function pointers

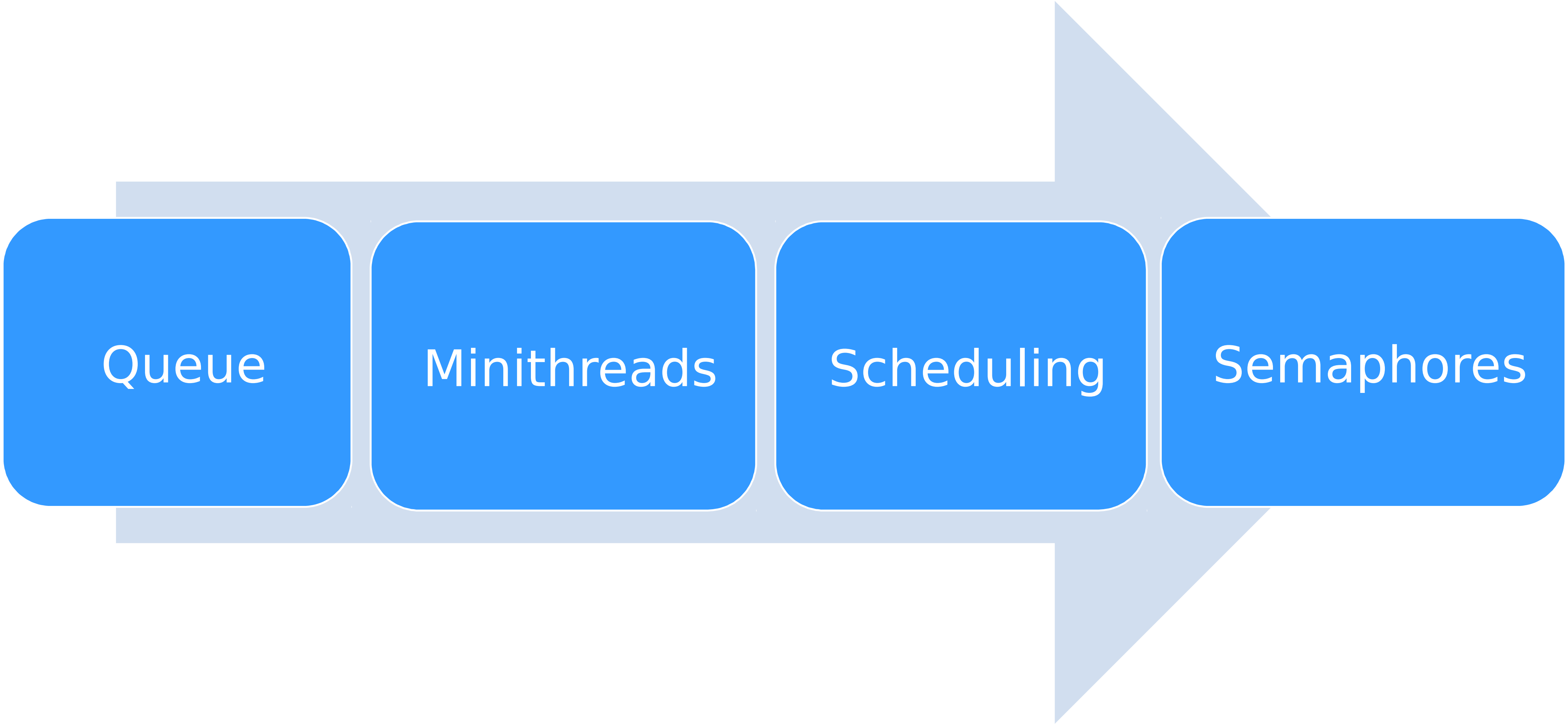
```
int inc(int i) {return i+1;}
int dec(int i) {return i-1;}

int apply (int (*f)(int), int i){
     return f(i);
}

int main() {
  printf(“++: %i\n”, apply(inc, 10));
  printf(“--: %i\n”, apply(dec, 10));
  return 0;
}
```

And now the fun part…

Goals of Project 1

- A “gentle” introduction to C and PortOS
- Learn how threading works
- Implement synchronization primitives
- This is going to be a large project
  → bad coding style WILL bite you later

Project Overview



Queue

Minithreads

Scheduling

Semaphores

Queues

- Just a simple FIFO queue (with some additions)
- Prepend, append and dequeue must be **O(1)**
  → Use a linked list under the hood

`queue_prepend(q,x)`

“Place item in the front of q”
→ needed for peeking

`queue_iterate(q,f,p)`

“Apply f(p) to every element in q ”

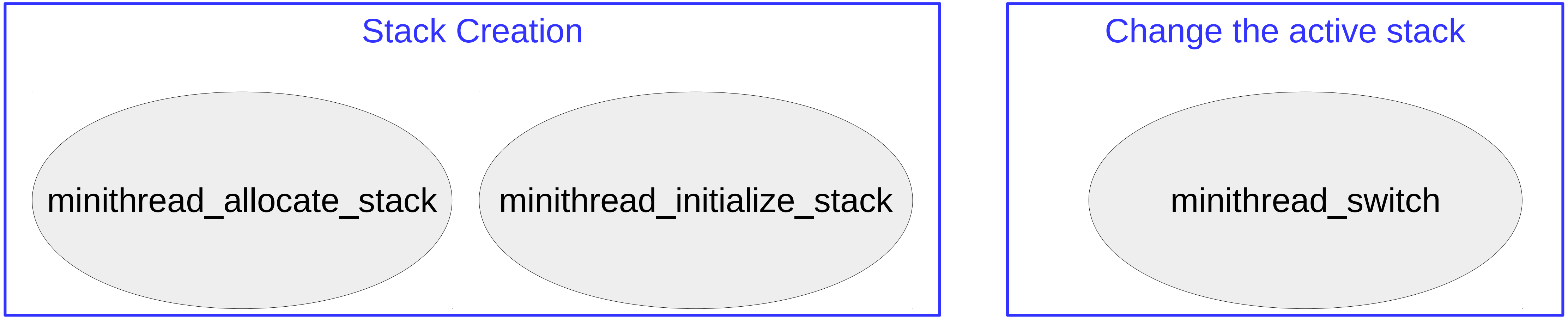
`queue_delete(q,x)`

“Delete the first instance of x in q”

Minithreads

- What we call threads in PortOS
- Majority of the project
- Each thread runs a body procedure (`body_proc`)
- Will need a Thread Control Block
  - Stack top pointer
  - Stack base pointer
  - Thread ID
  - Anything else you want

Useful functions for Thread Management



Stack Creation

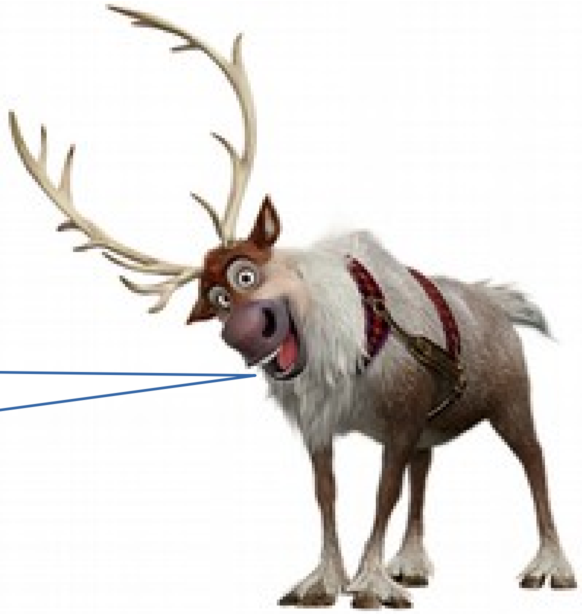
minithread_allocate_stack

minithread_initialize_stack

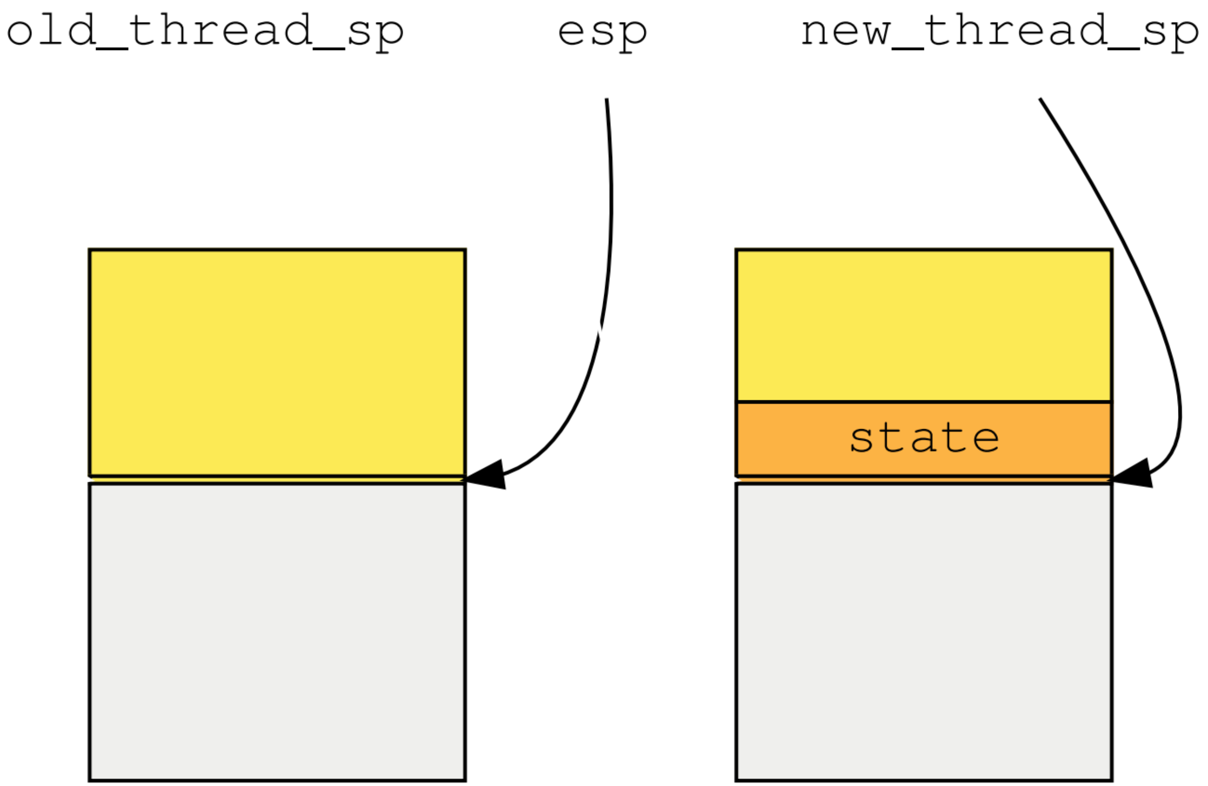
Change the active stack

minithread_switch

Make sure to read `machineprimitives.h`

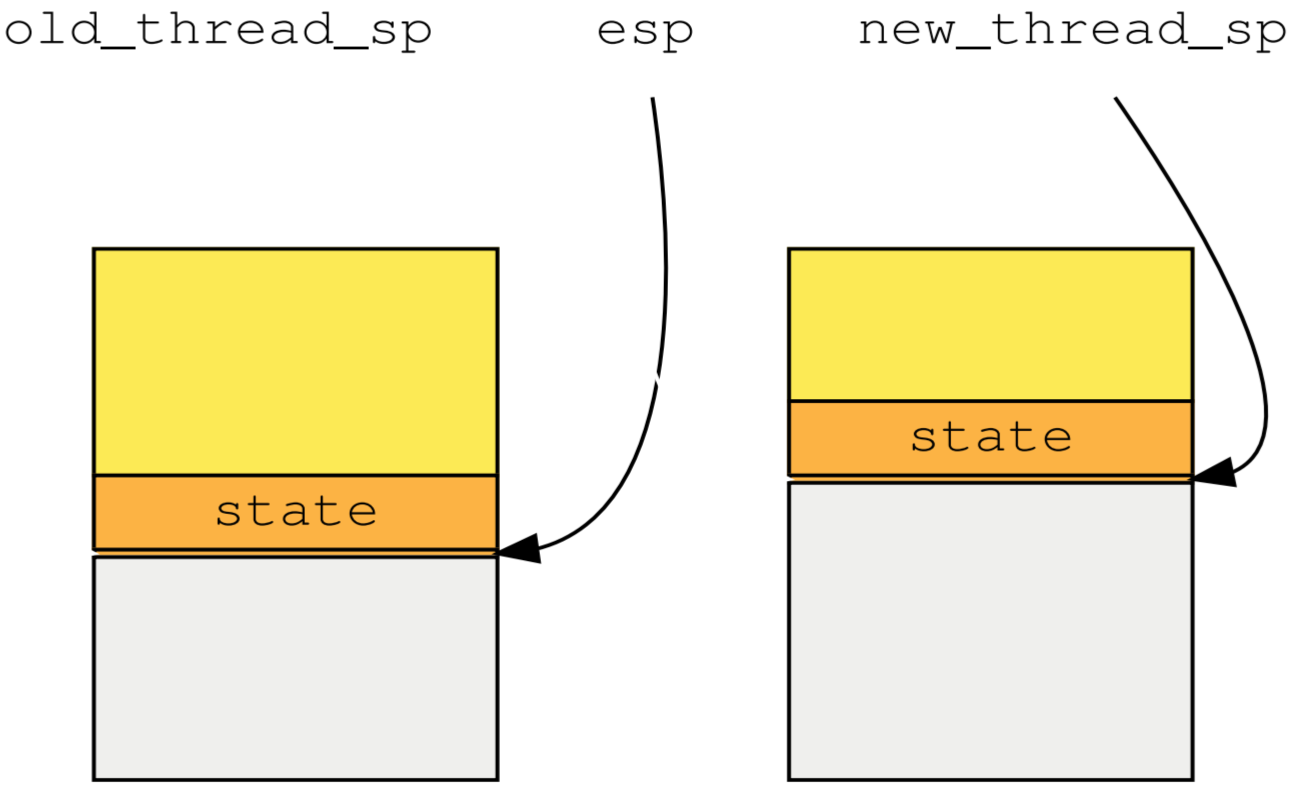


minithread_switch



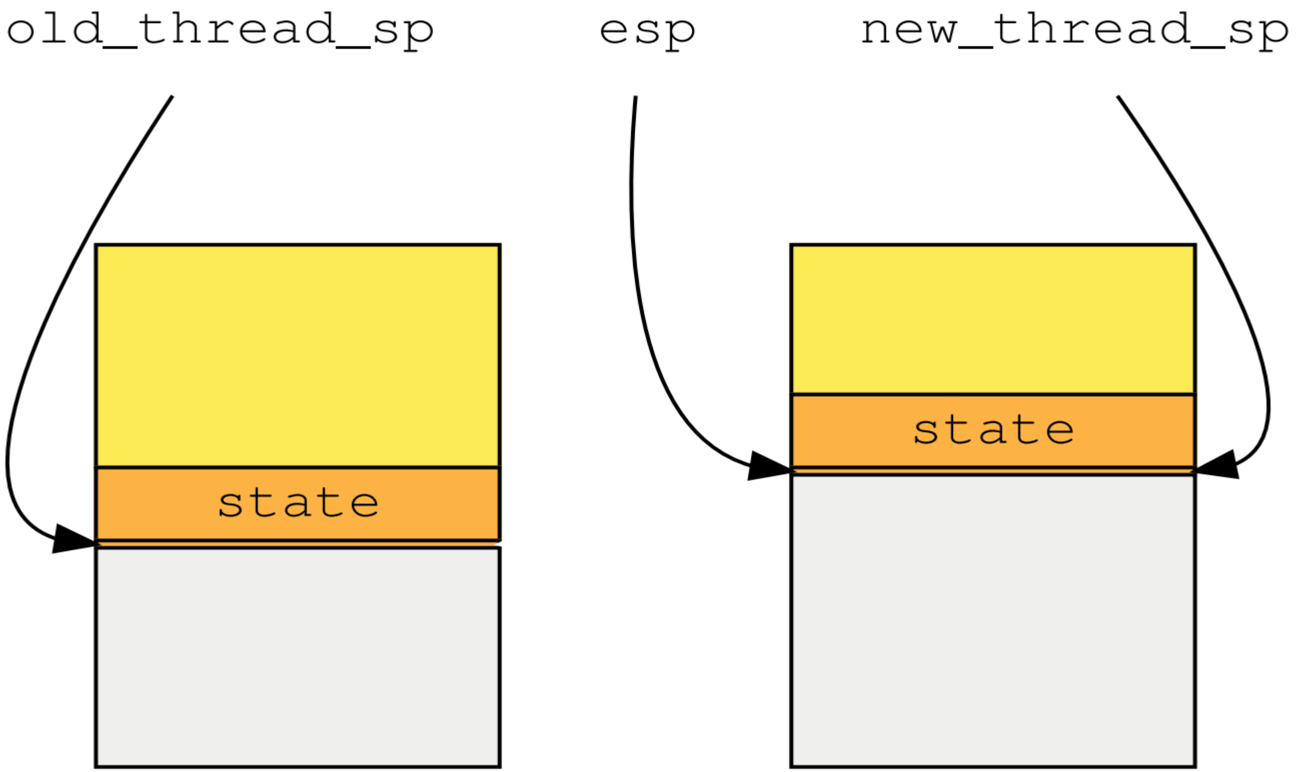
The stack pointer still points to the old thread’s stack, while the new thread is stored somewhere else in memory.

minithread_switch



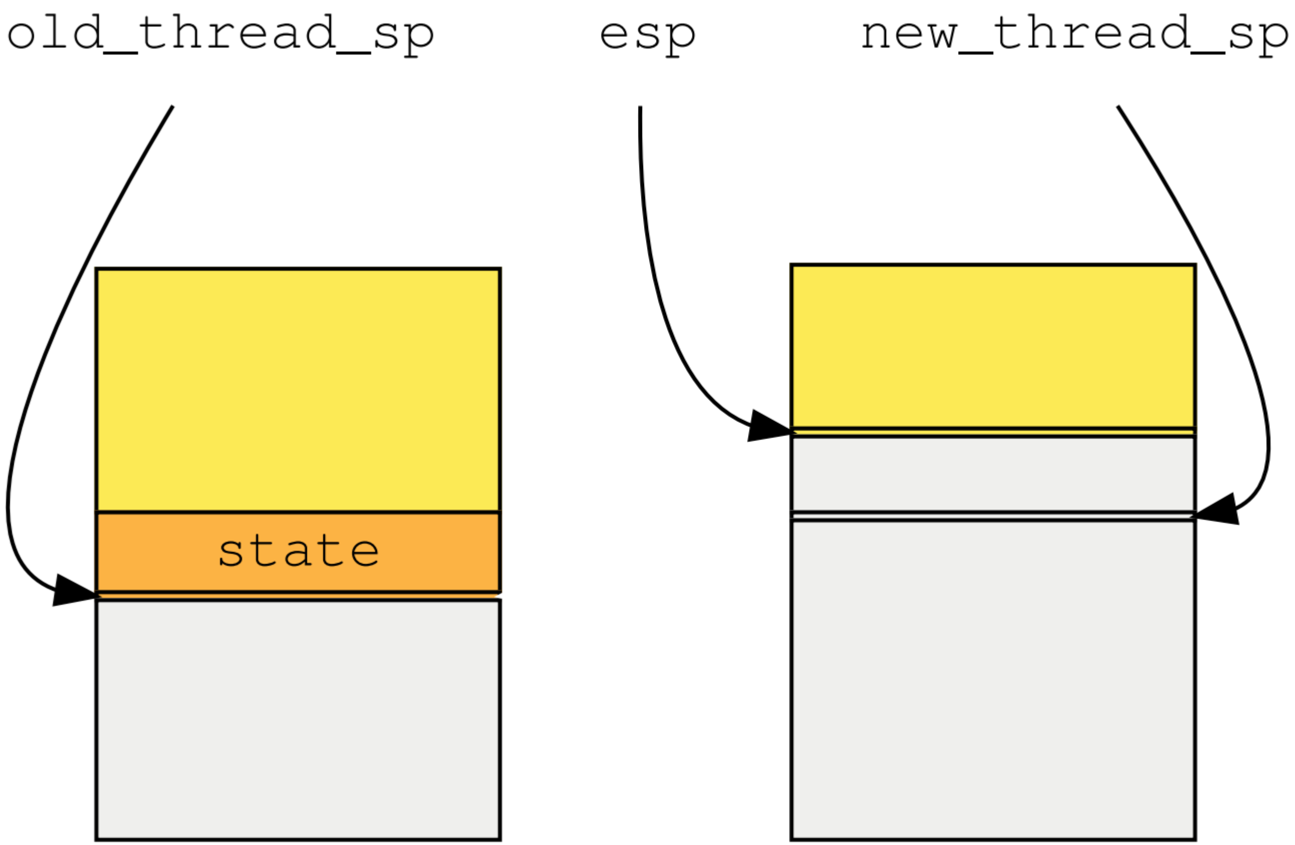
We store the current thread’s state on the current stack, so it is save to switch.

minithread_switch



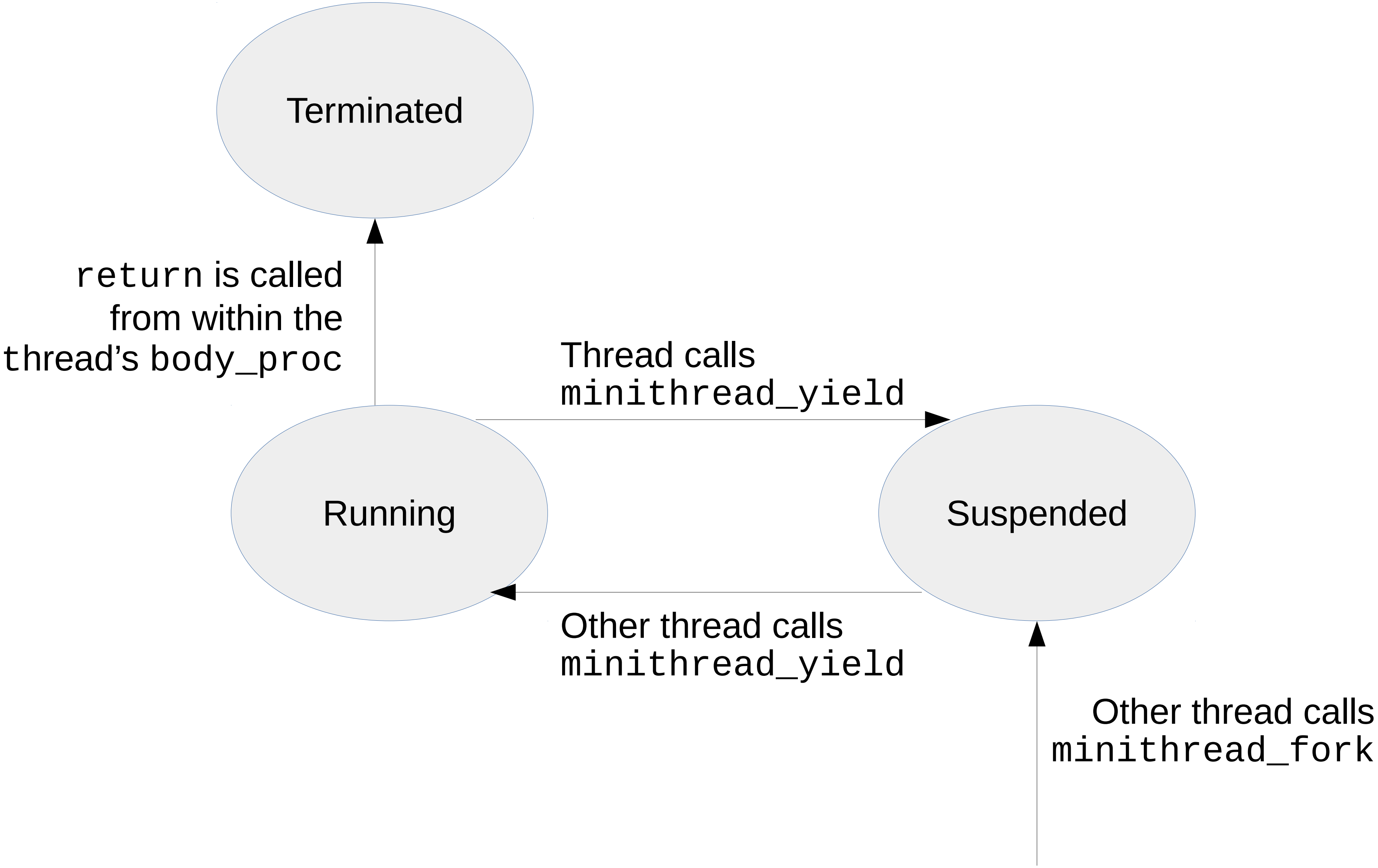
Now we can move the stack pointer to the new thread’s stack now.

minithread_switch



We now restore the thread’s state by reading it from the stack.

Life of a minithread



Terminated

`return` is called from within the thread's `body_proc`

Thread calls `minithread_yield`

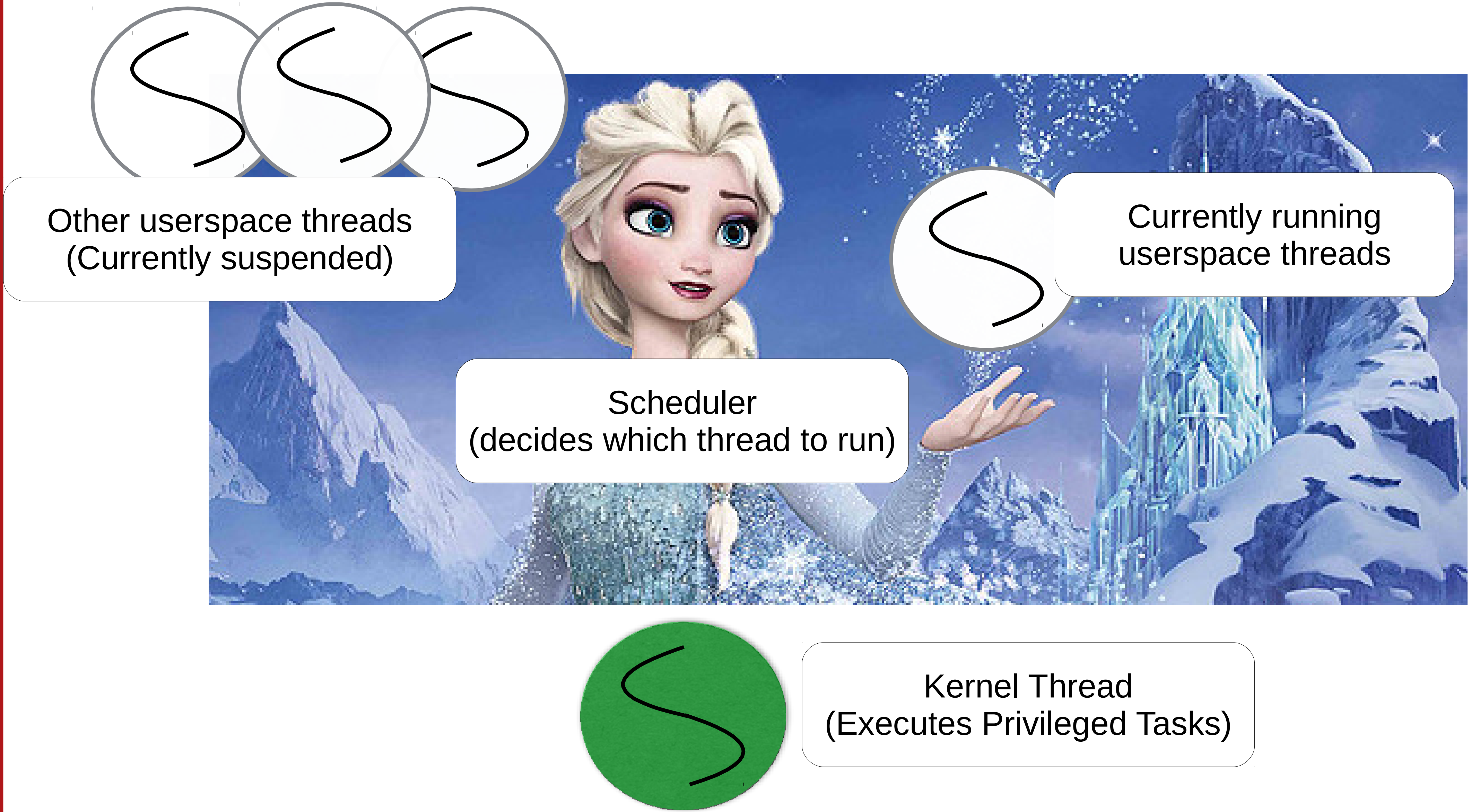
Running

Suspended

Other thread calls `minithread_yield`

Other thread calls `minithread_fork`

The Scheduler in a Nutshell



Other userspace threads (Currently suspended)

Currently running userspace threads

Scheduler (decides which thread to run)

Kernel Thread (Executes Privileged Tasks)

How to implement the Scheduler

- Store threads that are waiting in a queue
- minithread_yield gives control to thread at the head of the queue
- Expect scheduling to get more complicated in Project 2

  → Code style matters

What if there are no Userspace Threads?

- Operating Systems run “forever”
- Switch to an Idle Thread
  - In our case that is just the kernel thread
  - You can reuse the Stack from the host process → no need to allocate a new stack

Being Non-Preemptive

- What happens when a user thread runs forever?
  - In P1, we let it be!
- Assume that all threads are **good** and voluntarily yield
  - Threads yield by calling minithread_yield

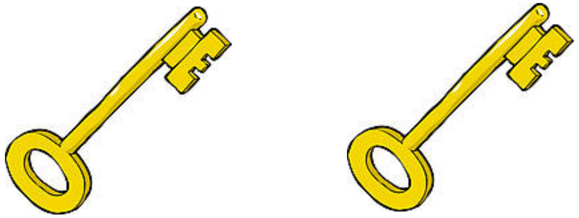
An example for concurrent access.

- Imagine you at a store and need to go to the bathroom.
- There is only a limited number of bathroom keys.
- You need to ask the clerk for a key.
- You are supposed to return the key after you went.

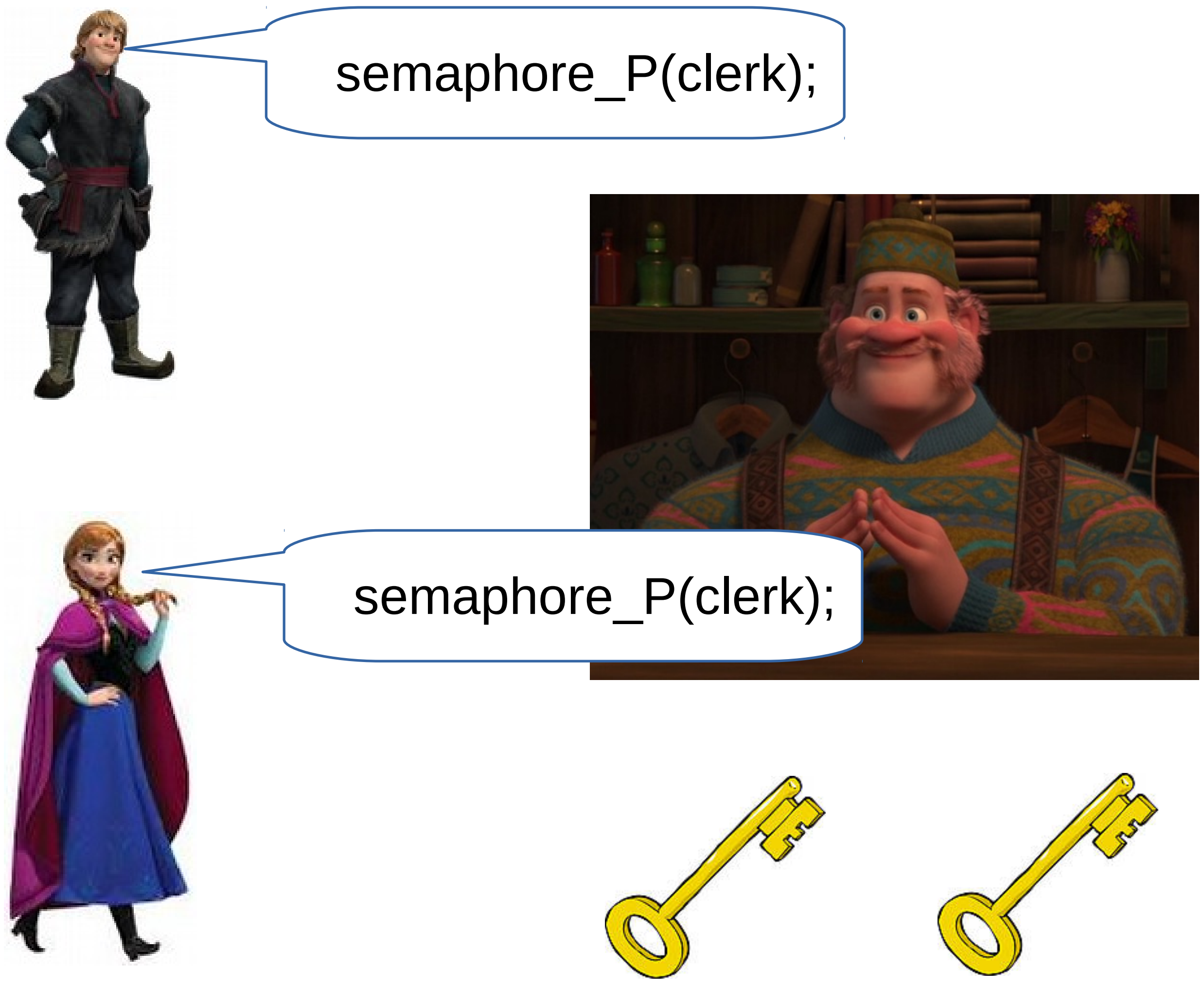
The clerk is a semaphore!

Initially the clerk has 2 keys

```
semaphore_init(clerk, 2);
```



Kristoff and Anna each request a key



```
semaphore_P(clerk);
```

```
semaphore_P(clerk);
```

Now the semaphore count is at 0

Other requests have to wait...

semaphore_P(clerk);

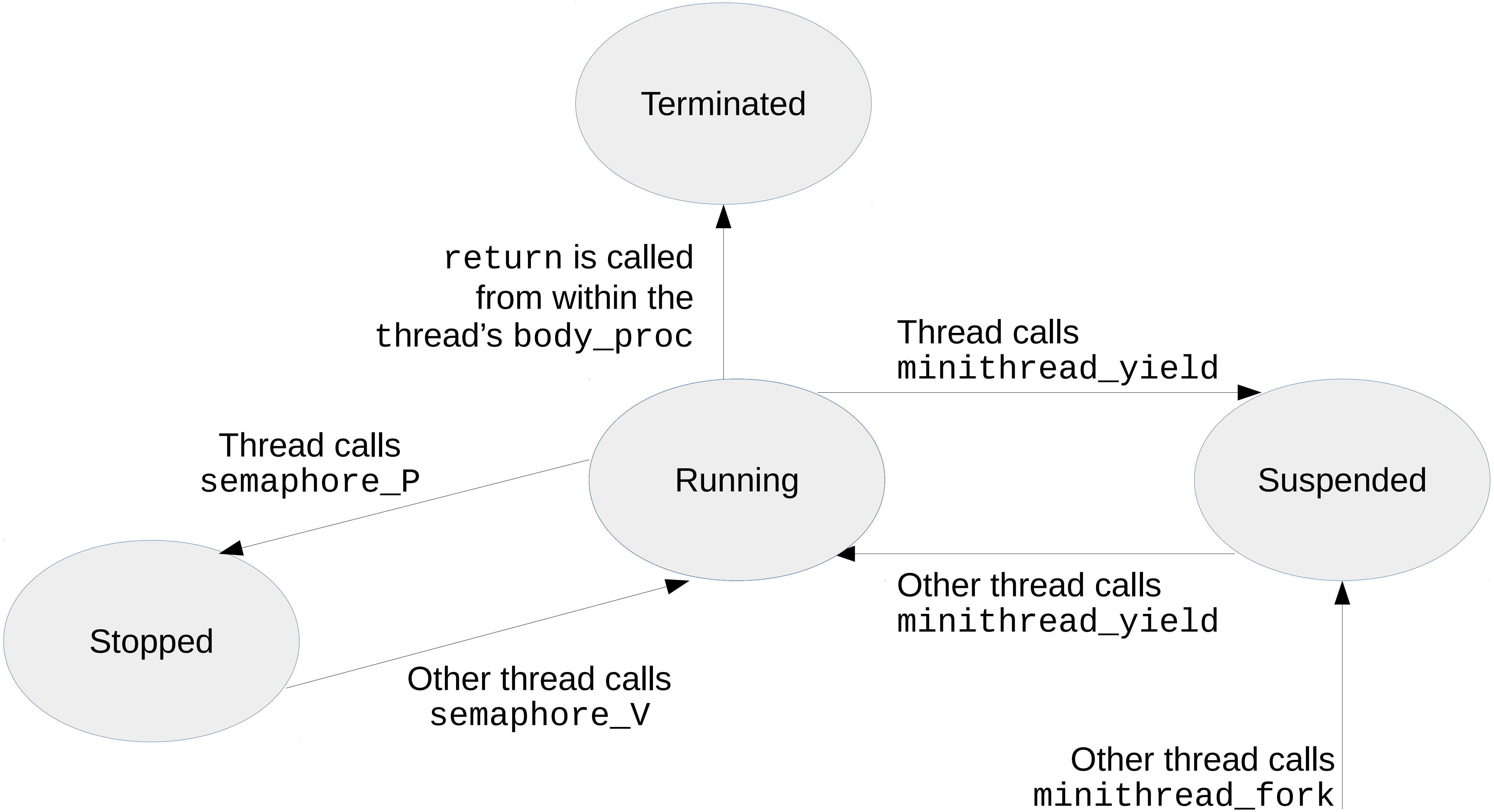
Sorry, I’m out of keys.

...until previous ones are done.

semaphore_V(clerk);

Sven, you can have the key now!

Life of a minithread (extended)



Terminated

`return` is called from within the thread's `body_proc`

Running

Thread calls `minithread_yield`

Suspended

Thread calls `semaphore_P`

Stopped

Other thread calls `semaphore_V`

Other thread calls `minithread_yield`

Other thread calls `minithread_fork`

Putting it all together

```
void minithread_system_initialize
```

- This bootstraps the system
- Use it to initialize queues, semaphores, global variables or data structures
- You will add more in projects to come

Files you need to change

- queue.c/h
- synch.c/h
- minithread.c/h

Comments are good, polling is not.

```
// Polling because CPUs like to be busy
while(!some_condition) {
  check_condition();
}
```

If you comment your code, we can give you partial credit easier!

More Code Style Tips

- Avoid using duplicate code
- Remove **ALL** of your print statements and dead code before submission!
- Comments should explain WHY not WHAT.
- Avoid using duplicate code

Testing

- We supply a few primitive tests
  - Use it to see how minithreads work
- Sieve and buffer are good stress tests
- GDB is your friend!

Questions?

- As always, come to office hours and/or ask on Piazza.
- Projects always look easier as they are
  → Make sure you start early

(Sorry for all the Frozen references ☺)