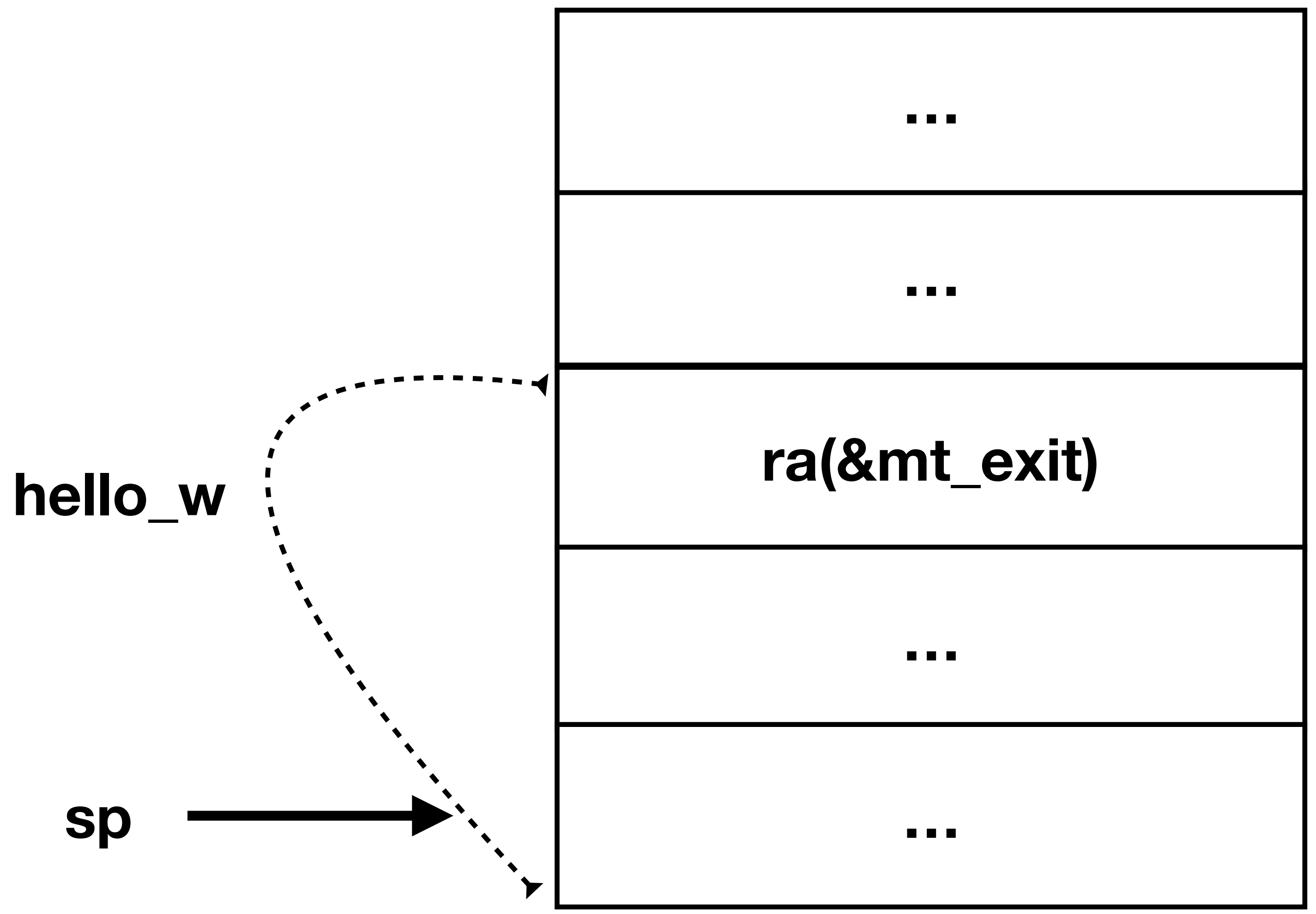
P1 - Semaphores

Drew Zagieboylo
2 / 9 / 18

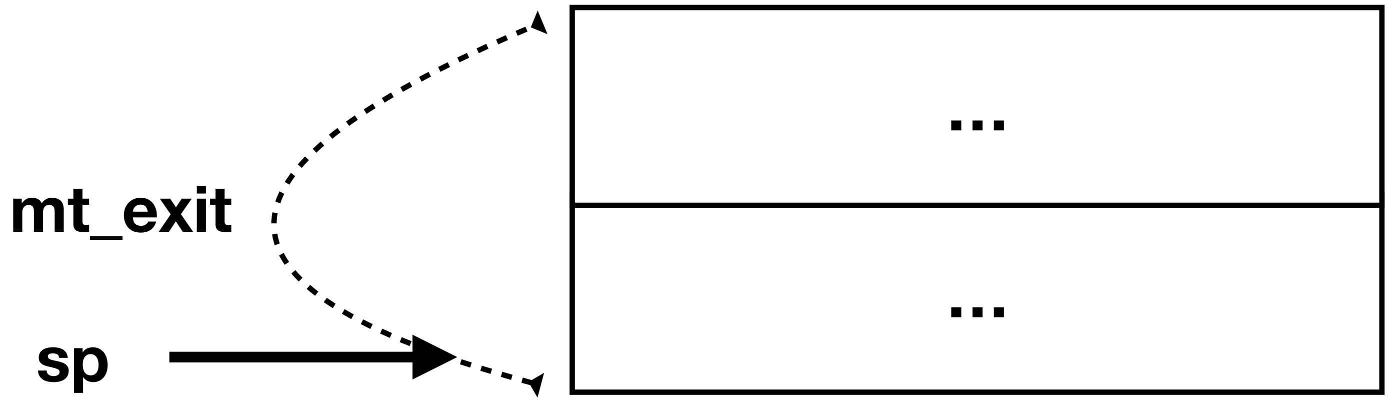
Thread Death

```
void hello_w() {
    printf("Hello World!");
    return;
}
```

```
void mt_exit() {
    //do cleanup
    while (1) {};
}
```



Thread Death

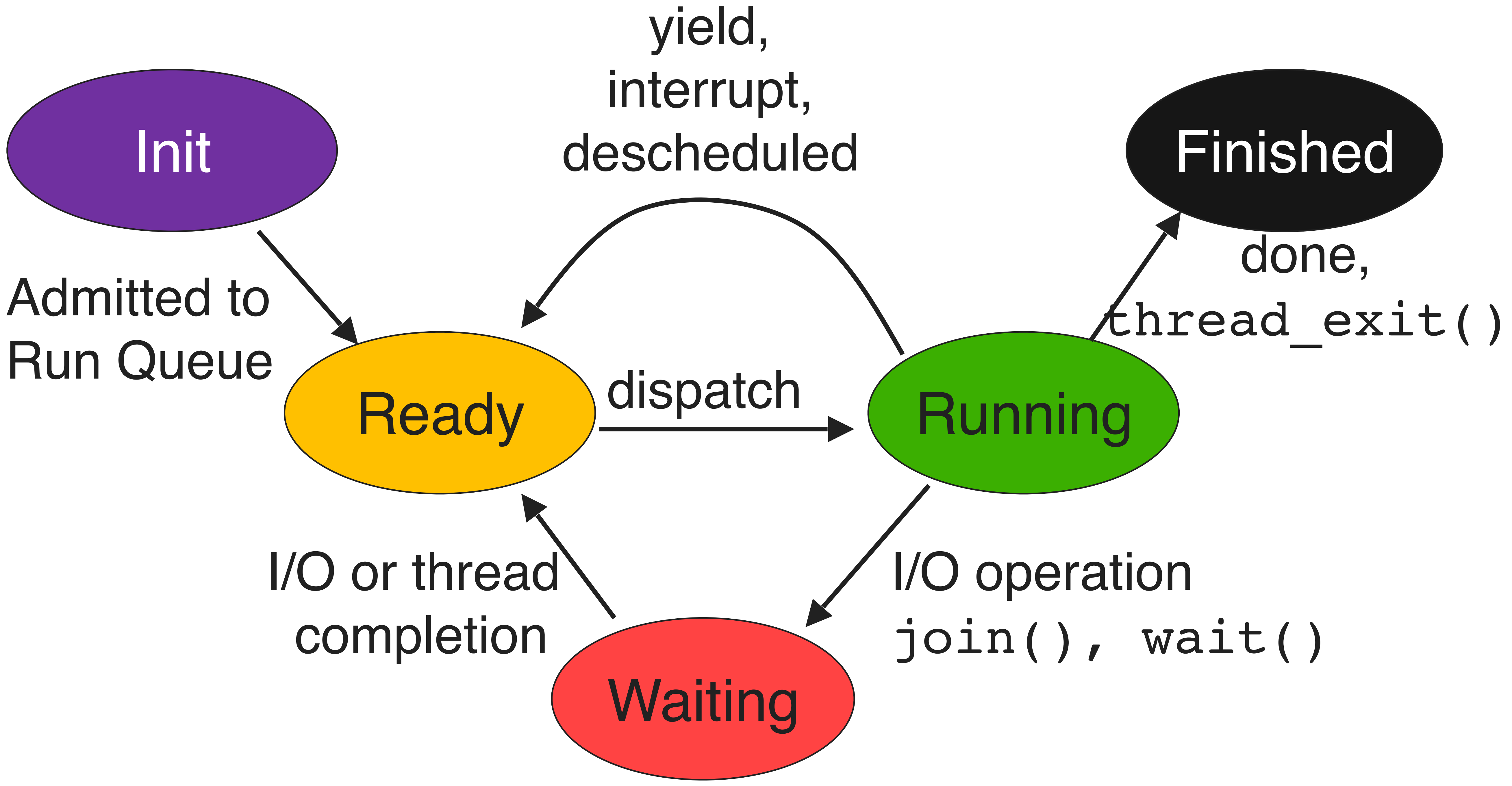


```
void hello_w() {
    printf("Hello World!");
    return;
}
```

pc

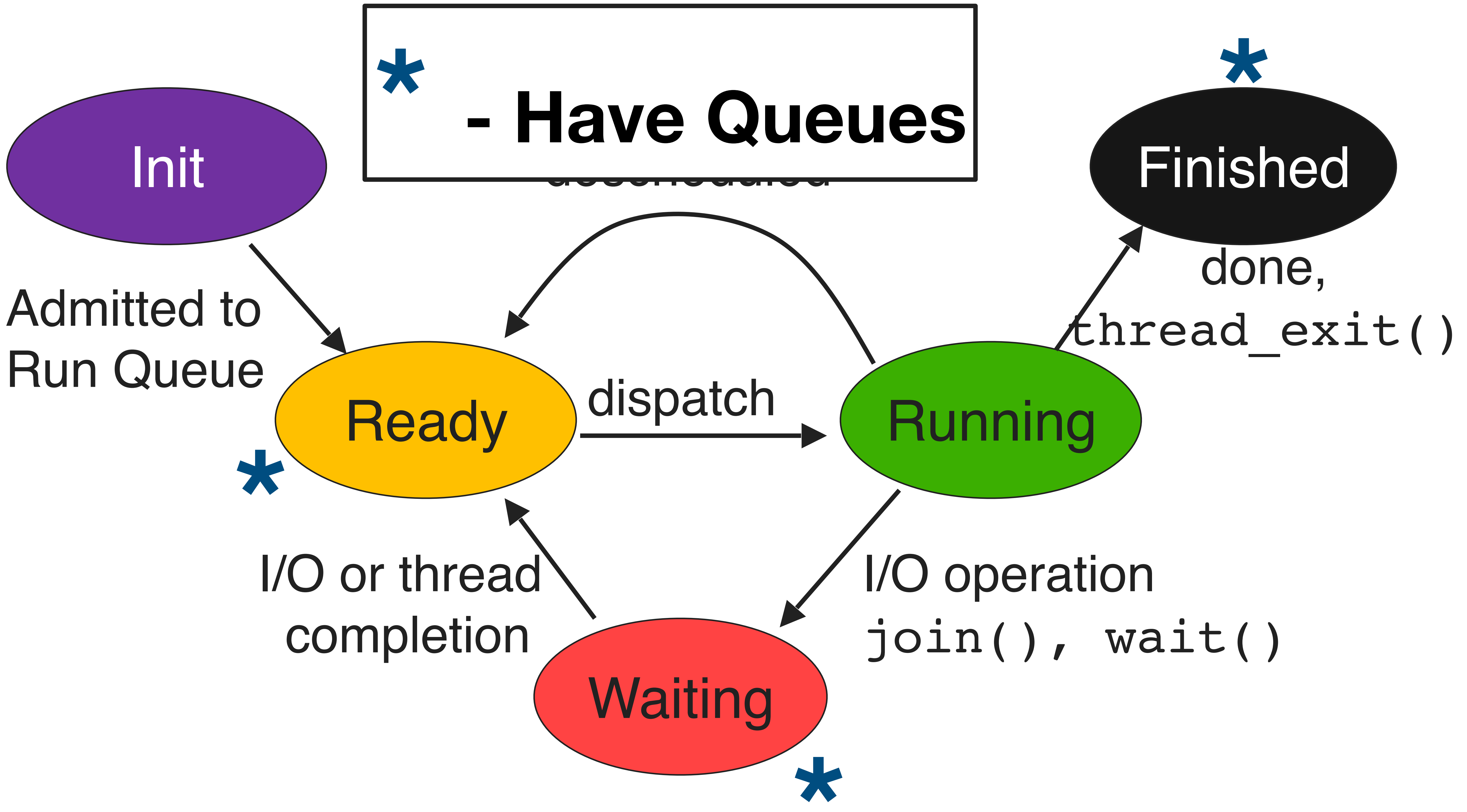
```
void mt_exit() {
    //do cleanup
    while (1) {};
}
```

Thread State Transitions



TCB:
Registers:

Thread State Transitions



TCB:
Registers:

Thread Death

```
void mt_exit() {
    //do cleanup
    put current thread on
    cleanup queue
    while (1) {};
}
```

Problems?

Cleanup *Thread*

```
void cleanup() {
   while (1) {
   //remove thread to clean
   //from queue

   //free its stack
   //free its tcb
   }
}
```

Thread Cleanup

- Cleanup thread needs to be scheduled
- Shouldn't run when there's nothing to clean

How?

```
void cleanup() {
   while (1) {
   //remove thread to clean
    /from queue

   ,/free its stack
   //free its tcb
   }
}
```

Semaphores!

Finally I should like to thank the members of the program committee who asked for more information on the ***synchronizing primitives*** *and some* ***justification of my claim*** *to be able to* ***prove logical soundness a priori.*** *In answer to this request* ***the appendix has been added,*** *of which I hope that it gives the desired information and justification.*

- *Edsger W. Dijkstra. 1967. The structure of the "the"-multiprogramming system. In Proceedings of the first ACM symposium on Operating System Principles (SOSP '67), J. Gosden and B. Randell (Eds.). ACM, New York, NY, USA, 10.1-10.6. DOI=http://dx.doi.org/10.1145/800001.811672*

- *https://dl.acm.org/citation.cfm?id=811672*

The Private Semaphores.

Each sequential process has associated with it a number of private semaphores and no other process will ever perform a P-operation on them. The universe initializes them with the value = 0, their maximum value = 1, their minimum value = - 1.

Whenever a process reaches a stage where the permission for dynamic progress depends on current values of state variables, it follows the pattern:

```
P(mutex);
"inspection and modification of state variables
including a conditional V(private semaphore)";
V(mutex);
P(private semaphore)
```

If the inspection learns that the process in question should continue, it performs the operation "V(private semaphore)" -the semaphore value then changes from 0 to 1-, otherwise this V-operation is skipped, leaving to the other processes the obligation to perform this V-operation at a suitable moment. The absence or presence of this obligation is reflected in the final values of the state variables upon leaving the critical section.

Whenever a process reaches a stage where as a result of its progress possibly one (or more) blocked processes should now get permission to continue, it follows the pattern

```
P(mutex);
"modification and inspection of state variables
including zero or more V-operations on private
semaphores of other processes";
V(mutex)
```

By the introduction of suitable state variables and appropriate programming of the critical sections any strategy assigning peripherals, buffer areas etc. can be implemented.

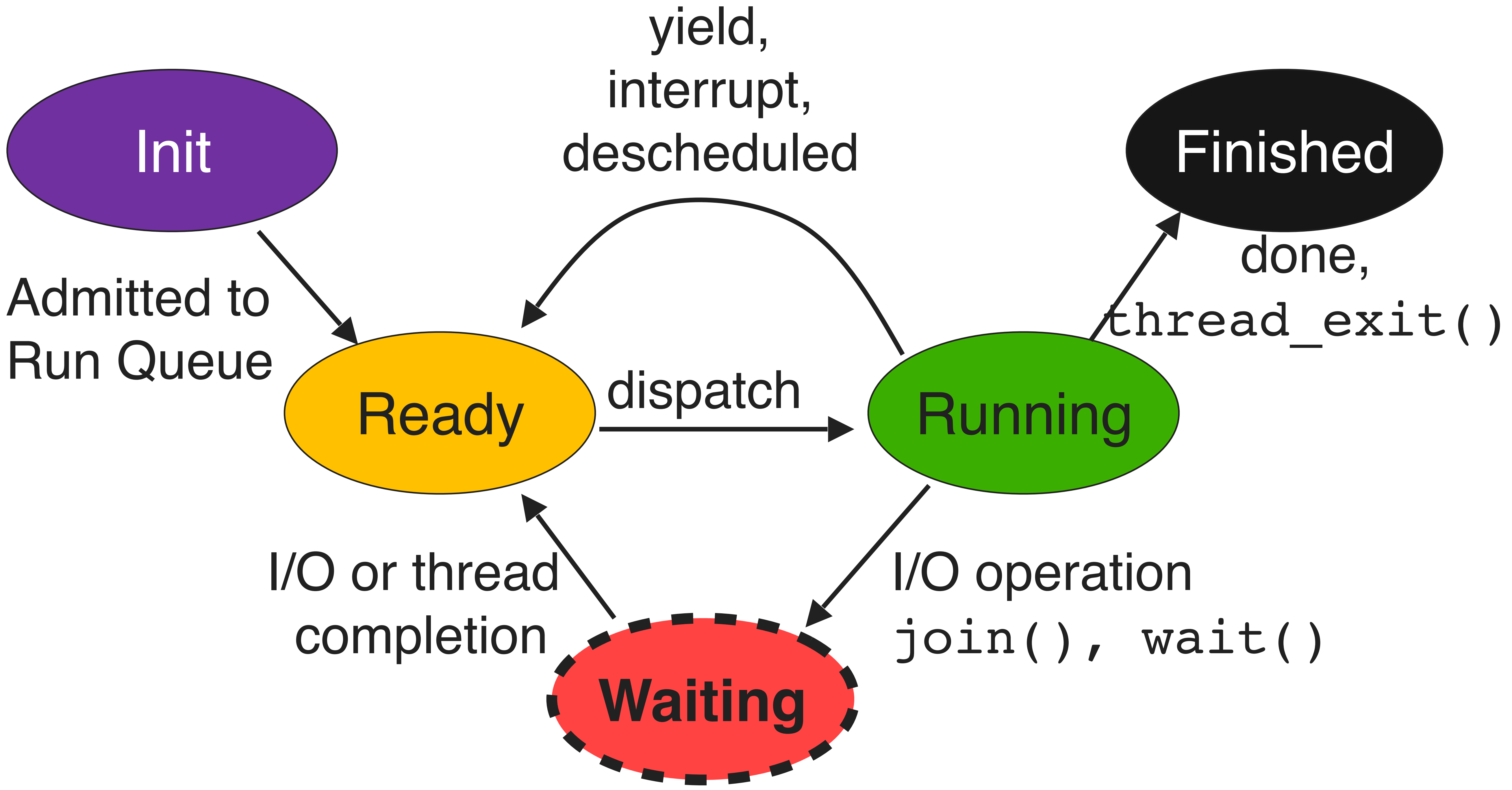
The amount of coding and reasoning can be greatly reduced by the observation that in the two complementary critical sections sketched above, the same inspection can be performed by the introduction of the notion of "an unstable situation", such as a free reader and a process needing a reader. Whenever an unstable situation emerges it is removed (including one or more V-operations on private semaphores) in the very same critical section in which it has been created.

Semaphores

- Stateful:
  - count
  - queue of threads

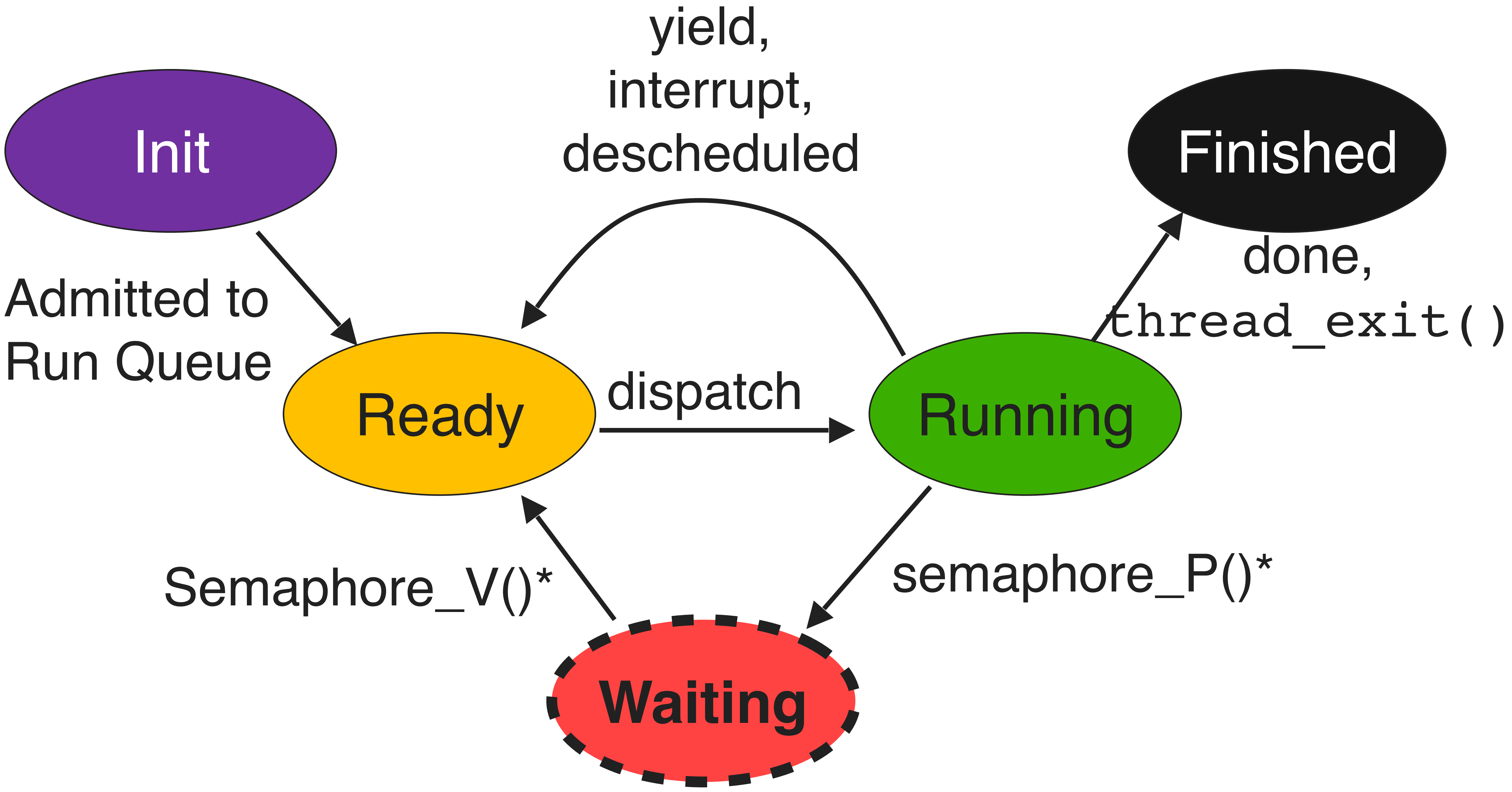
- Functions:
  - P(sema)
    procure -> block this thread until resource can be procured
  - V(sema)
    vacate -> release this resource and continue

Thread State Transitions



TCB: On semaphore's Queue
Registers: On stack

Thread State Transitions



TCB: On semaphore's Queue
Registers: On stack

Example

```
var x = 0;

void inc() {            void dec() {
  int i = 0;              int i = 0;
  (for; i < 1000; i++)    (for; i < 1000; i++)
    x++;                    x--;
}                       }
```

Example

```
var x = 0;

void inc() {
  int i = 0;
  (for; i < 1000; i
   x = x + 1;
}
```

```
id dec() {
nt i = 0;
or; i < 1000; i++)
   x = x - 1;
}
```

NOT ATOMIC!

Example

```
var x = 0;
var lock = sema(1);

void inc() {           void dec() {
  int i = 0;             int i = 0;
  (for; i < 1000; i++)   (for; i < 1000; i++)
    x = x + 1;             x = x - 1;
}                      }
```

Example

```
var x = 0;
var lock = sema(1);

void inc() {
   int i = 0;
   (for; i < 1000; i++){
      P(lock);
      x = x + 1;
      V(lock);
   }
}
```

```
void dec() {
   int i = 0;
   (for; i < 1000; i++)
      x = x - 1;
}
```

Example

```
var x = 0;
var lock = sema(1);

void inc() {
   int i = 0;
   (for; i < 1000; i++){
     P(lock);
     x = x + 1;
     V(lock);
   }
}
```

```
void dec() {
   int i = 0;
   (for; i < 1000; i++){
     P(lock);
     x = x - 1;
     V(lock);
   }
}
```

Semaphores

- Stateful:
  - count
  - queue of threads

```
requests[illegible]sema[illegible];

while (req[illegible]->count > 0) {
   …
   …
}
```

- Functions:
  - P(sema)
    procure -> block this thread until resource can be procured
  - V(sema)
    vacate -> release this resource and continue

Semaphore Invariants

- Count:
  - If $c \geq 0$
    - The number of resources available
  - If $c \leq 0$
    - The number of threads waiting on the queue

Thread Cleanup

- Cleanup thread needs to be scheduled
- Shouldn’t run when there’s nothing to clean

```
void cleanup() {
    while (1) {
        //wait for thread
        //to be on queue

        //remove thread to clean
        //from queue

        //free its stack
        //free its tcb
    }
}
```