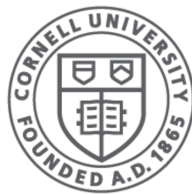
Processes & Threads

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 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 privilege level
- process identifier (`pid`)
- process arguments (for identification with `ps`)
- process status (Ready, waiting, finished, *etc.*)
- register values
- scheduling information
- PC, SP, eflags/status register

… and more!

Usually lives on the kernel stack

Who should be allowed to start a process?

Possibility #1:

Only the kernel may start a process

Possibility #2:

User-level processes may start processes

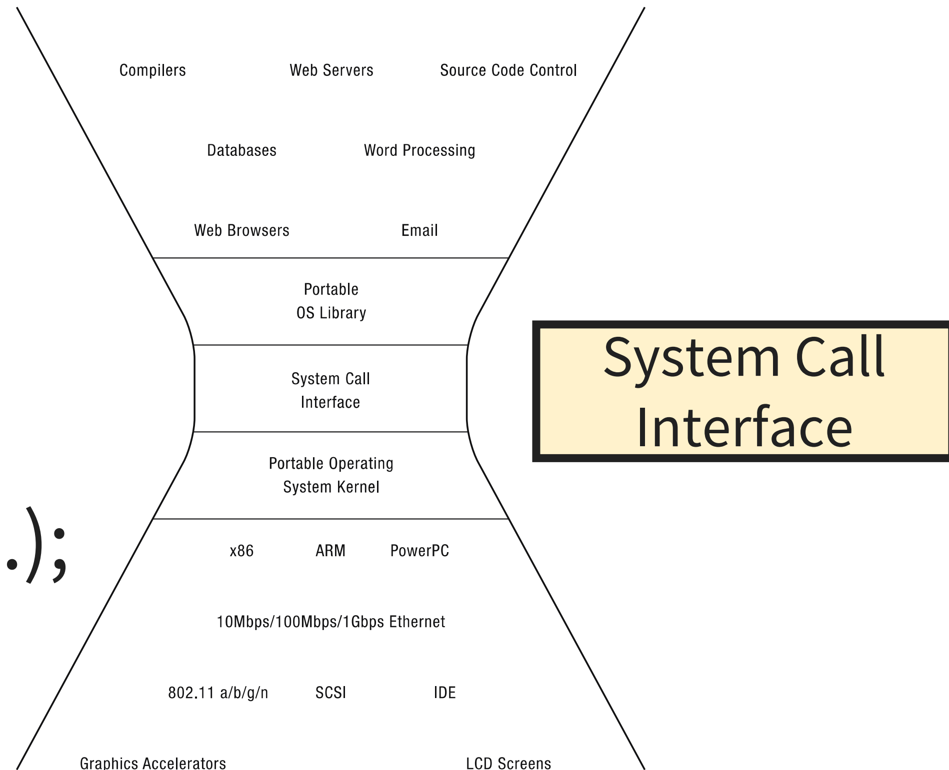
System Call Interface

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 )
)
```

(The argument lines from `NULL,  // No module name` through `&pi )` are struck through on the slide.)

[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``(pid)`	Pause until the child process has exited.
`exit`	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

Process 1
Program A

PC →

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

child_pid ?

Fork + Exec

fork returns twice!

Process 1
Program A

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

PC → if (child_pid==0)

child_pid: 42

Process 42
Program A

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

PC → if (child_pid==0)

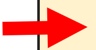
child_pid: 0

[UNIX]

Fork + Exec

Process 1
Program A

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

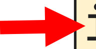


Waits until child exits.

child_pid 42

Process 42
Program A

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



child_pid 0

[UNIX]

Fork + Exec

Process 1
Program A

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

PC → wait(child_pid);

child_pid 42

Process 42
Program A

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

PC → exec(B);

child_pid 0

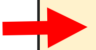
***if** and **else** both executed!*

[UNIX]

Fork + Exec

Process 1
Program A

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



child_pid 42

Process 42
Program B

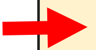
```
main() {
    ...


}
```

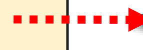
PC

Fork + Exec

Process 1
Program A

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





child_pid 42

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)

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 another 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 (signal.c)

```
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 (handler.c)

```
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);
}
```

Context Switch

- Exchange current running process for another runnable process
- Similar to interrupt handling:
  - Save the registers of process 1 on its stack
  - Save the SP and PC of process 1 in its PCB
  - Restore the SP and PC of process 2 from its PCB
  - Restore the registers of process 2 from its stack
- Also: change virtual memory address space from process 1 to process 2*

(Virtual memory details coming later)

Threads!

Other terms for threads:

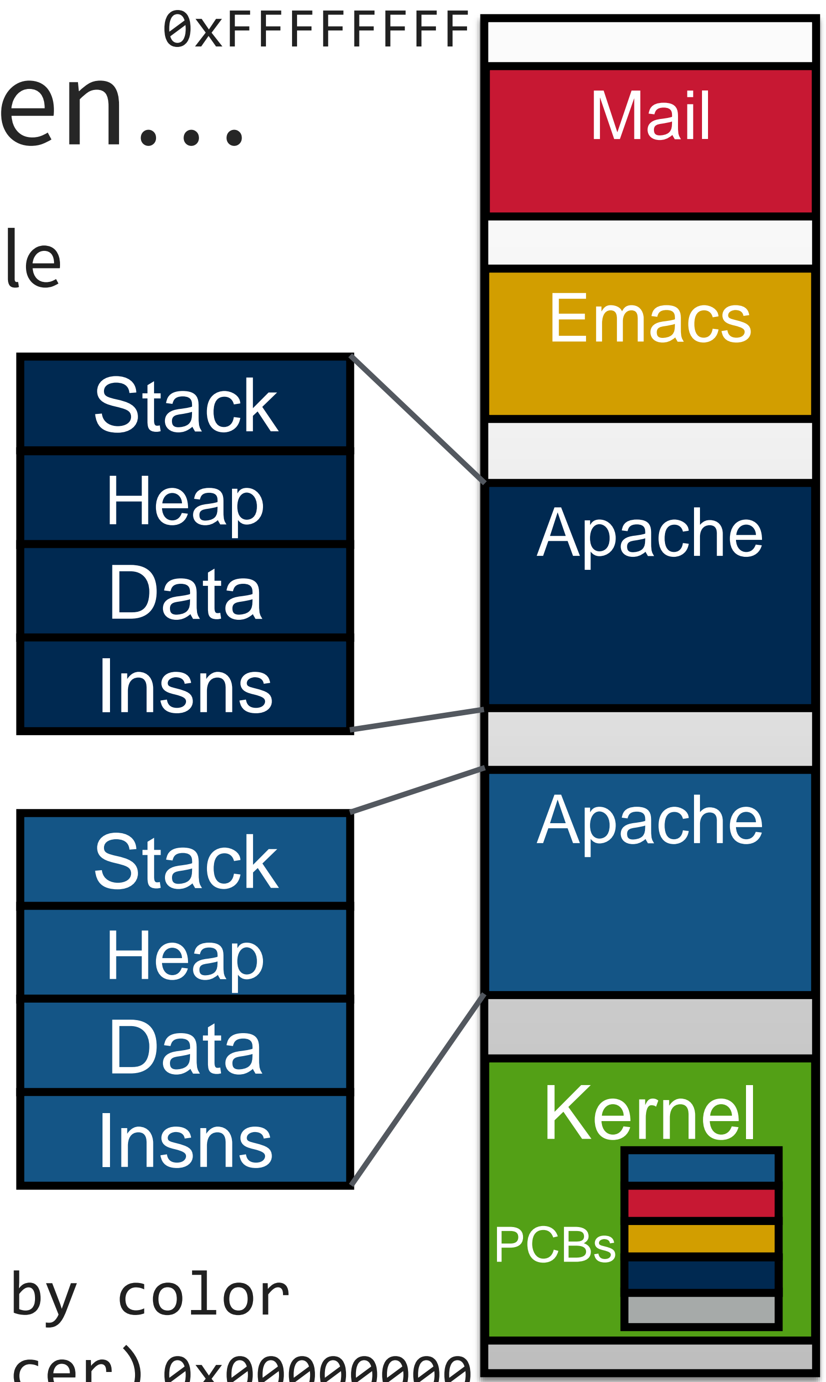
- Lightweight Process
- Thread of Control
- Task

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?

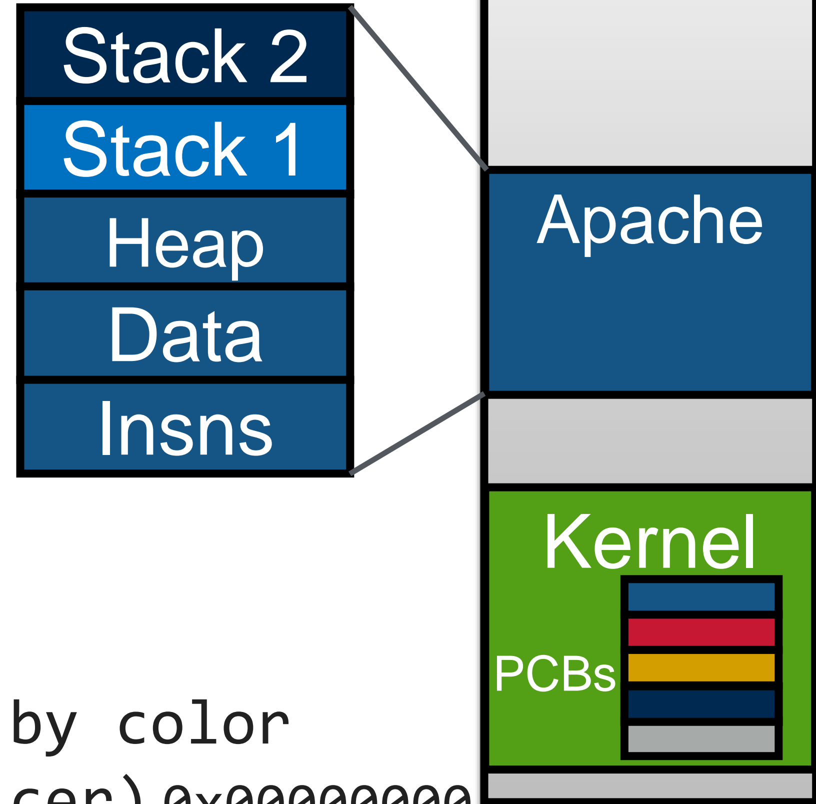


Physical address space
Each process' address space by color
(shown contiguous to look nicer)

Idea

Place concurrent computations in the same address space!

0xFFFFFFFF



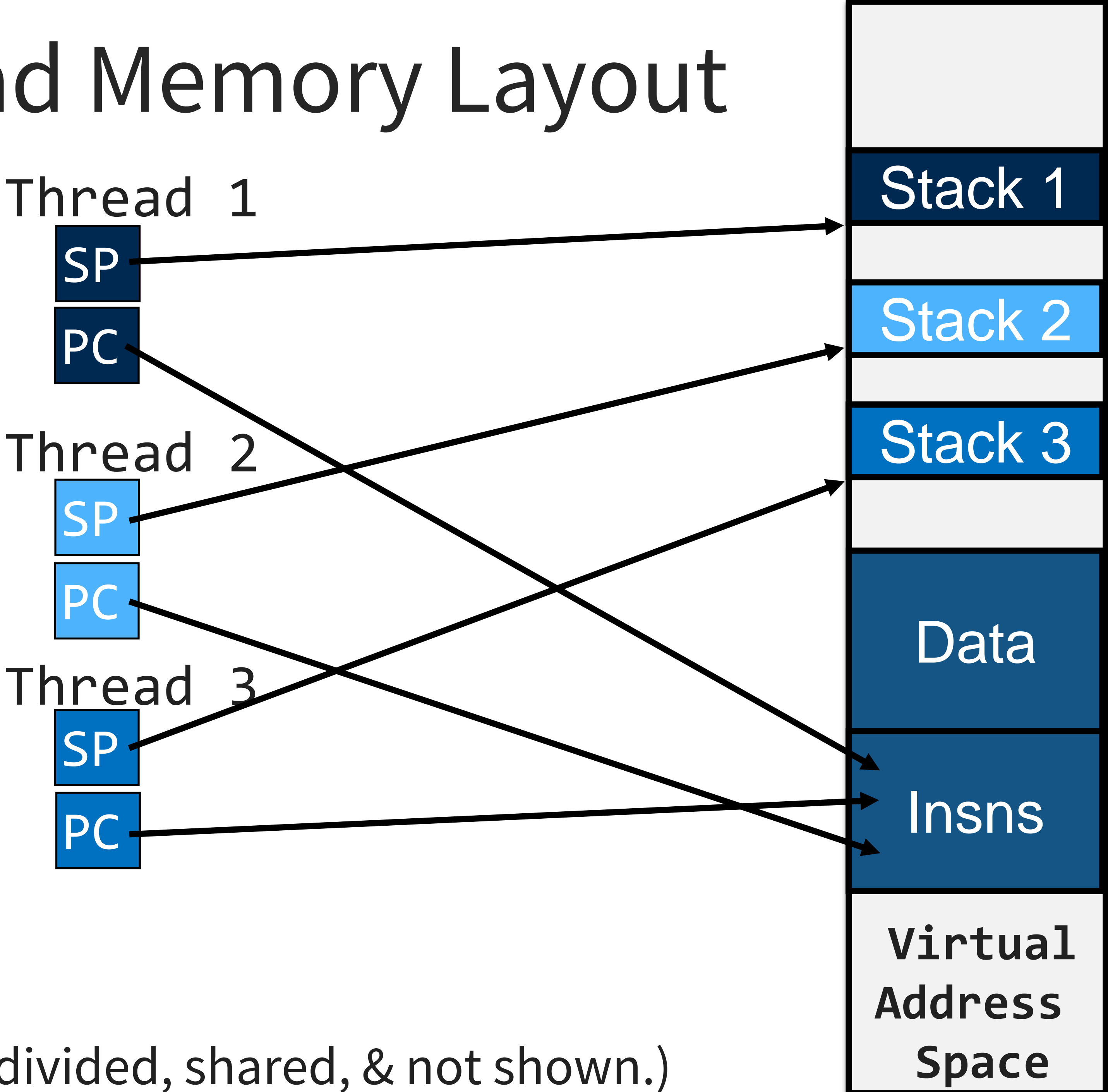

```
Physical address space
Each process’ address space by color
(shown contiguous to look nicer)
```

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



(Heap subdivided, shared, & not shown.)

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
- execution speed is unspecified, non-zero

Virtualizes the processor

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

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

Processes vs. Threads

- Have data/code/heap/stack
- Have at least one thread
- Process dies → resources reclaimed, its threads die
- Interprocess communication via OS and data copying
- Have own address space, isolated from other processes'
- **Expensive** creation and context switch

- Have own stack
- 1+ threads live in a process
- Thread dies → its stack reclaimed
- Inter-thread communication via memory
- Have own stack and regs, but no isolation from other threads in the same process
- **Inexpensive** creation and context switch

- Each can run on a different processor

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.

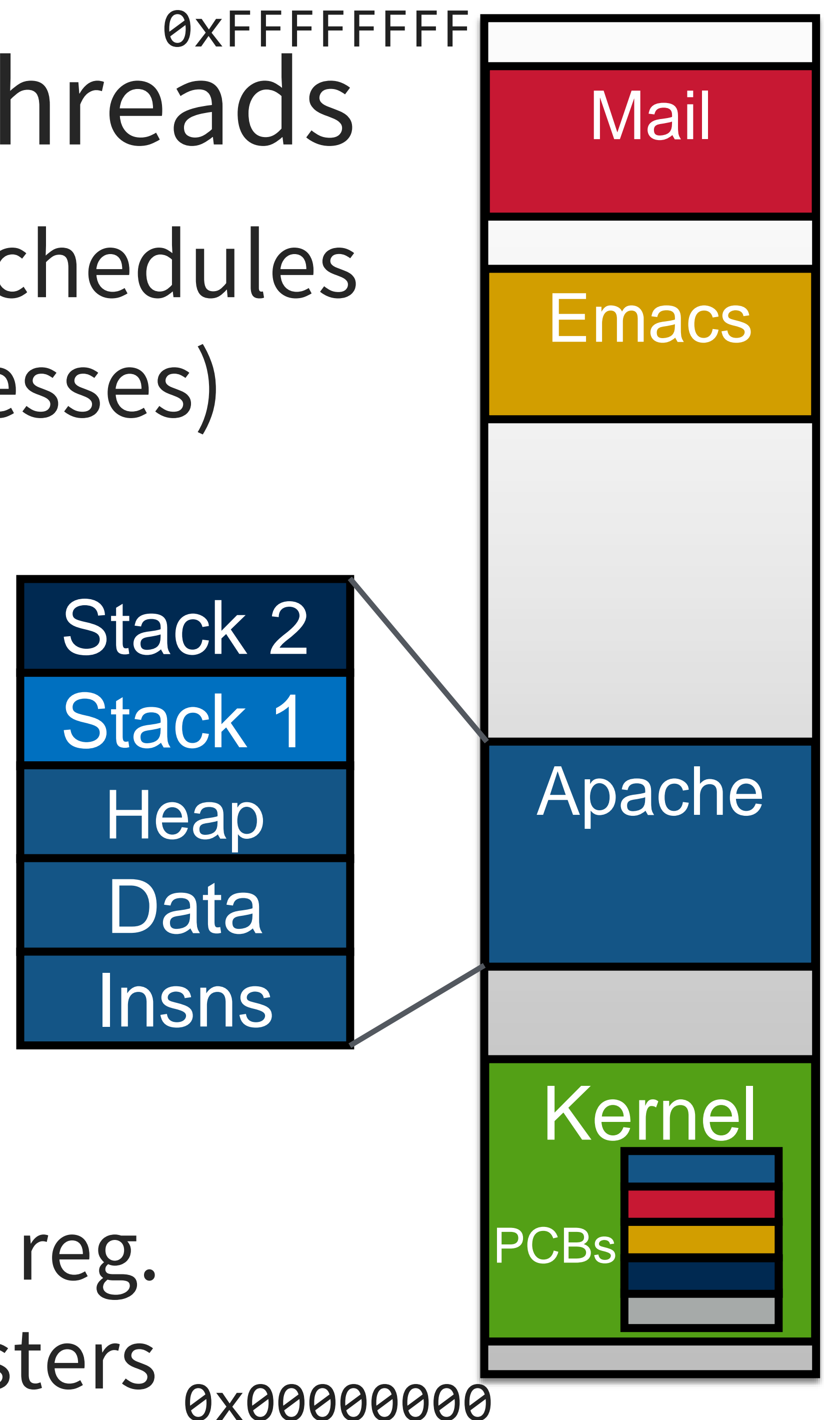
`int`
`thread_join`
`(thread)`
	Wait for **thread** to finish, then return the value **thread** passed to thread_exit.
	May be called only once for each thread.

`void`
`thread_exit`
`(ret)`
	Finish caller; store **ret** in caller’s TCB and wake up any thread that invoked thread_join(caller).

#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



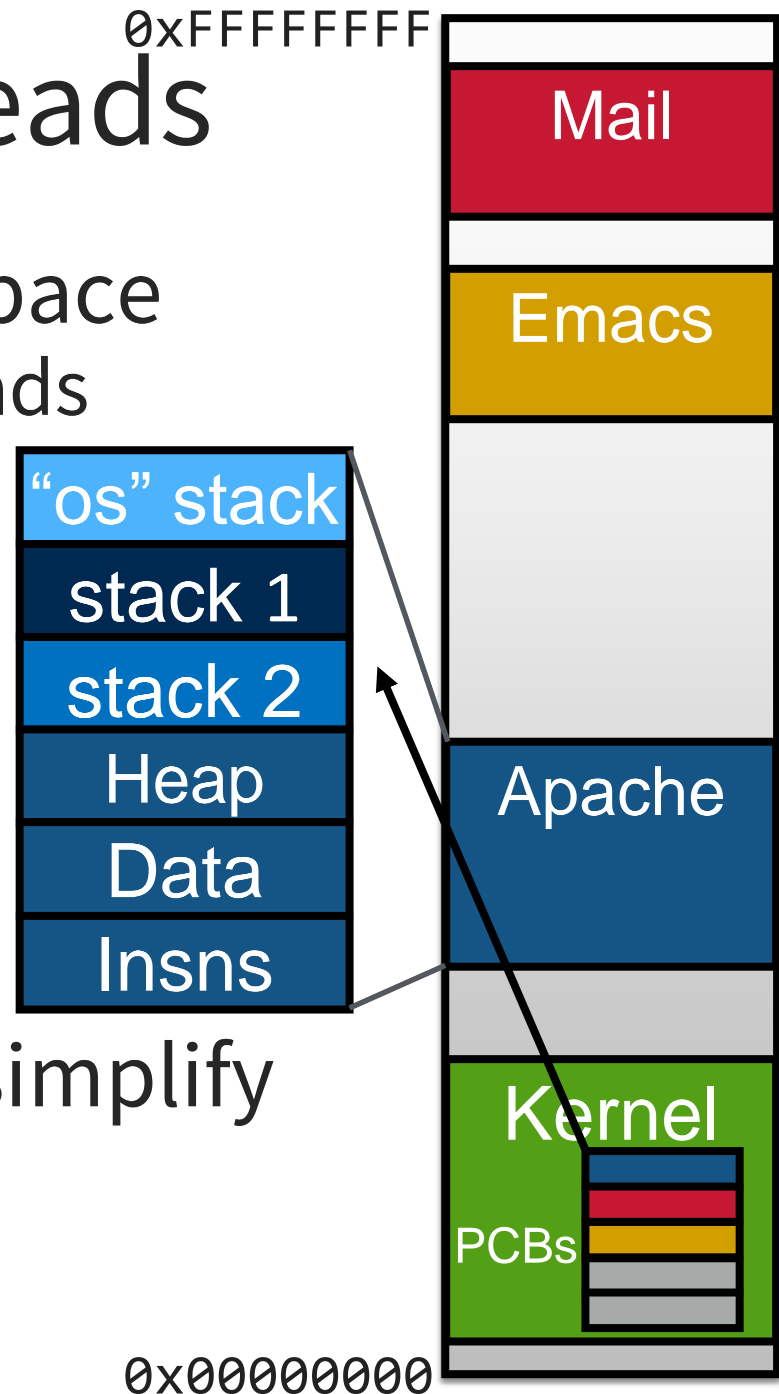
#2: User-Level Threads

Build a mini-OS in user space

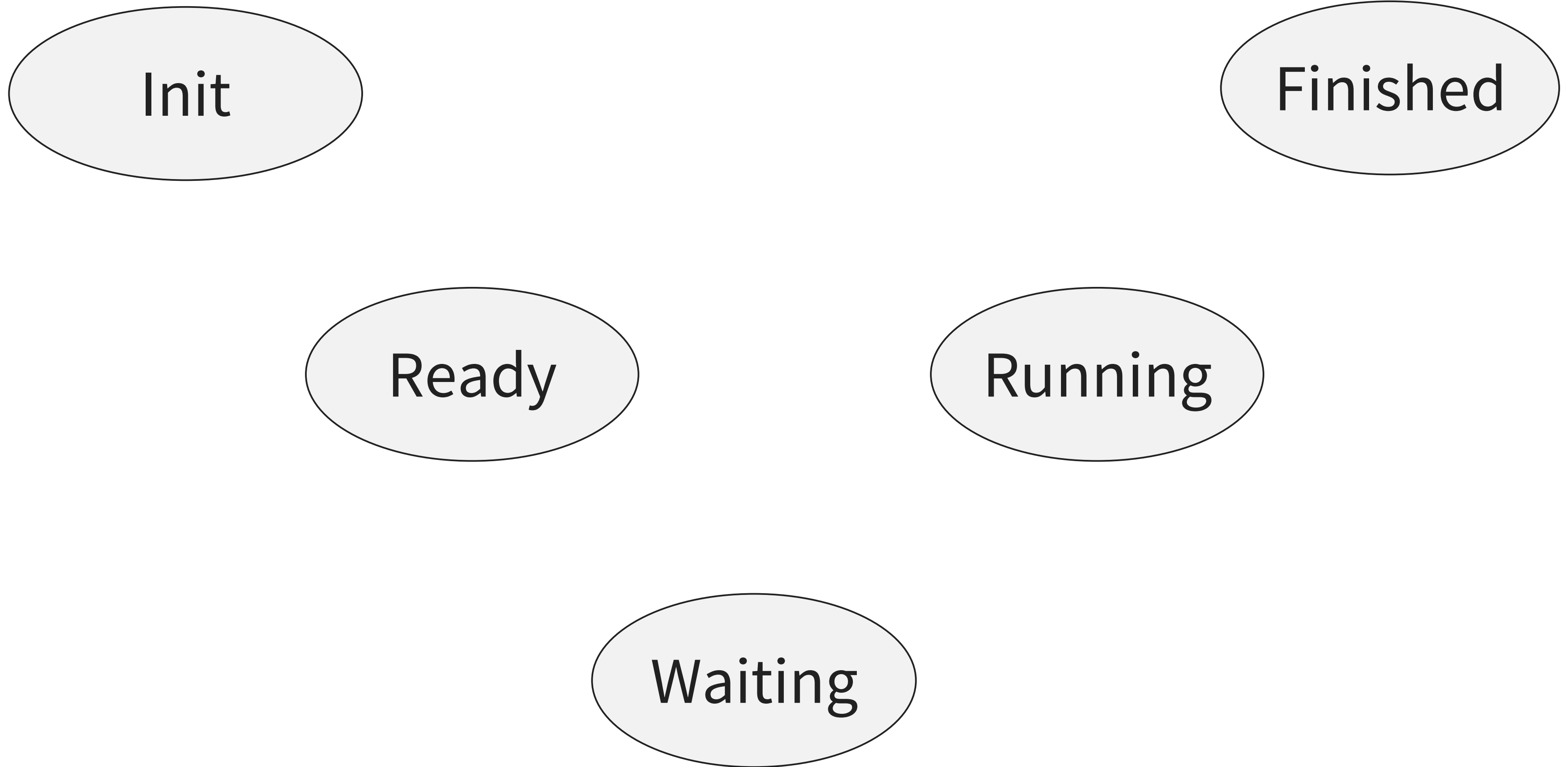
- Real OS unaware of threads
- Single PCB

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

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



Thread Life Cycle

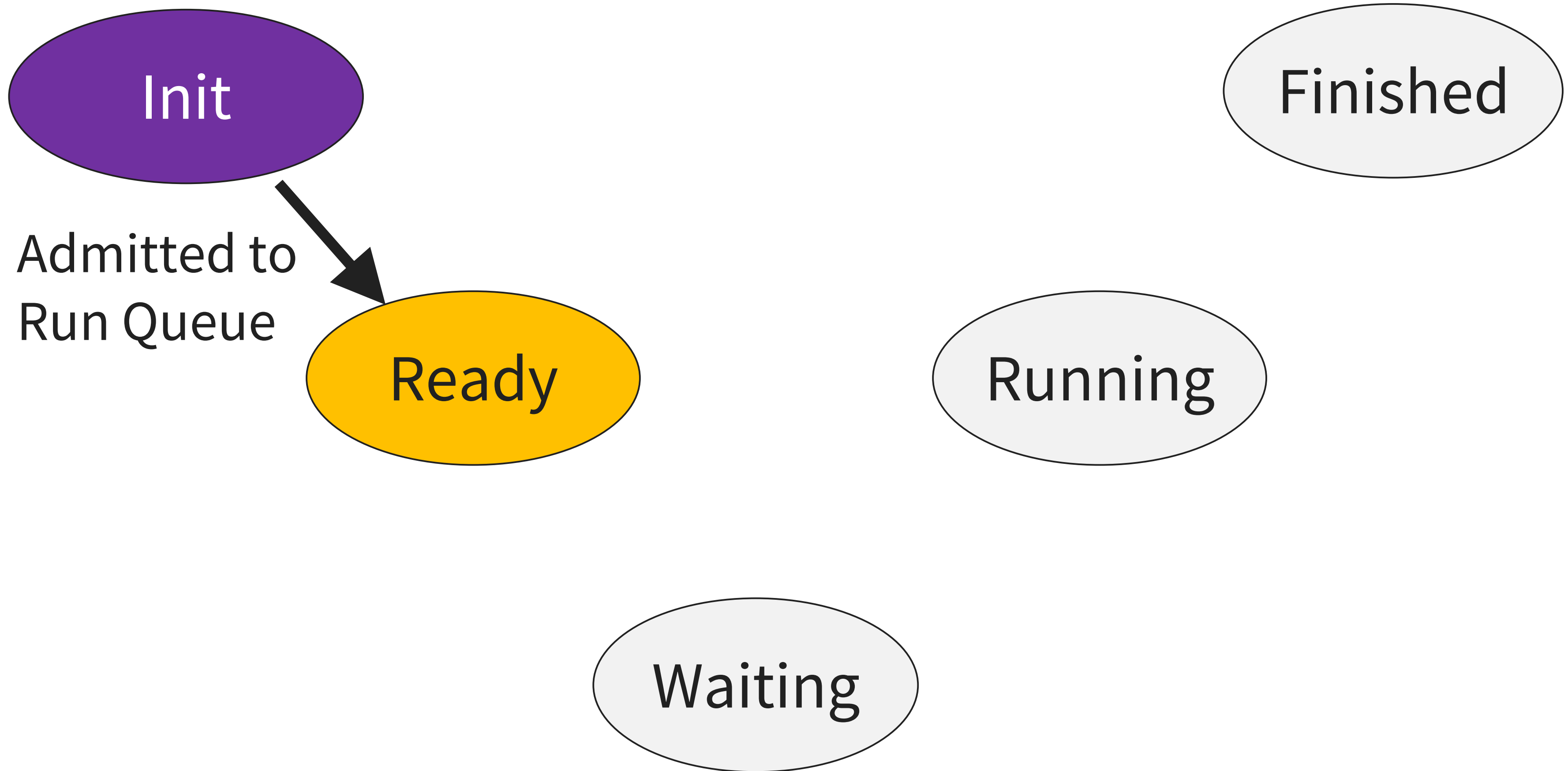


Processes go through these states, too.

Thread creation

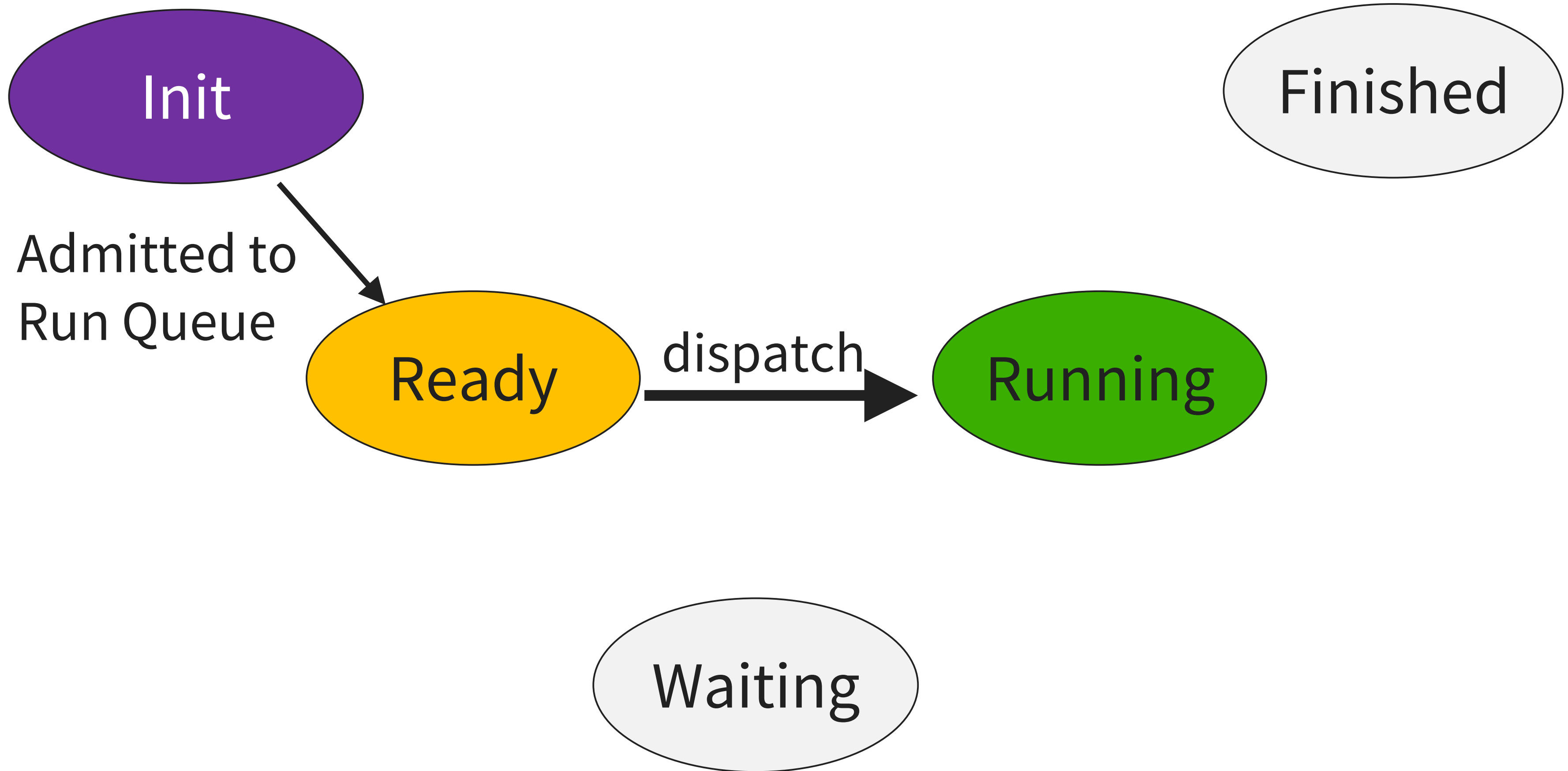
TCB status: being created
Registers: in TCB

Thread is Ready to Run



TCB: on Ready list
Registers: in TCB

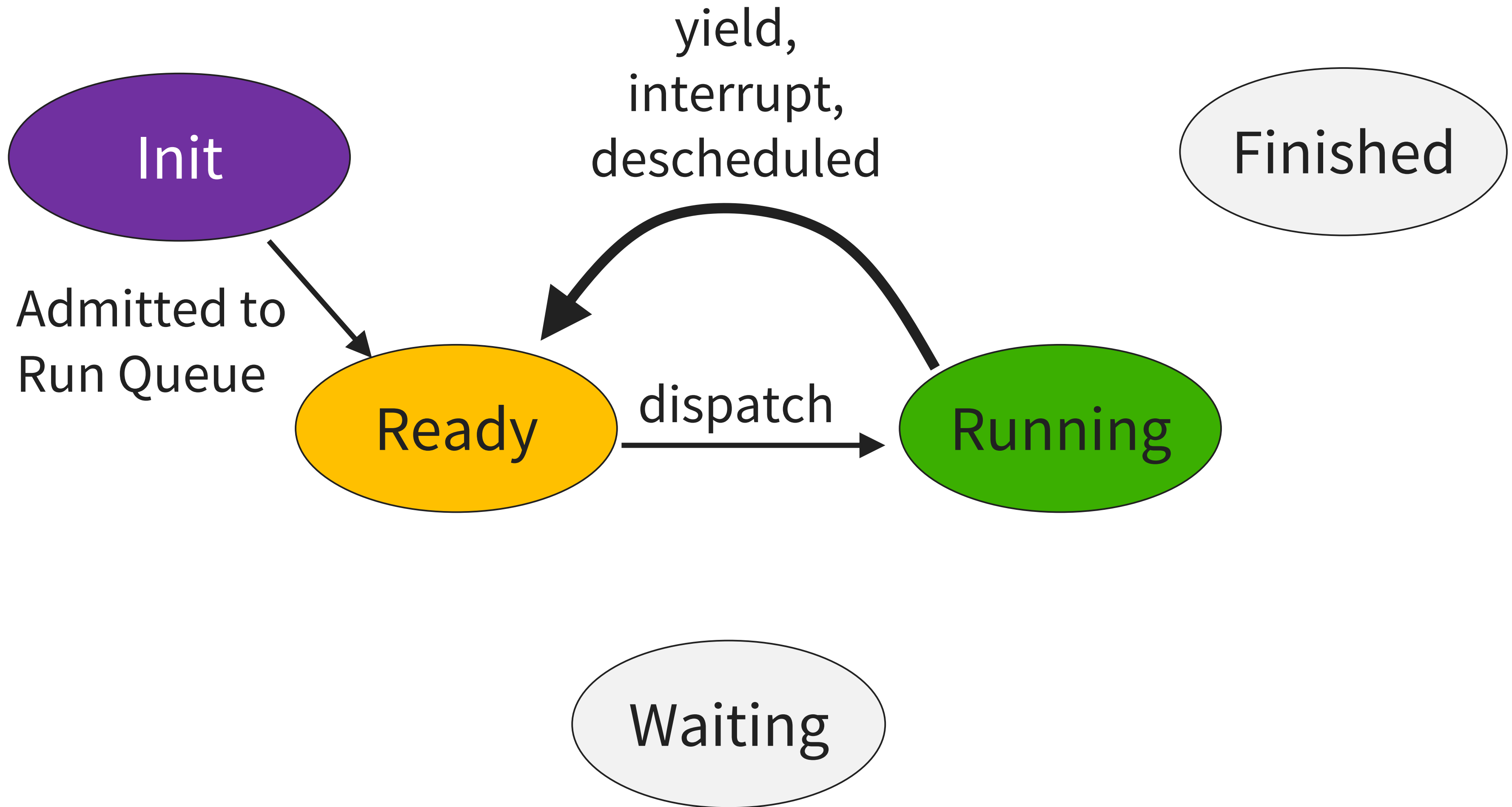
Thread is Running



TCB: on Running list

Registers: Processor

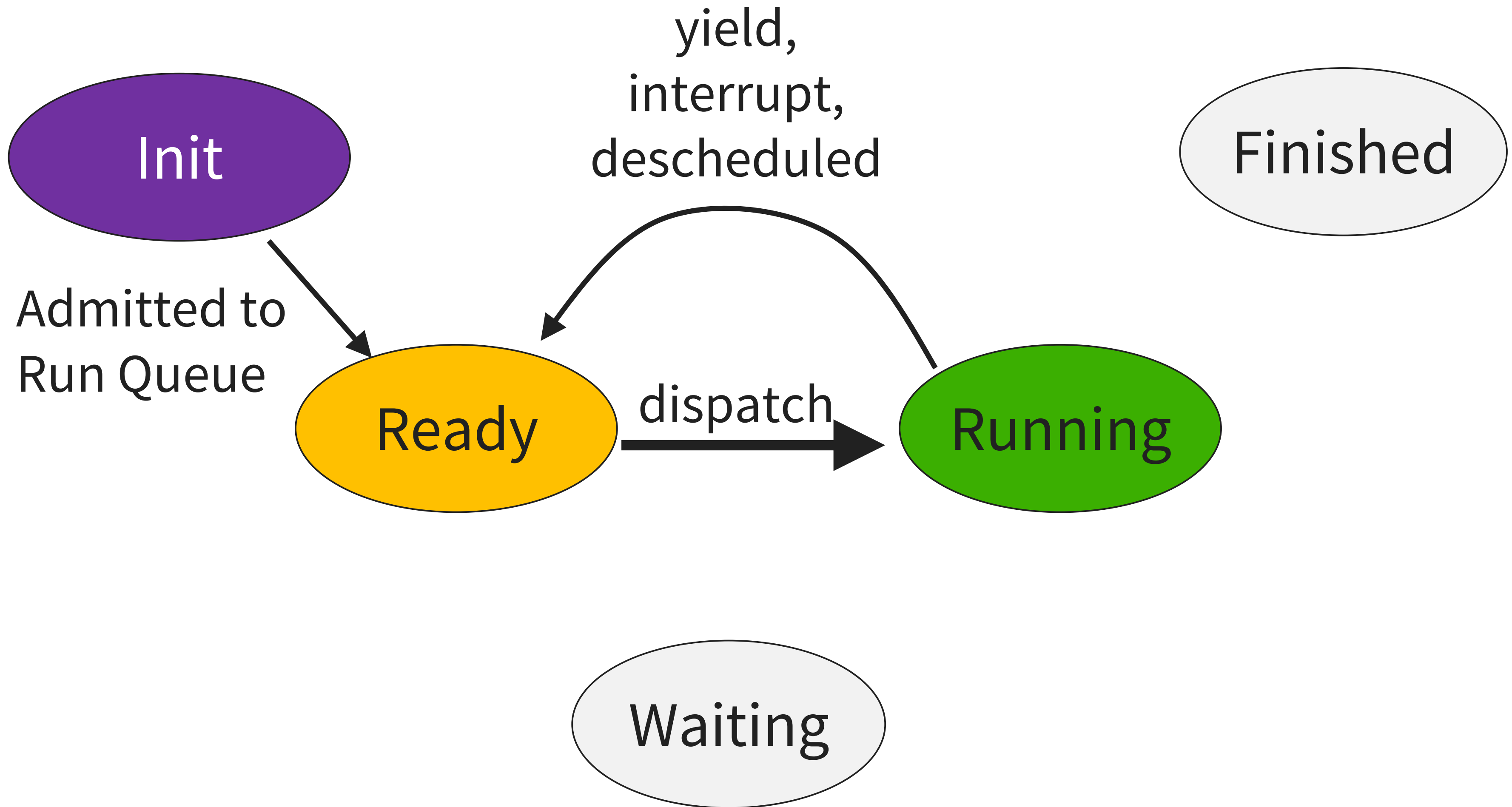
Thread Yields (back to Ready)



TCB: on Ready list

Registers: in TCB

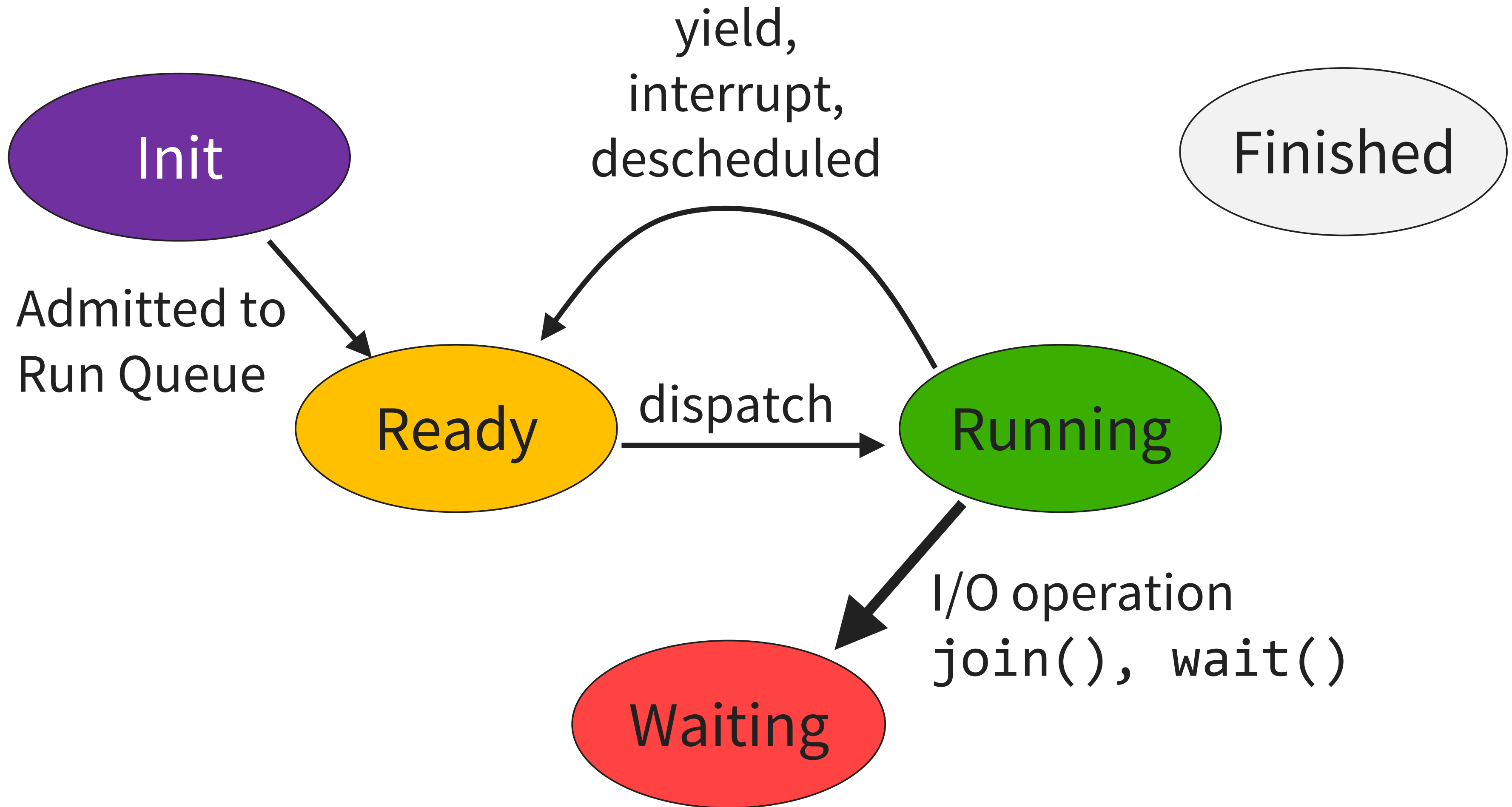
Thread is Running Again!



TCB: on Running list

Registers: Processor

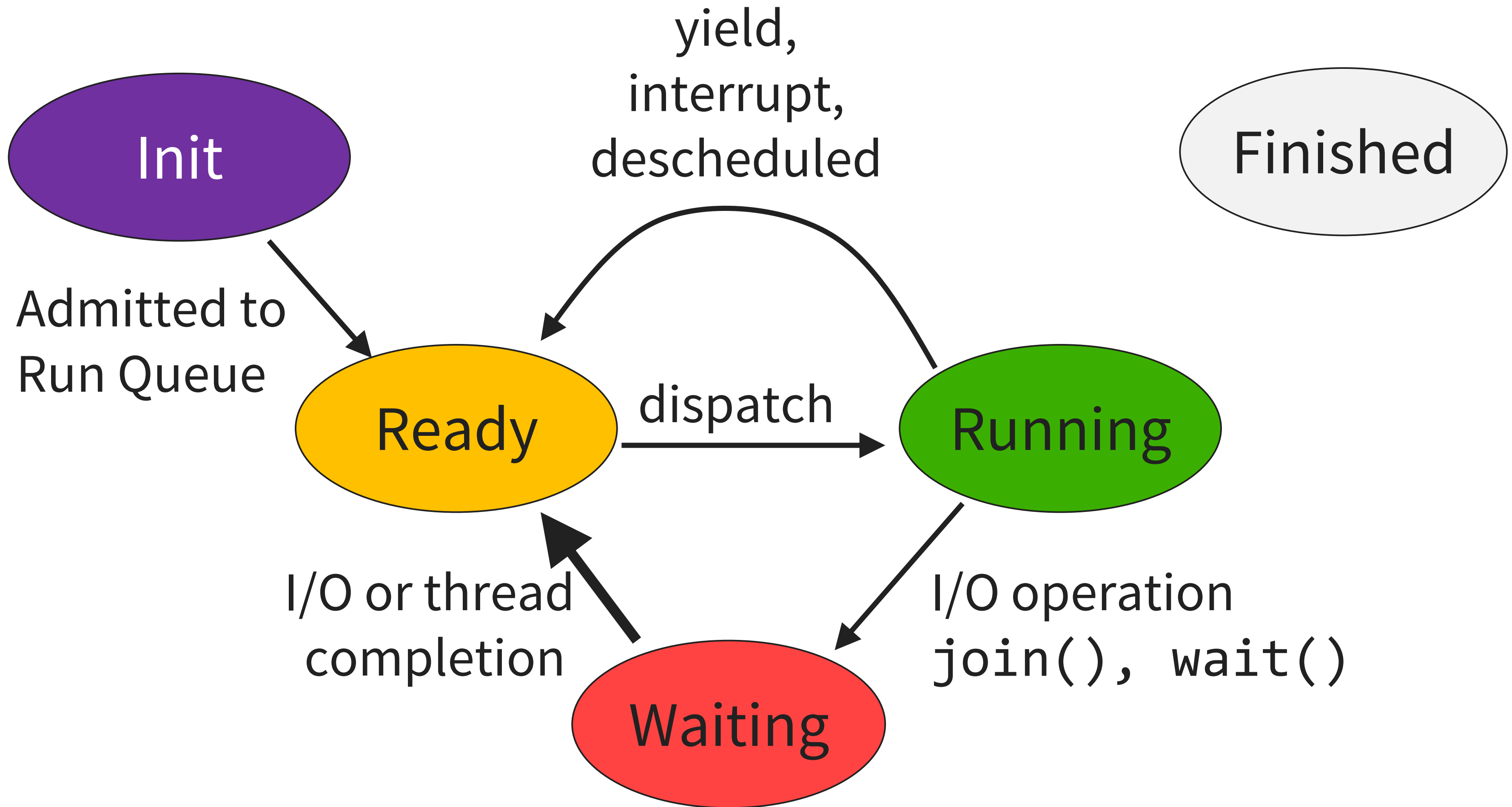
Thread is Waiting



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

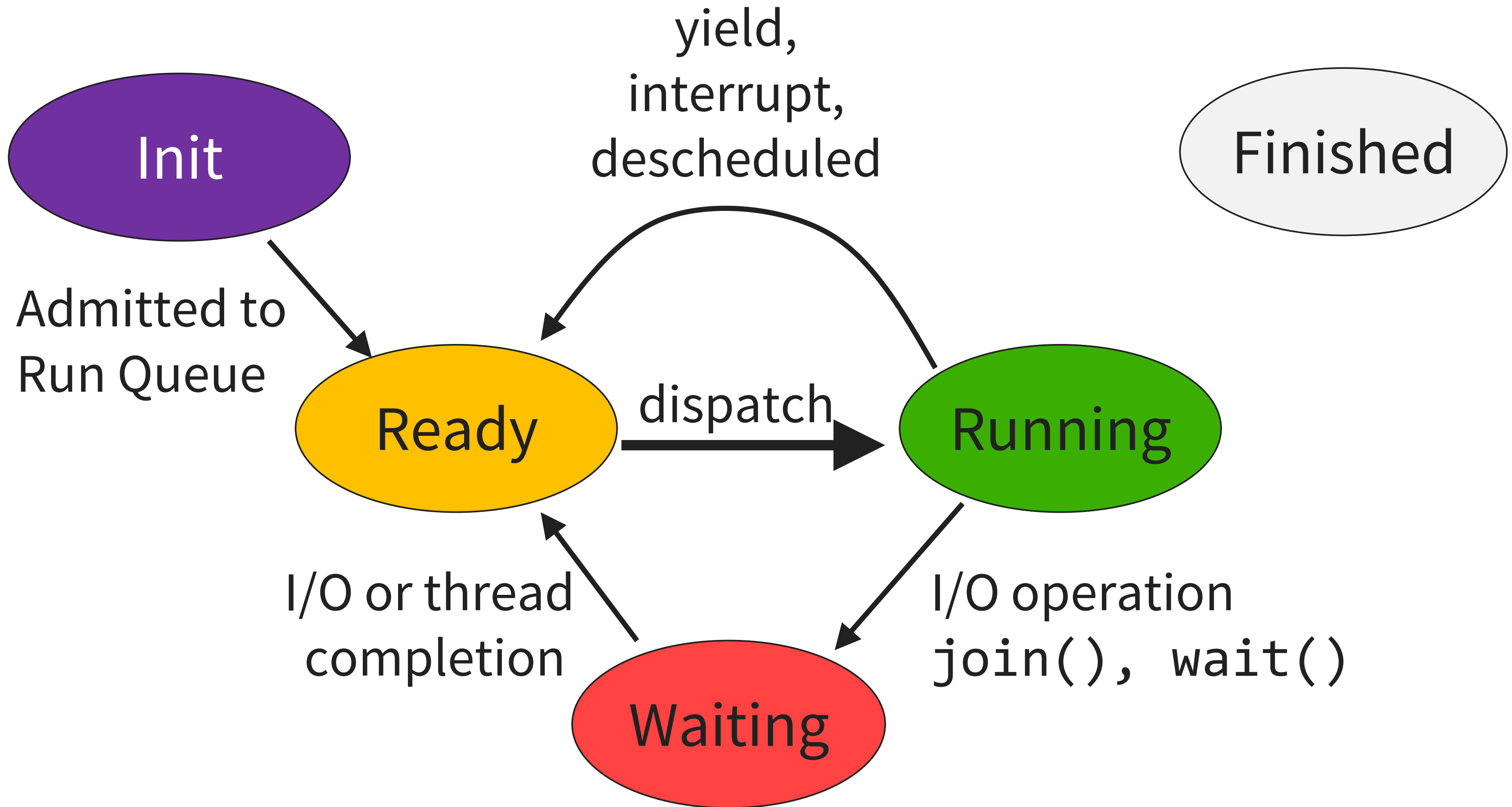
Registers: TCB

Thread is Ready Again!



TCB: on Ready list
Registers: in TCB

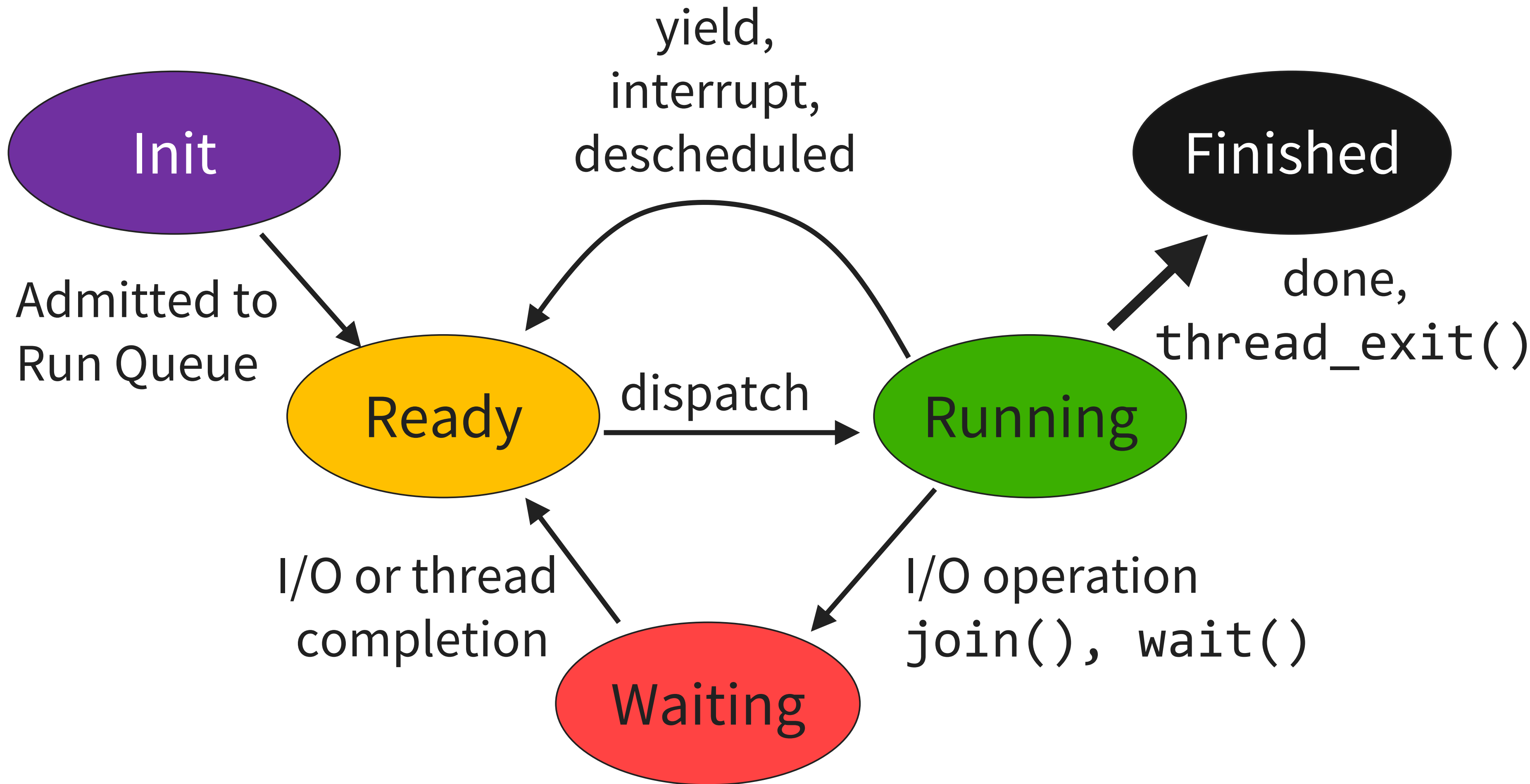
Thread is Running Again!



TCB: on Running list

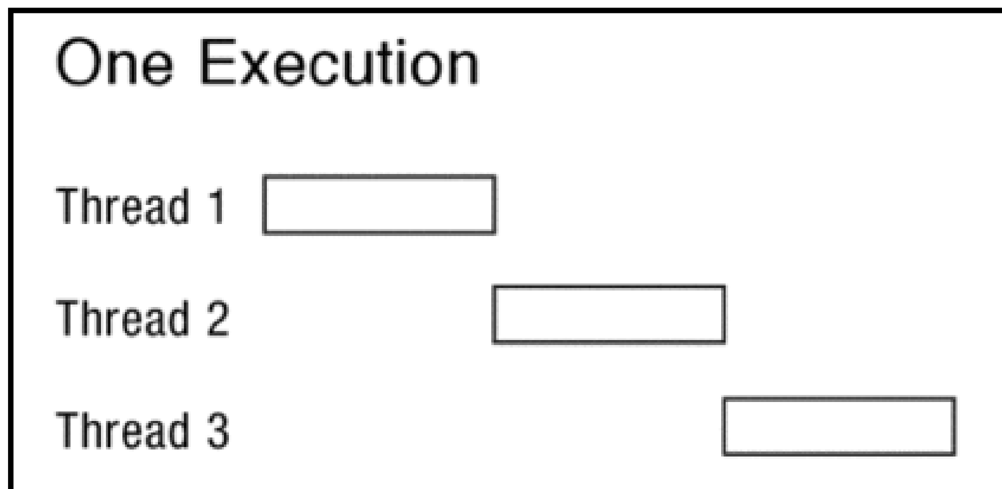
Registers: Processor

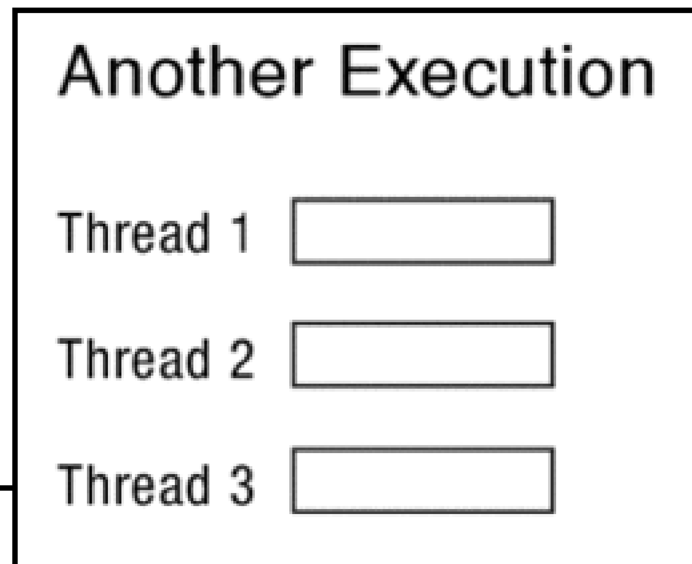
Thread is Finished (Process = Zombie)

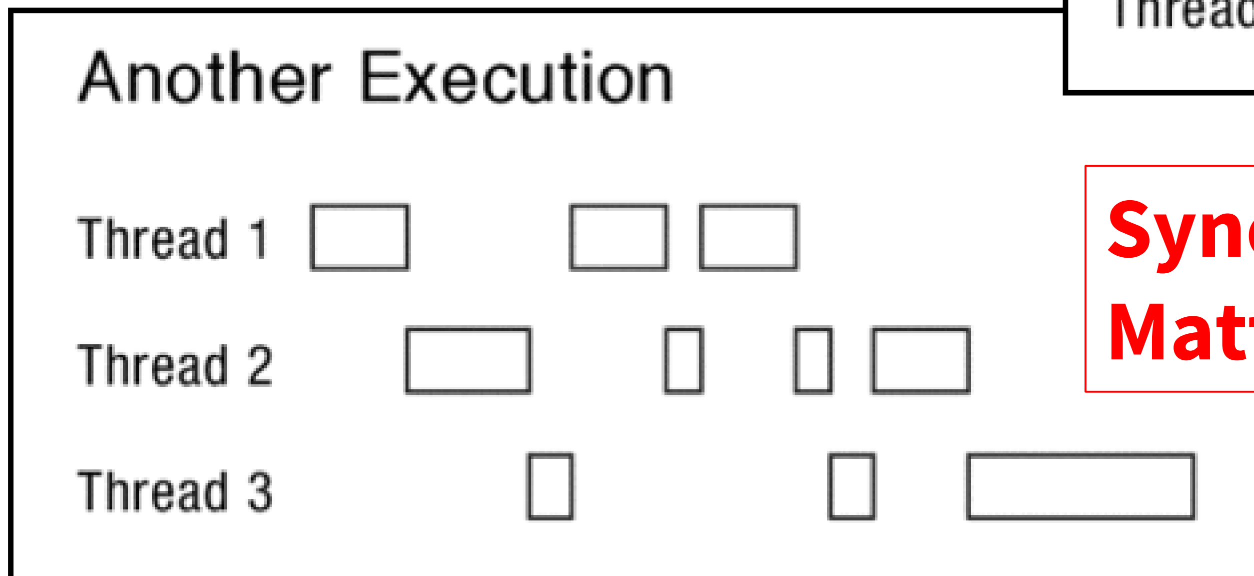


TCB: on Finished list (to pass exit value), ultimately deleted
Registers: TCB

Do **not** presume to know the schedule







Synchronization Matters!