Processes & Threads
(Chapters 3-6)

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]

Processes!

What is a Program?

Program is a file containing:

- executable code (machine instructions)
- data (information manipulated by these instructions)

that together describe a computation

- Resides on disk
- Obtained via compilation & linking

What is a Process?

- An instance of a program
- An abstraction of a computer:

Address Space + Execution Context + Environment

A good abstraction:

- is portable and hides implementation details
- has an intuitive and easy-to-use interface
- can be instantiated many times
- is efficient and reasonably easy to implement

Process != Program

A program is passive:

code + data

A process is ***alive:***

code + data + stack + registers + PC…

Same program can be run multiple time simultaneously. (1 program, 2 processes)

```
> ./bestprogram &
> ./bestprogram &
```

CPU runs each process directly

But *somehow* each process has its own:

- Registers
- Memory
- I/O resources
- “thread of control”

Process Control Block (PCB)

For each process, the OS has a PCB containing:

- location in memory
- location of executable on disk
- which user is executing this process
- process identifier (`pid`)
- process status (ready, waiting, finished, *etc.*)
- scheduling information
- kernel SP (points in interrupt stack)
  - interrupt stack contains saved process registers
- *… and more!*

System Call Interface

Skinny! (why?)

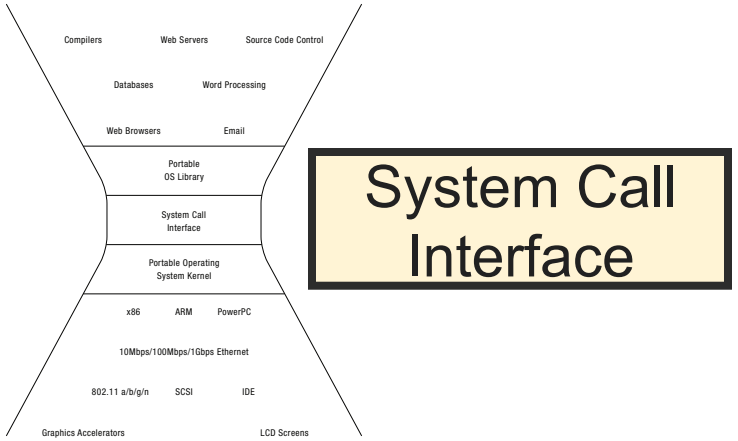
Example:
Creating a Process

Windows:

CreateProcess(…);

UNIX

fork + exec



`CreateProcess` (Simplified)

System Call:

```
if (!CreateProcess(
    NULL,  // No module name (use command line)
    argv[1],// Command line
    NULL,  // Process handle not inheritable
    NULL,  // Thread handle not inheritable
    FALSE, // Set handle inheritance to FALSE
    0,     // No creation flags
    NULL,  // Use parent's environment block
    NULL,  // Use parent's starting directory
    &si,   // Pointer to STARTUPINFO structure
    &pi )  // Ptr to PROCESS_INFORMATION structure
)
```

[Windows]

Beginning a Process via `CreateProcess`

Kernel has to:

- Allocate ProcessID
- Create & initialize PCB in the kernel
- Create and initialize a new address space
- Load the program into the address space
- Copy arguments into memory in address space
- Initialize h/w context to start execution at "start"
- Inform scheduler that new process is ready to run

[Windows]

~~CreateProcess~~ (Simplified)
fork (actual form)

System Call:

```
int pid = fork(  void ☺
    NULL,  // No module name (use command line)
    argv[1],// Command line
    NULL,  // Process handle not inheritable
    NULL,  // Thread handle not inheritable
    FALSE, // Set handle inheritance to FALSE
    0,     // No creation flags
    NULL,  // Use parent's environment block
    NULL,  // Use parent's starting directory
    &si,   // Pointer to STARTUPINFO structure
    &pi )
)
```

[UNIX]

Beginning a Process via ~~`CreateProcess`~~ `fork()`

Kernel has to:

- Allocate ProcessID
- Create & initialize PCB in the kernel
- Create ~~and initialize~~ a new address space
- ~~Load the program into the address space~~
- ~~Copy arguments into memory in address space~~
- Initialize the address space with a copy of the entire contents of the address space of the parent
- ~~Initialize h/w context to start execution at "start"~~
- Inherit execution context of parent (*e.g.*, open files)
- Inform scheduler that new process is ready to run

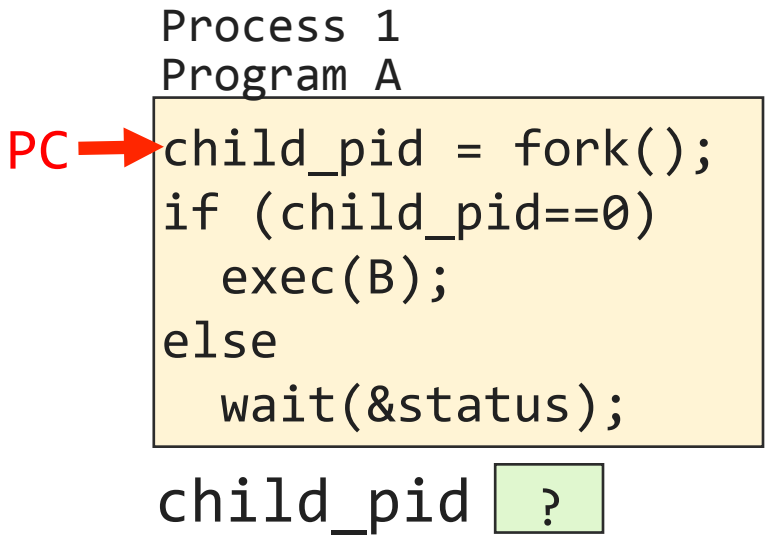
[UNIX]

Creating and Managing Processes

`fork()`	Create a child process as a clone of the current process. Returns to both parent and child. Returns child pid to parent process, 0 to child process.
`exec` `(prog, args)`	Run the application **prog** in the current process with the specified arguments.
`wait` `(&status)`	Pause until some child process has exited.
`exit` `(status)`	Tell the kernel the current process is complete, and its data structures (stack, heap, code) should be garbage collected. Why not necessarily PCB?
`kill` `(pid, type)`	Send an interrupt of a specified type to a process. (a bit of a misnomer, no?)

[UNIX]

Fork + Exec



[UNIX]

Fork + Exec

fork returns twice!

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC →

child_pid 42

Process 42
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC →

child_pid 0

[UNIX]

Fork + Exec

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → `wait(&status);`

child_pid: 42

Process 42
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → `if (child_pid==0)`

child_pid: 0

Waits until child exits.

[UNIX]

Fork + Exec

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → `wait(&status);`

child_pid 42

Process 42
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → `exec(B);`

child_pid 0

***if** and **else** both executed!*

[UNIX]

Fork + Exec

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC → wait(&status);

child_pid 42

Process 42
Program B

PC →

```
main() {
   ...

   exit(3);
}
```

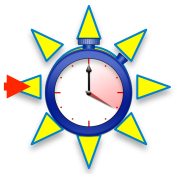
[UNIX]

Fork + Exec

Process 1
Program A

```
child_pid = fork();
if (child_pid==0)
  exec(B);
else
  wait(&status);
```

PC



child_pid 42

status 3

[UNIX]

Code example (`fork.c`)

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    int child_pid = fork();

    if (child_pid == 0) {          // child process
        printf("I am process %d\n", getpid());
    }
    else {                         // parent process.
        printf("I am the parent of process %d\n", child_pid);
    }
    return 0;
}
```

Possible outputs?

What is a Shell?

Job control system

- runs programs on behalf of the user
- allows programmer to create/manage programs

- sh — Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- csh — BSD Unix C shell (tcsh: enhanced csh at CMU and elsewhere)
- bash — “Bourne-Again” Shell

Runs at user-level. Uses syscalls: fork, exec, etc.

Built-In UNIX Shell Commands

jobs	List all jobs running in the background + all stopped jobs.
bg <job>	Run the application prog in the current process.
fg <job>	Change a stopped or running background job to a running in the foreground.
kill <job>	Terminate a job.

Signals (virtualized interrupt)

Allow applications to behave like operating systems.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal (e.g. ctrl-z from keyboard)

[UNIX]

Sending a Signal

Kernel delivers a signal to a destination process

For one of the following reasons:

- Kernel detected a system event (*e.g.*, div-by-zero (SIGFPE) or termination of a child (SIGCHLD))
- A process invoked the **kill system call** requesting kernel to send signal to a process
    - debugging
    - suspension
    - resumption
    - timer expiration

Receiving a Signal

A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.

Three possible ways to react:

1. Ignore the signal (do nothing)
2. Terminate process (+ optional core dump)
3. Catch the signal by executing a user-level function called signal handler
   - Like a hardware exception handler being called in response to an asynchronous interrupt

Signal Example

```
int main() {
   pid_t pid[N];
   int i, child_status;

   for (i = 0; i < N; i++) // N forks
       if ((pid[i] = fork()) == 0) {
          while(1); //child infinite loop
       }
   /* Parent terminates the child processes */
   for (i = 0; i < N; i++) {    // parent continues executing
       printf("Killing proc. %d\n", pid[i]);
       kill(pid[i], SIGINT);
   }
   /* Parent reaps terminated children */
   for (i = 0; i < N; i++) {
       pid_t wpid = wait(&child_status);
       if (WIFEXITED(child_status)) // parent checks for each child's exit
           printf("Child %d terminated w/exit status %d\n", wpid,
                  WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
   }
   exit(0);
}
```

Handler Example

```
void int_handler(int sig) {
   printf("Process %d received signal %d\n", getpid(), sig);
   exit(0);
}
int main() {
   pid_t pid[N];
   int i, child_status;
   signal(SIGINT, int_handler); //register handler for SIGINT
   for (i = 0; i < N; i++) // N forks
       if ((pid[i] = fork()) == 0) {
          while(1); //child infinite loop
       }
   for (i = 0; i < N; i++) {     // parent continues executing
       printf("Killing proc. %d\n", pid[i]);
       kill(pid[i], SIGINT);
   }
   for (i = 0; i < N; i++) {
       pid_t wpid = wait(&child_status);
       if (WIFEXITED(child_status)) // parent checks for each child's exit
           printf("Child %d terminated w/exit status %d\n", wpid,
           WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
   }
   exit(0);
}
```

Threads! (Chapters 25-27)

Other terms for threads:

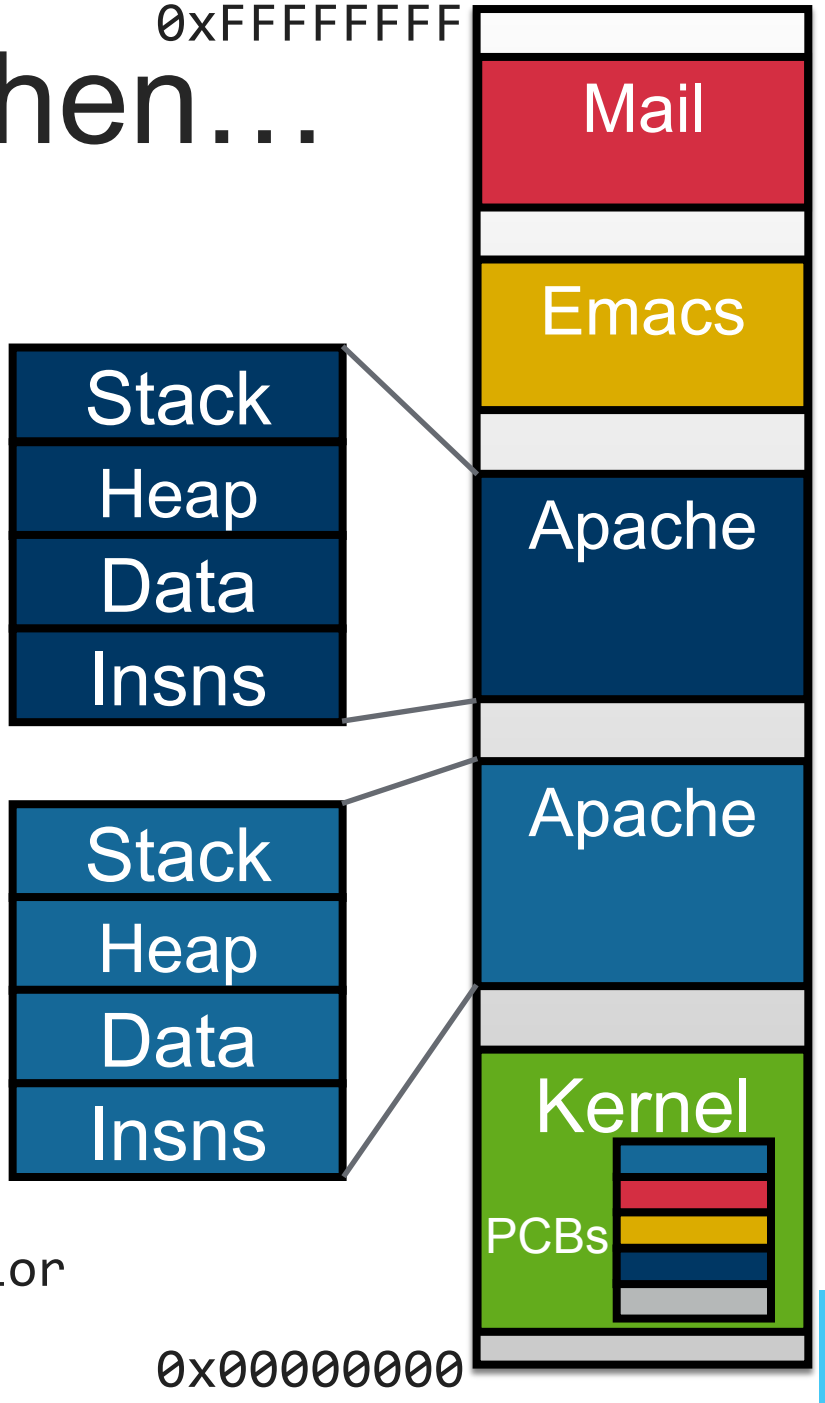
- Lightweight Process
- Thread of Control
- Task

What happens when…

Apache wants to run multiple concurrent computations?

Two heavyweight address spaces for two concurrent computations?

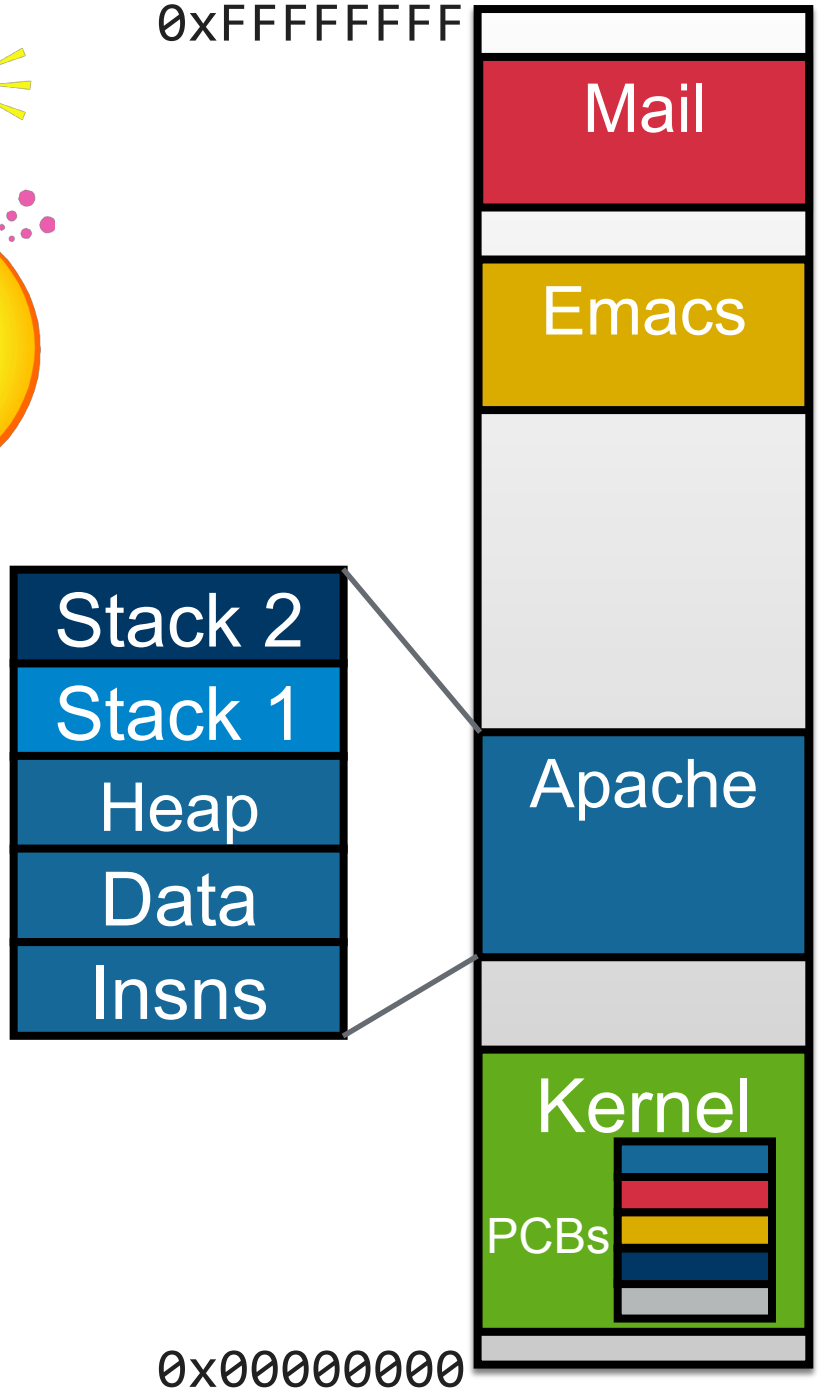
What is distinct about these address spaces?



Physical address space
Each process' address space by color
(shown contiguous to look nicer)

Idea

Place concurrent computations in the same address space!

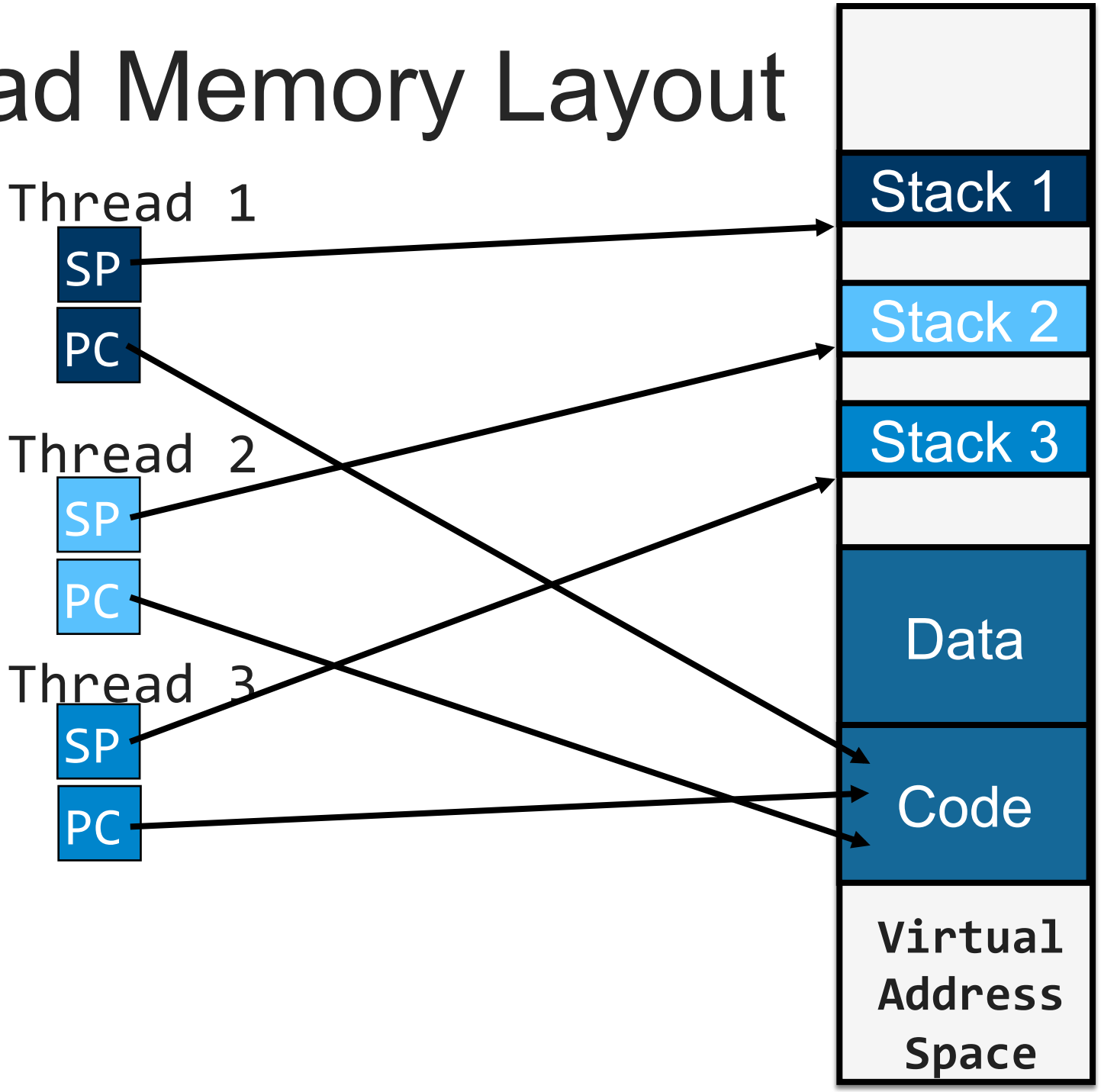


Process vs. Thread

Process:

- Privilege Level
- Address Space
- Code, Data, Heap
- Shared I/O resources
- One or more Threads:
  - Stack
  - Registers
  - PC, SP

Thread Memory Layout



Processes and Threads

Process abstraction combines two concepts

- **Concurrency:** each process is a sequential execution stream of instructions
- **Protection:** Each process has own address space

Threads decouple concurrency & protection

- A **thread** represents a sequential execution stream of instructions.
- A **process** defines the address space that may be shared by multiple threads
- Threads must be mutually trusting. Why?

Thread: abstraction for concurrency

A single-execution stream of instructions; represents a separately schedulable task

- OS can run, suspend, resume it at any time
- bound to a process

Virtualizes the processor

- programs run on machine with an infinite number of processors (*hint: not true!*)

Process: abstraction of computer

- Consists of a virtual memory + a set of threads (= virtual cores)
  - The virtual memory is implemented by space-partitioning physical memory and MMU registers (such as base + size registers)
  - Threads are implemented by time-partitioning the underlying CPU and letting threads run *N* at a time (*N* = #physical cores), temporarily saving registers when a

Why Threads?

Performance: exploiting multiple processors

Do threads make sense on a single core?

Encourages natural program structure

- Expressing logically concurrent tasks
- update screen, fetching data, receive user input

Responsiveness

- splitting commands, spawn threads to do work in the background

Mask long latency of I/O devices

- do useful work while waiting

Some Thread Examples

```
for (k = 0; k < n; k++) {
    a[k] = b[k] × c[k] + d[k] × e[k]
}
```

Web server:

1. get network message (URL) from client
2. get URL data from disk
3. compose response
4. send response

Simple Thread API

`void thread_create (thread,func,arg)`	Creates a new thread in **thread**, which will execute function **func** with the arguments **arg**
`void thread_yield()`	Calling thread gives up processor. Scheduler can resume running this thread at any point.
`int thread_join (thread)`	Wait for **thread** to finish, then return the value **thread** passed to thread_exit. May be called only once for each thread.
`void thread_exit (ret)`	Finish caller; store **ret** in caller's TCB and wake up any thread that invoked thread_join(caller).

Implementation of Threads

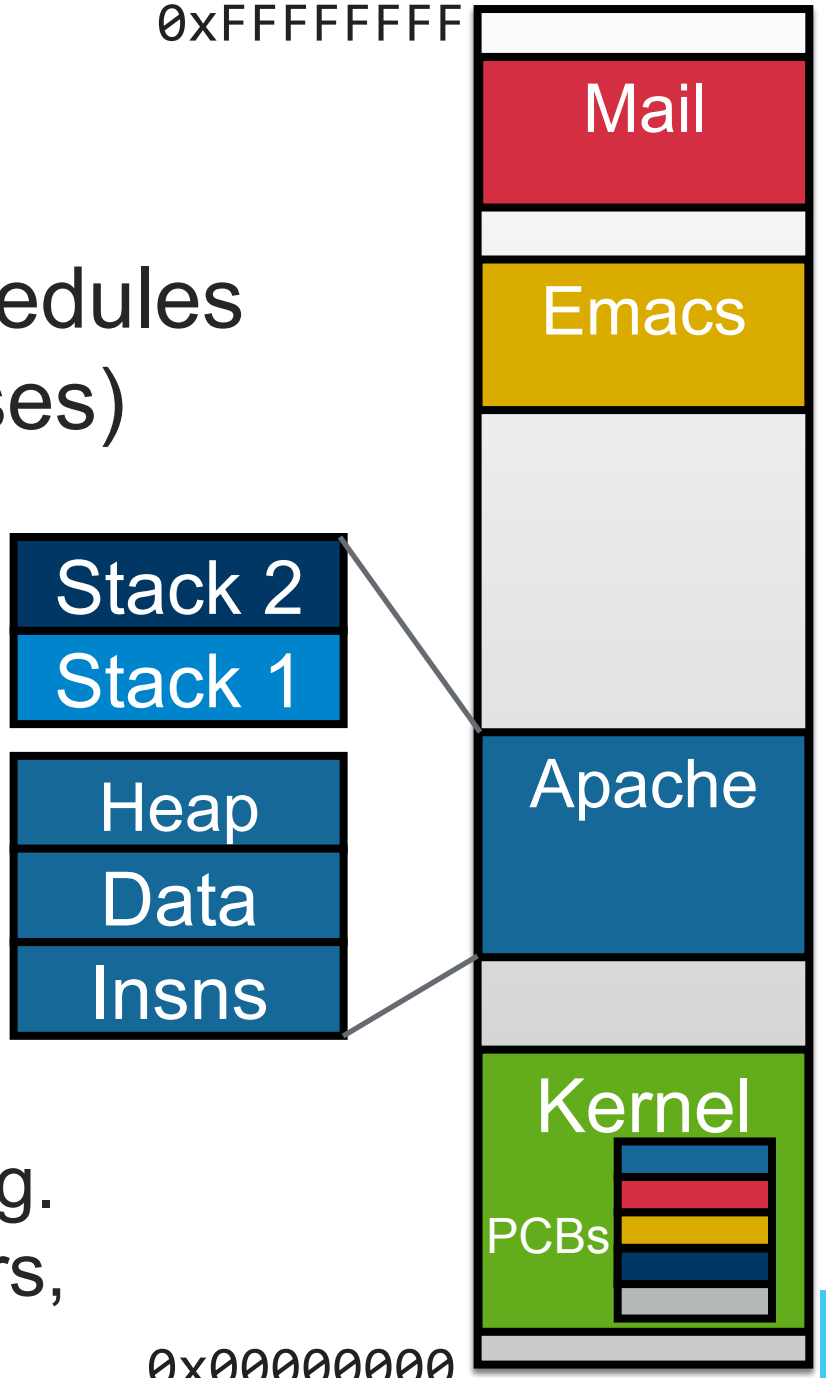
- One abstraction, two implementations:

1. "kernel threads": each thread has its own PCB in the kernel, but the PCBs point to the same physical memory
2. "user threads": one PCB for the process; threads implemented entirely in user space. Each thread has its own Thread Control Block (TCB)

#1: Kernel-Level Threads

Kernel knows about, schedules threads (just like processes)

- Threads share virtual address space
- Separate PCB (**T**CB) for each thread
- PCBs have:
  - **same:** page table base reg.
  - **different:** PC, SP, registers, kernel interrupt stack



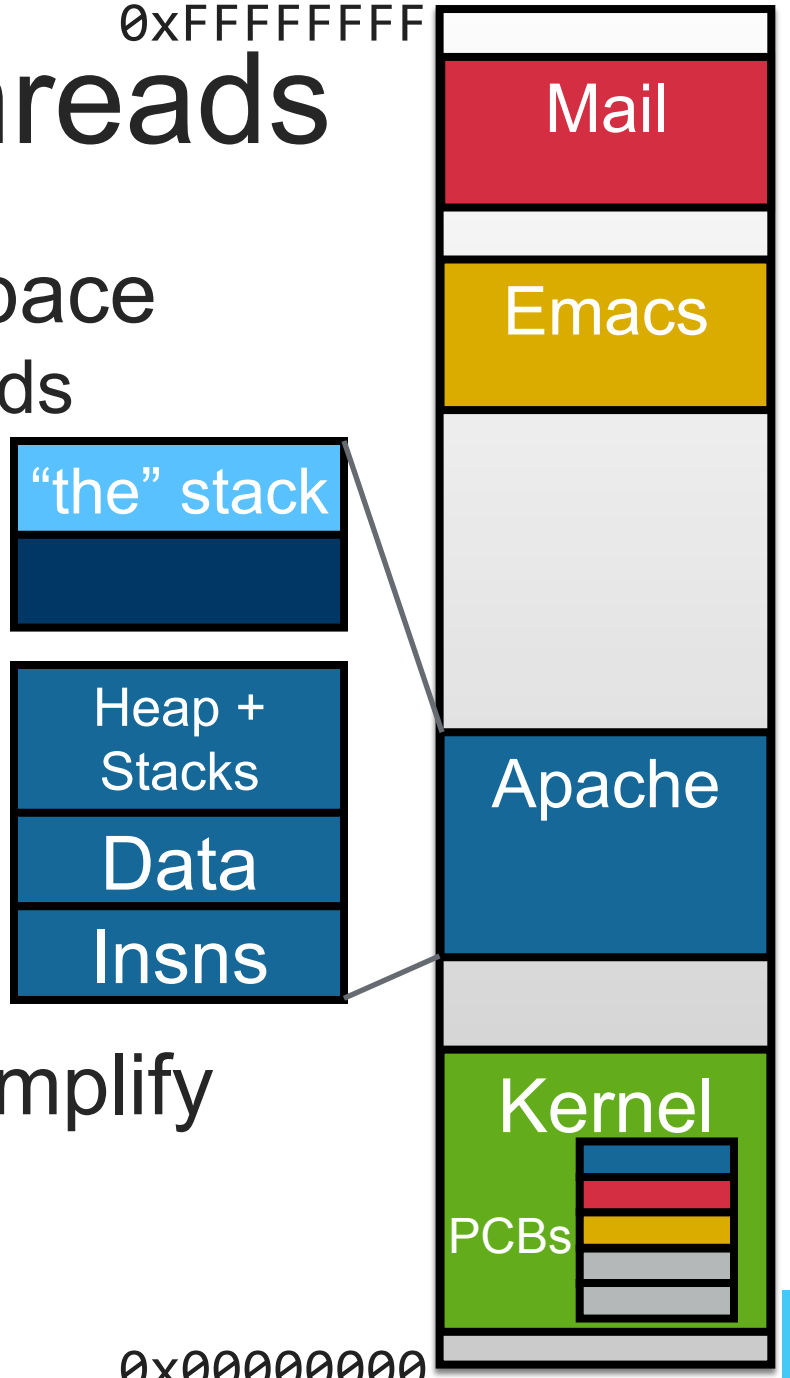
#2: User-Level Threads

Build a mini-OS in user space

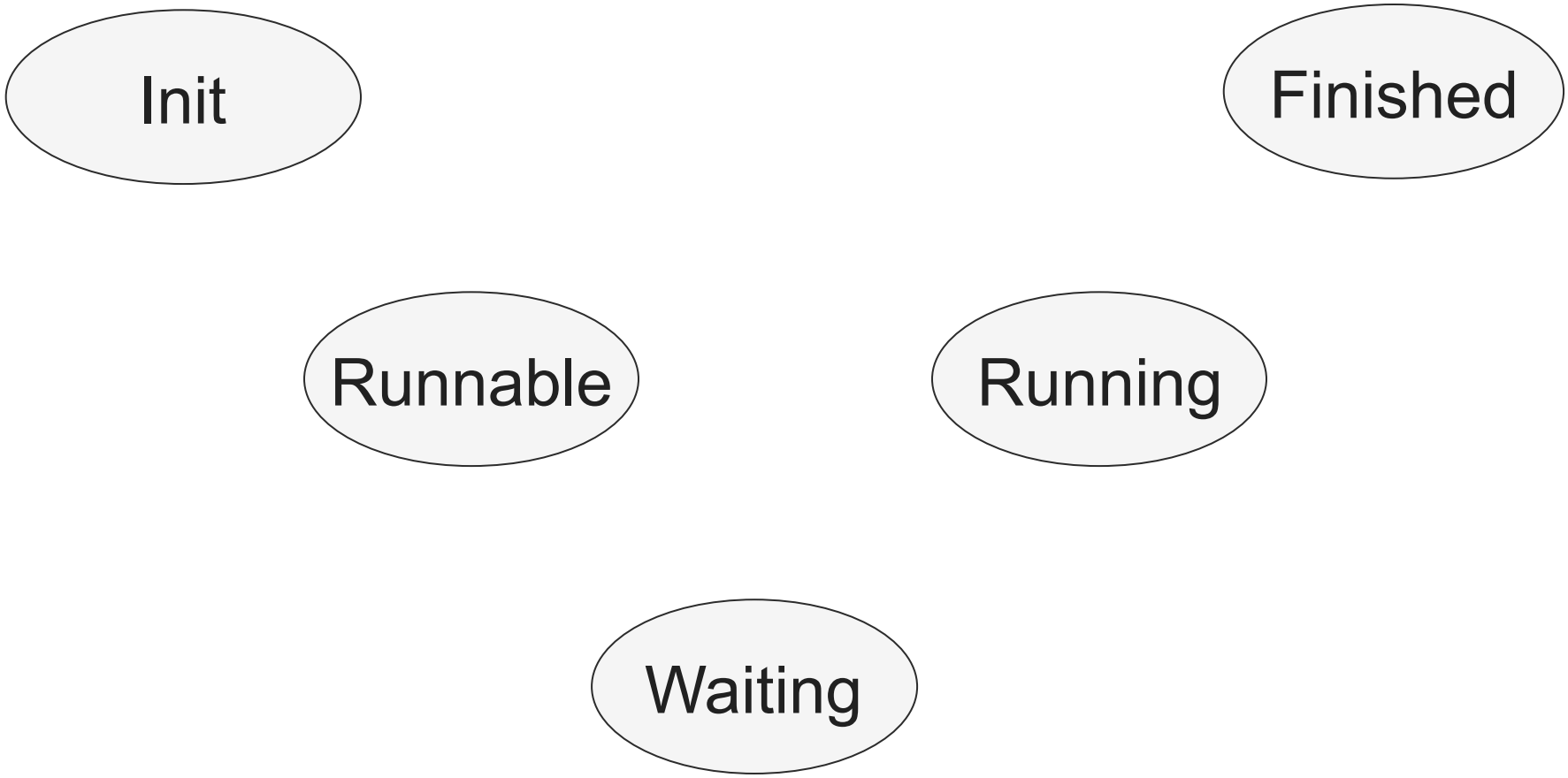
- Real OS unaware of threads
- Single PCB

Generally more efficient than kernel-level threads (Why?)

But kernel-level threads simplify system call handling and scheduling (Why?)

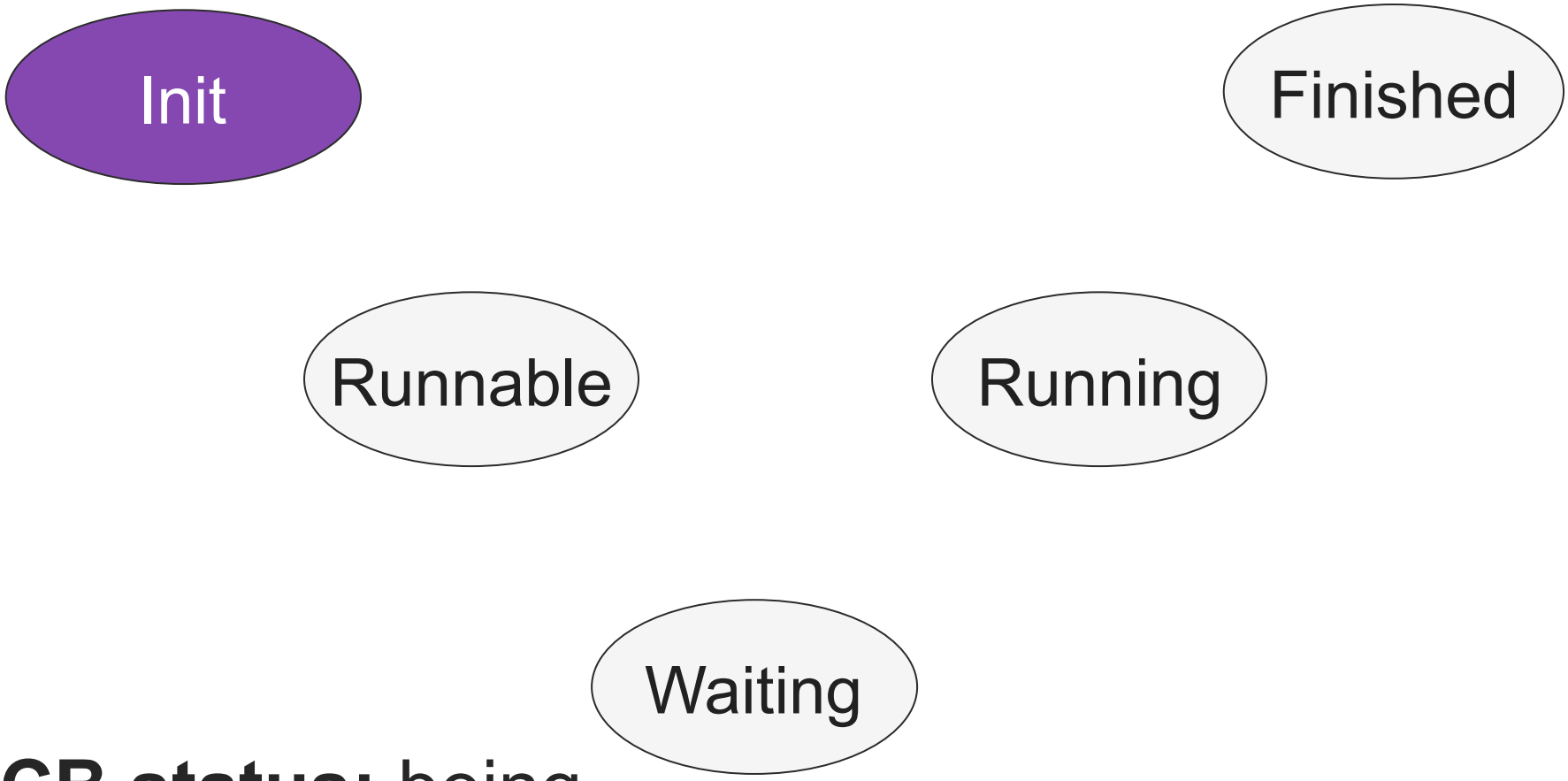


Thread (or Process) Life Cycle



Processes go through these states, too.

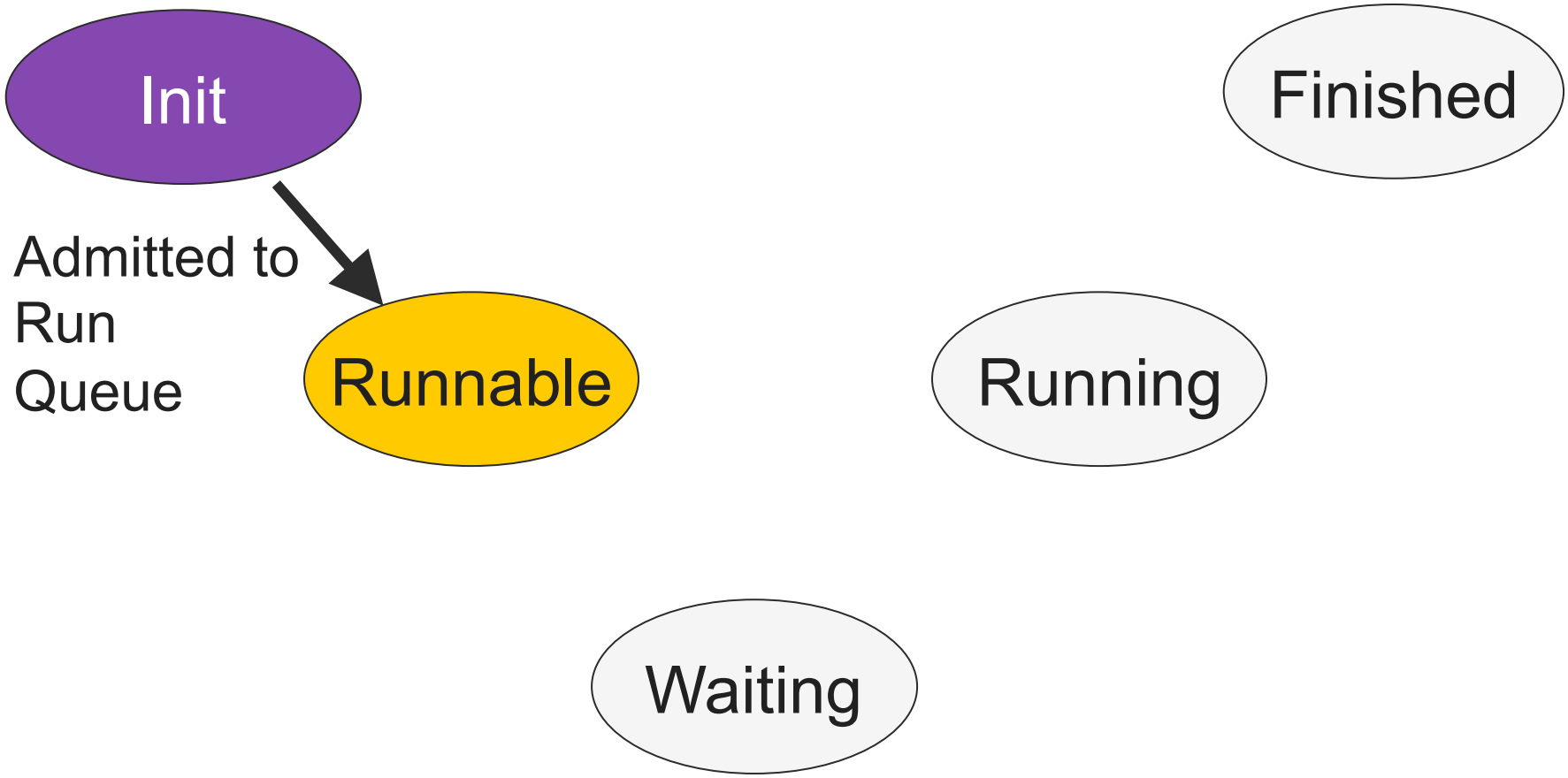
Thread creation



TCB status: being created

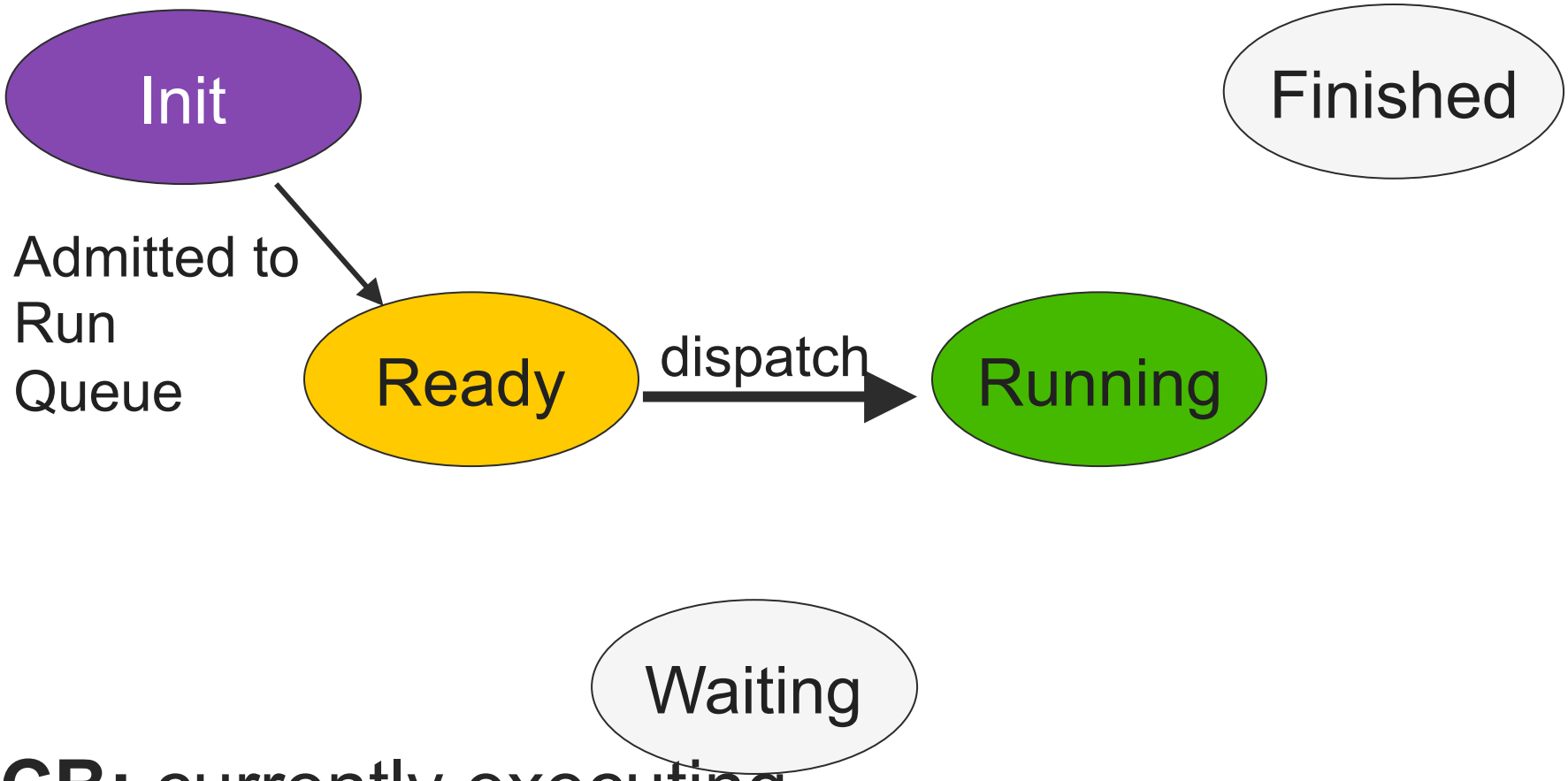
Registers: uninitialized

Thread is Ready to Run



TCB: on Run Queue (aka Ready Queue)
Registers: pushed onto thread's stack

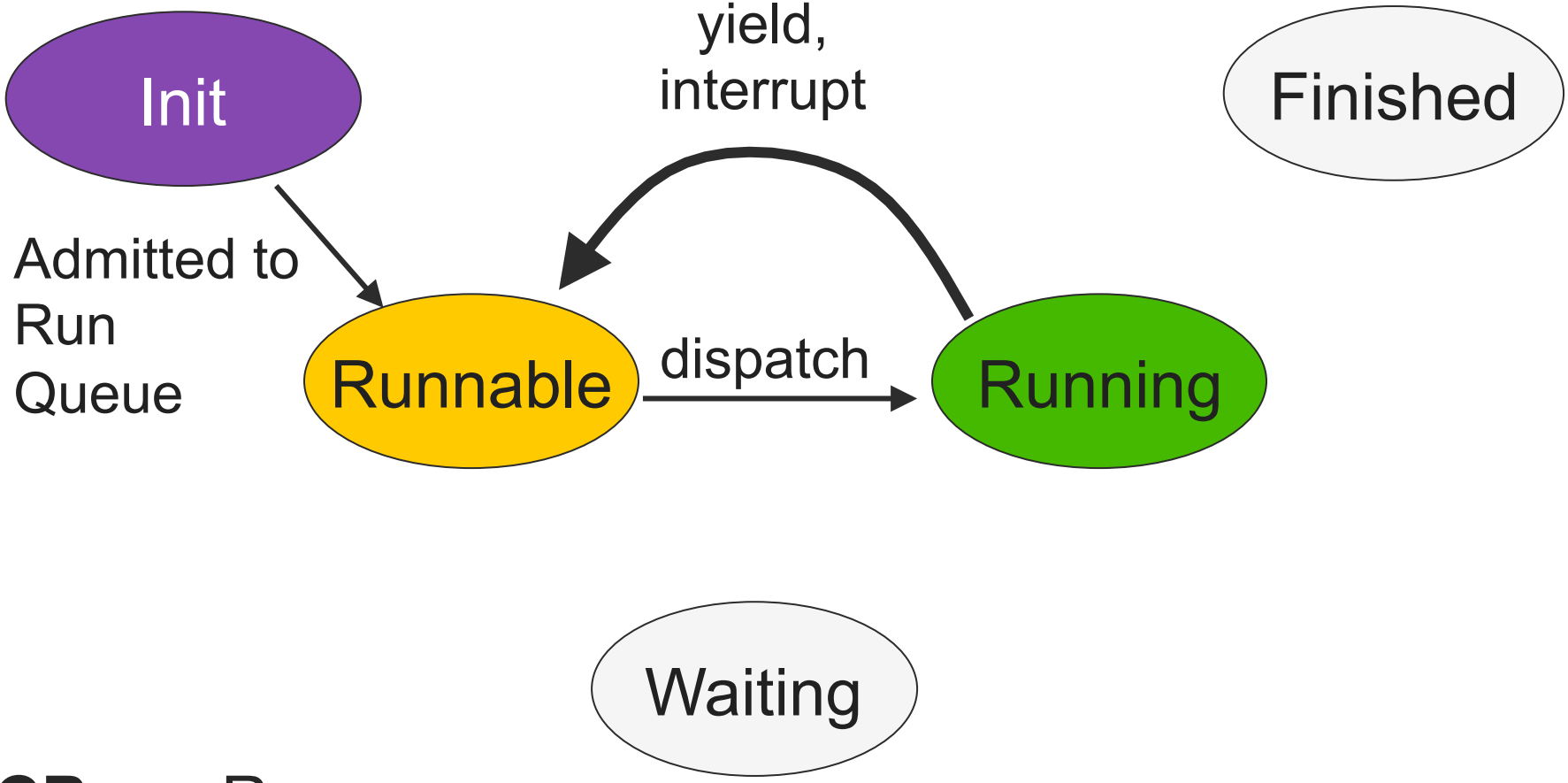
Thread is Running



TCB: currently executing

Registers: popped from thread's stack into CPU

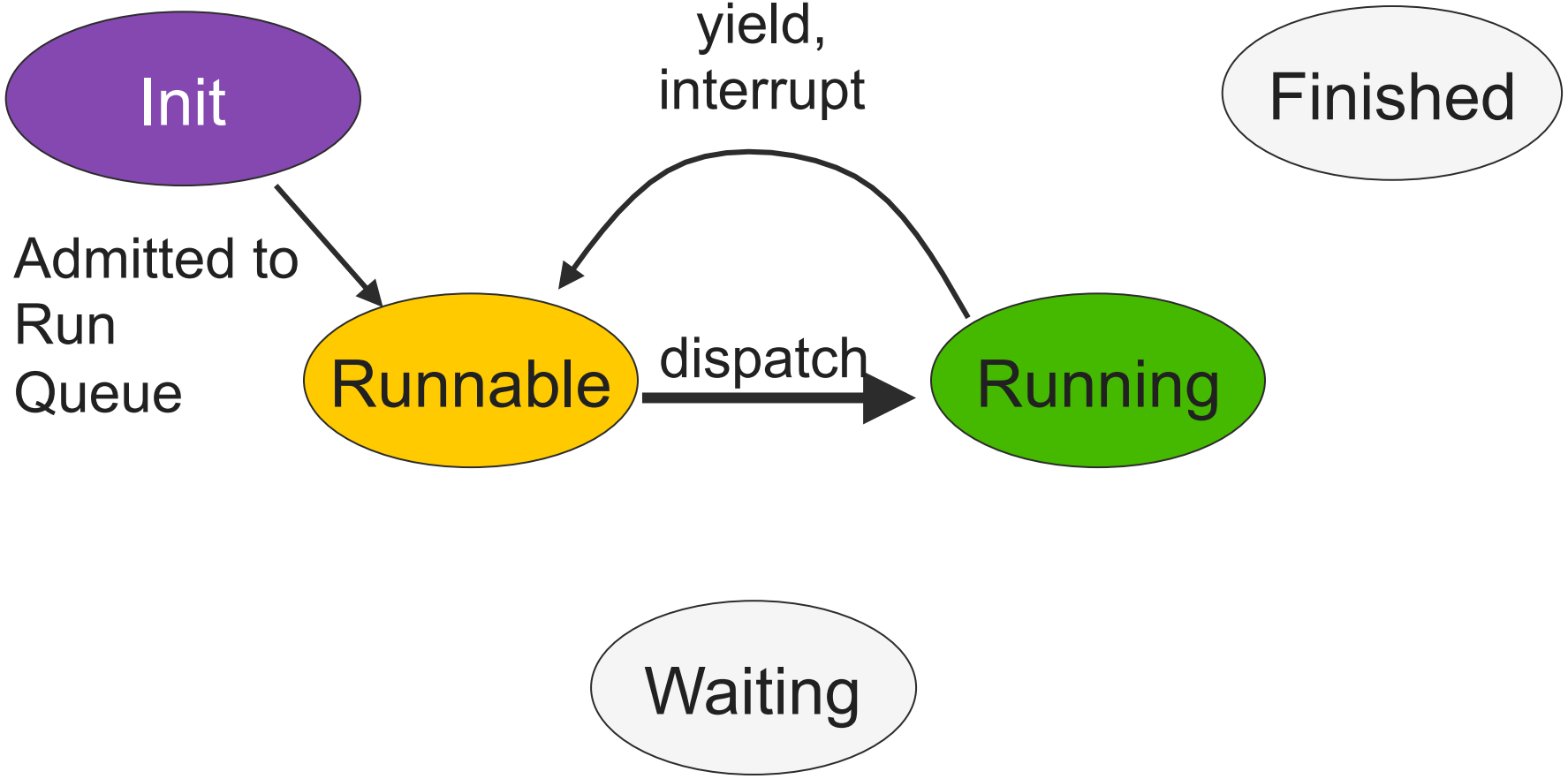
Thread Yields (back to Ready)



TCB: on Run queue

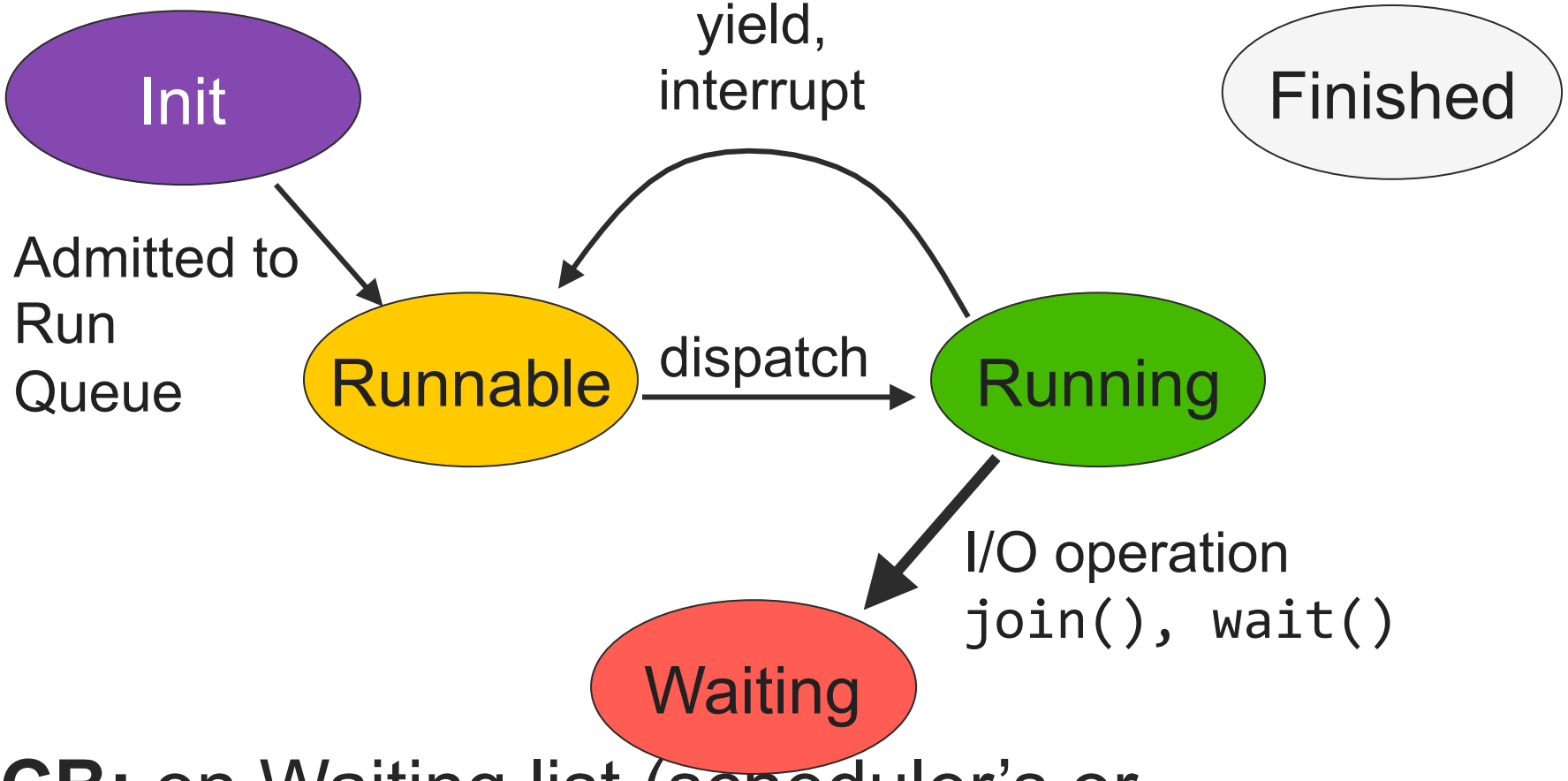
Registers: pushed onto thread's stack (sp in TCB)

Thread is Running Again!



TCB: currently executing
Registers: sp restored from TCB; others restored from stack

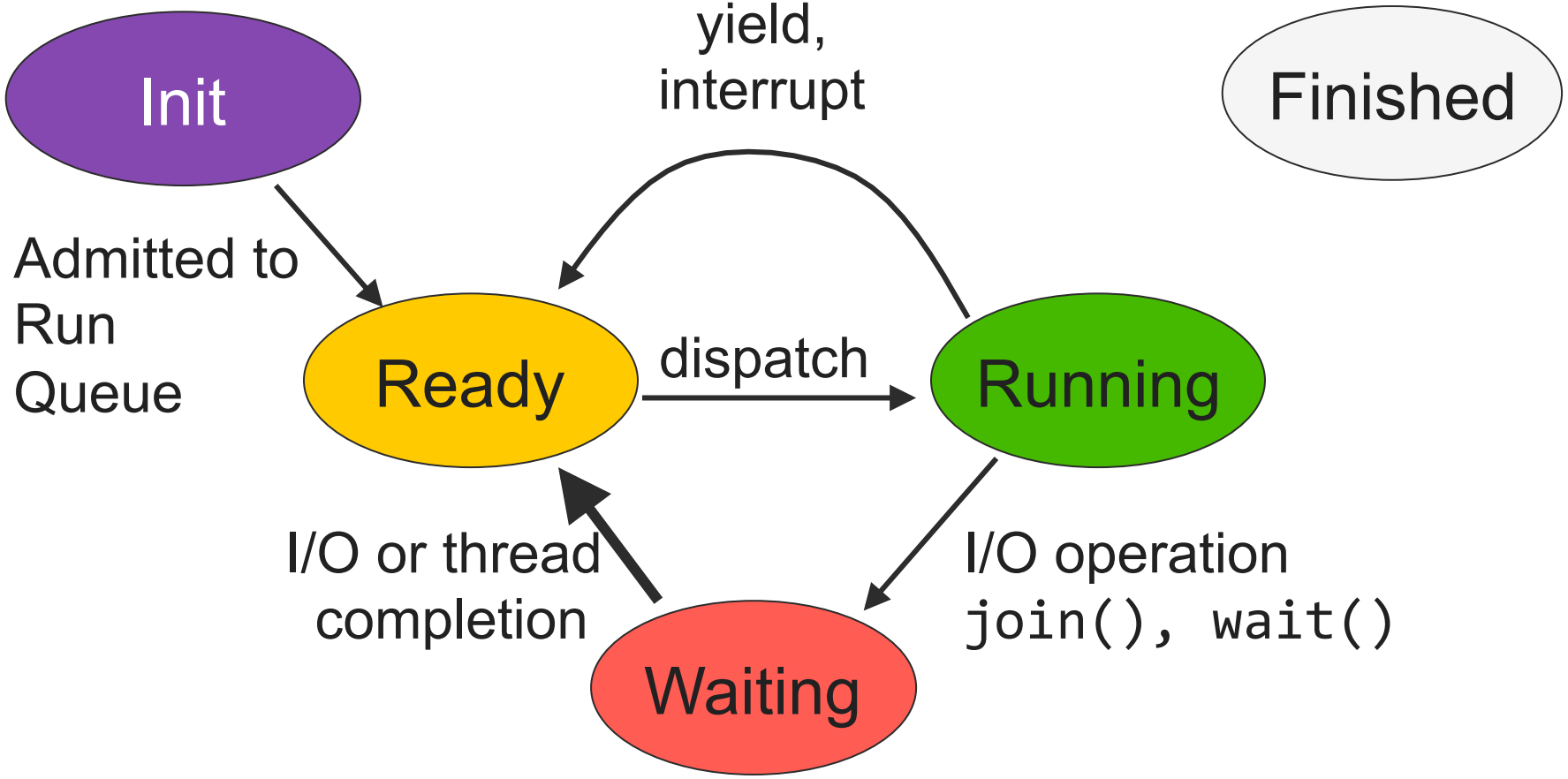
Thread is Waiting



TCB: on Waiting list (scheduler's or other)

Registers: on thread's stack

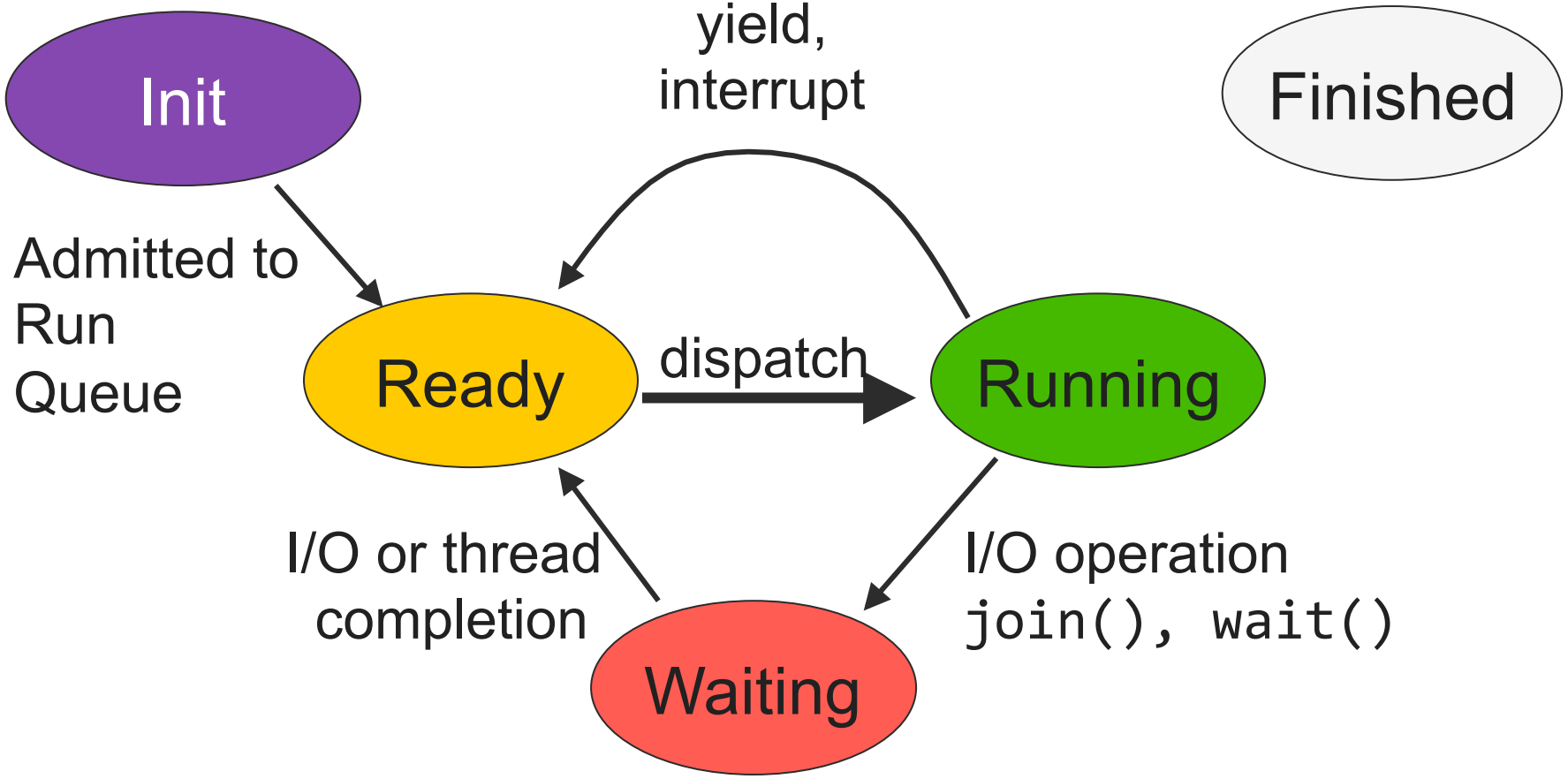
Thread is Ready Again!



TCB: on run queue

Registers: on thread's stack

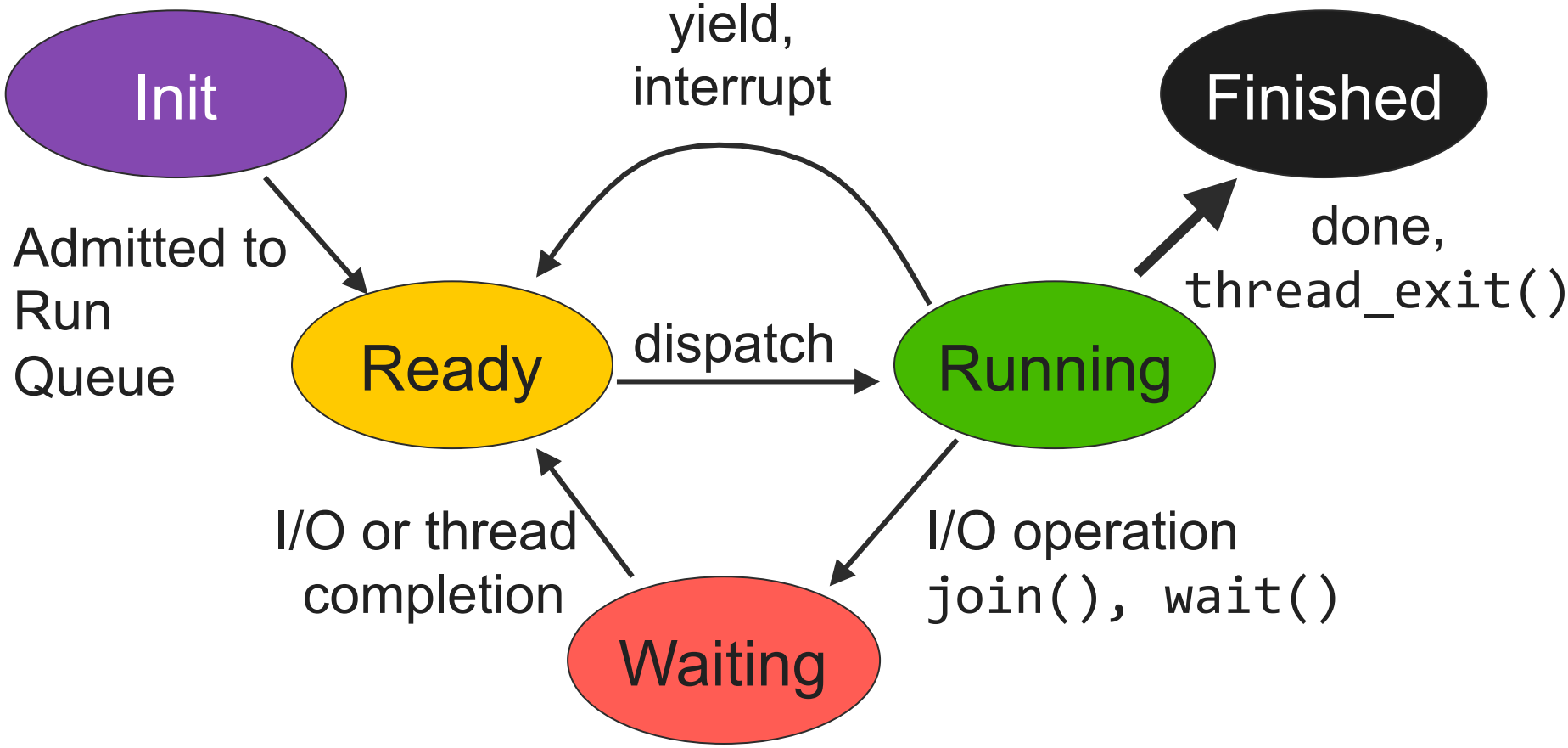
Thread is Running Again!



TCB: currently executing

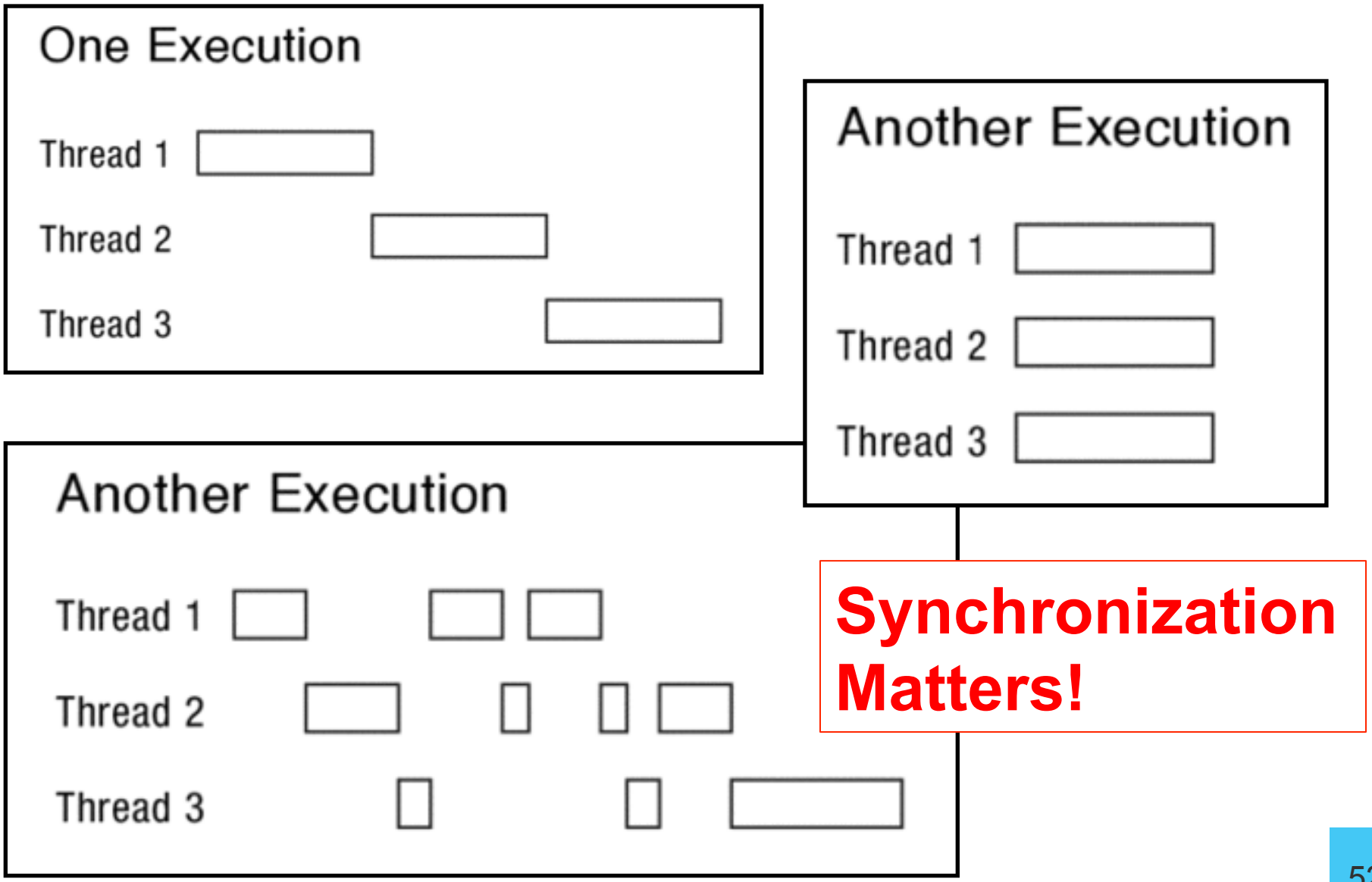
Registers: restored from stack into CPU

Thread is Finished (Process = Zombie)



TCB: on Finished queue, ultimately deleted
Registers: no longer needed

Do **not** presume to know the schedule



Context Switch

- Switching from executing a running thread to runnable thread, exchanging their status
  - save the registers of thread 1 on its stack
  - save the sp of thread 1 in its TCB
  - restore the sp of thread 2 from its TCB
  - restore the registers

ctx_switch(&old_sp, new_sp)

```
ctx_switch:  // ip already pushed!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    popq    %r8
    popq    %r9
    popq    %r10
    popq    %r11
    popq    %r12
    popq    %r13
    popq    %r14
    popq    %r15
    popq    %rbx
    popq    %rbp
    retq
```

```
ctx_start:     // ip already pushed!
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    callq   ctx_entry
```