COSC 3360
Spring 2020
First Assignment

Jehan-François Pâris
jfparis@uh.edu

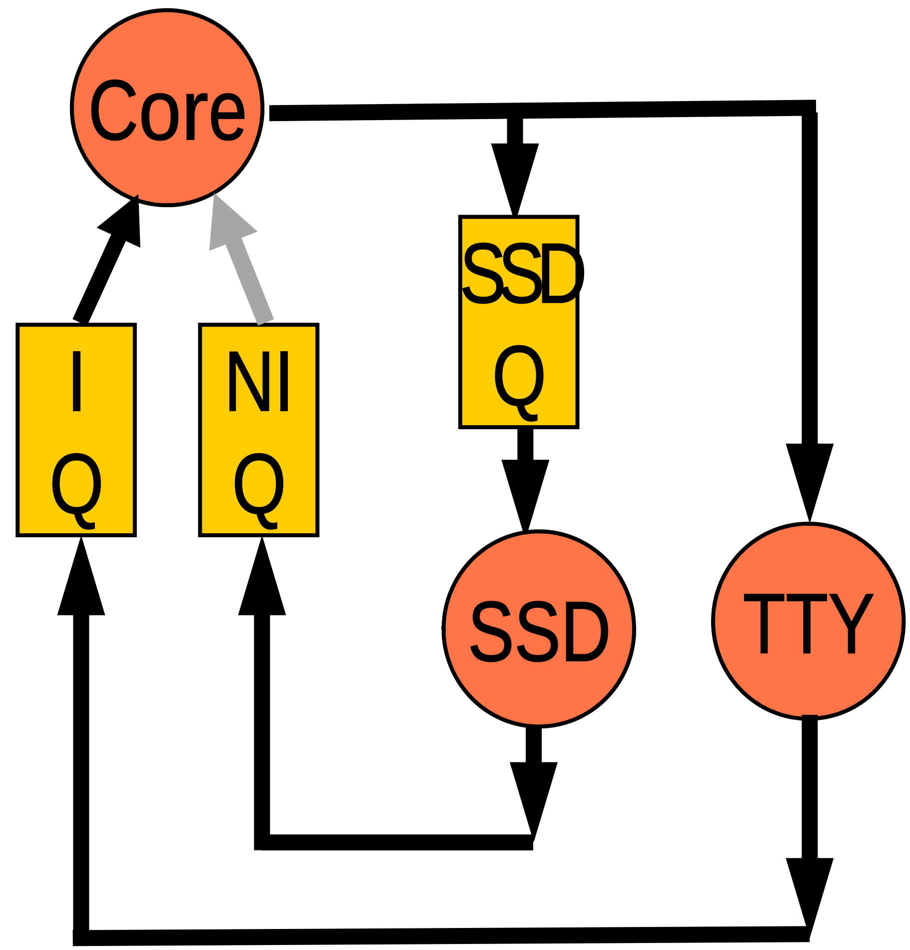
A very simple case

- **NCORES 1** // number of cores
  START 120 // new process at T=1200
  PID 23 // process ID
  CORE 100 // request CORE for 100 ms
  TTY 5000 // 5000 ms user interaction
  CORE 80 // request CORE for 80 ms
  SSD 1 // request SSD for 1 ms
  CORE 30 // request CORE for 30 ms
  SSD 1 // request SSD for 1 ms
  CORE 20 // request CORE for 20 ms

The model

We have

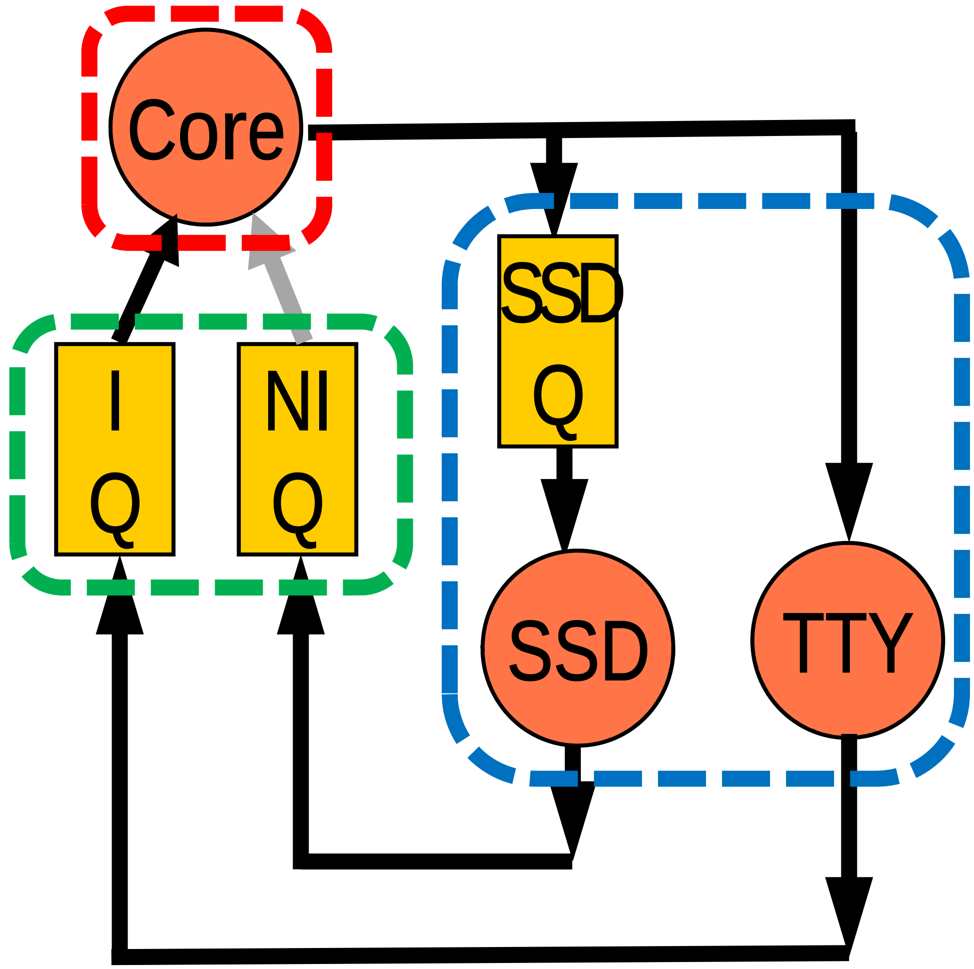
- One single-core CPU
  - NCORES = 1
- One SSD
- Many user windows
- Two CPU queues
  – Interactive
  – Non-interactive



Process states

A process can be

- ***Running***
  - It *o*ccupies a core
- ***Ready***
  - It waits for a core
- ***Blocked***
  - It wait for an I/O completion



The solution (I)

- **`NCORES 1`**
  - CPU has one core
- **`START 120`**
  - Set clock at T=120 ms
- **`PID 23`**
  - Record arrival of process 23 at T=120ms

The solution (II)

- **`CORE 100`**
  - T=120ms
  - Allocate core to process 23 for 100ms
  - Move 23 to RUNNING state
  - Core completion at T=120+100=220ms
- **`TTY 5000`**
  - T=220ms
  - Release core
  - Move 23 to BLOCKED state
  - TTY completion at T=220+5,000=5,220ms

The solution (III)

- `CORE 80`
  - T=5,220ms
  - Allocate core to process 23 for 80ms
  - Move 23 to RUNNING state
  - Core completion at T=5,220+80=5,300ms
- `SSD 1`
  - T=5,300ms
  - Release core
  - Move 23 to BLOCKED state
  - SSD completion at T=5,300+1=5,301ms

The solution (IV)

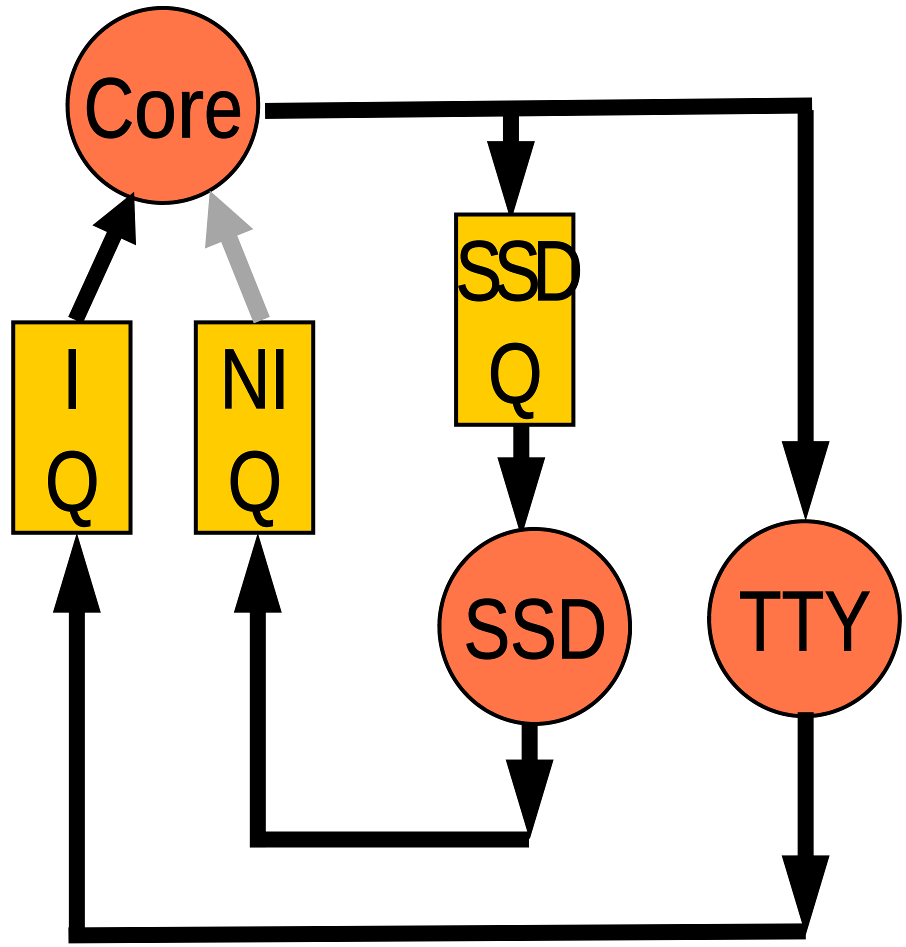
- **`CORE 30`**
  - T = 5,301ms
  - Allocate core to process 23 for 30ms
  - Move 23 to RUNNING state
  - Core completion at T=5,301+30=5,331ms
- **`SSD 1`**
  - T=5,331ms
  - Release core
  - Move 23 to BLOCKED state
  - SSD completion at T=5,331+1=5,332ms

The solution (V)

- **CORE 20**
  - T=5,332ms
  - Allocate core to process 23 for 20ms
  - Move 23 to RUNNING state
  - Core completion at T=5,332+20=5,352ms
- ***Process terminates***
  - Move 23 to TERMINATED state

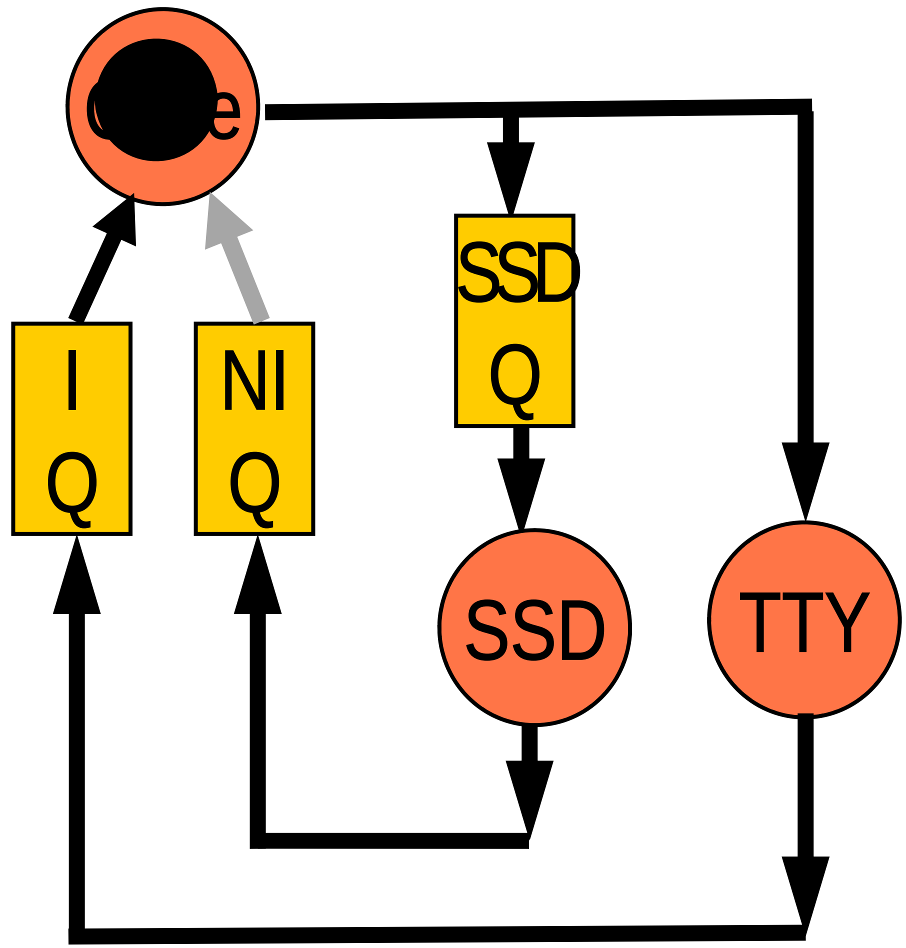
Another way to look at it

```
NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20
```



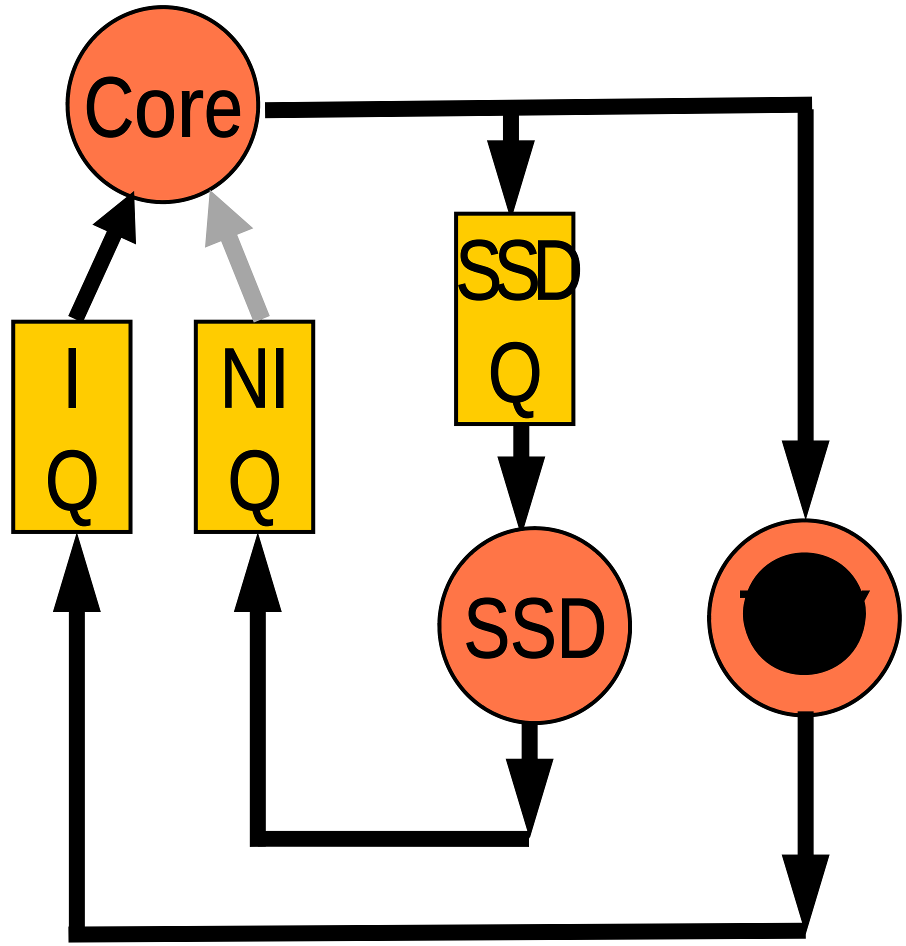
T=120ms Process 23 arrives
Gets core until T = 220ms

```
NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20
```



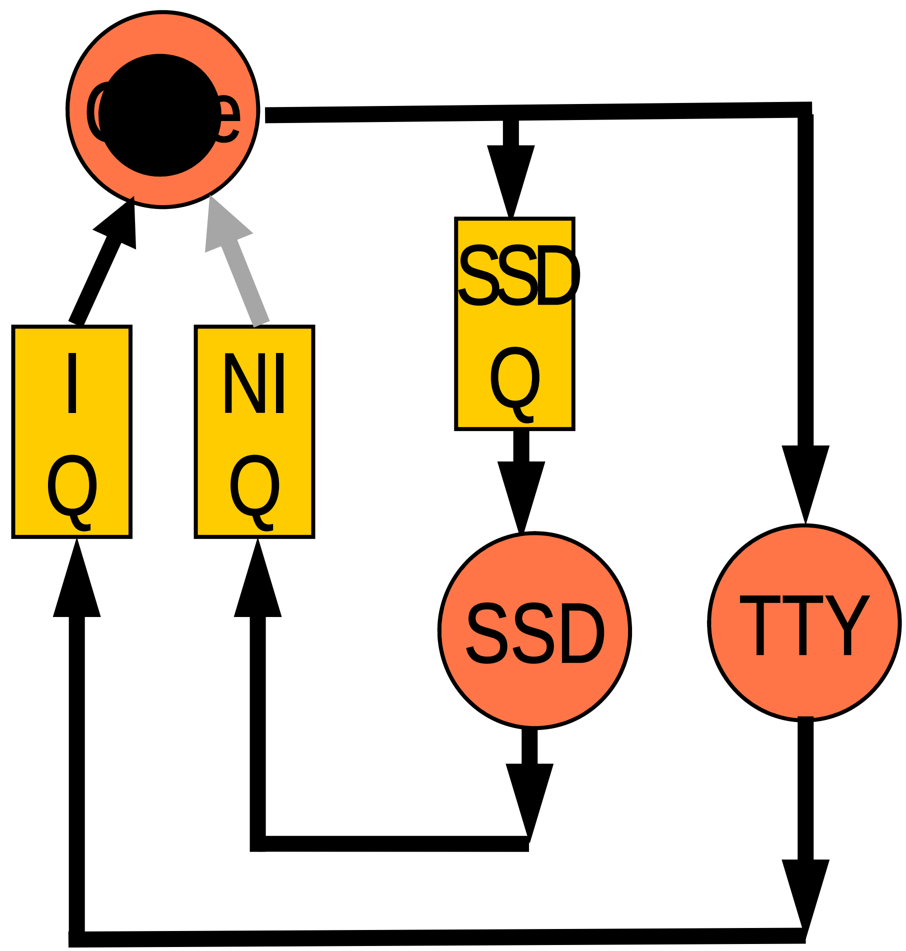
<u>T=220ms</u> Process 23 releases core gets TTY until = 5,220ms

```
NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20
```



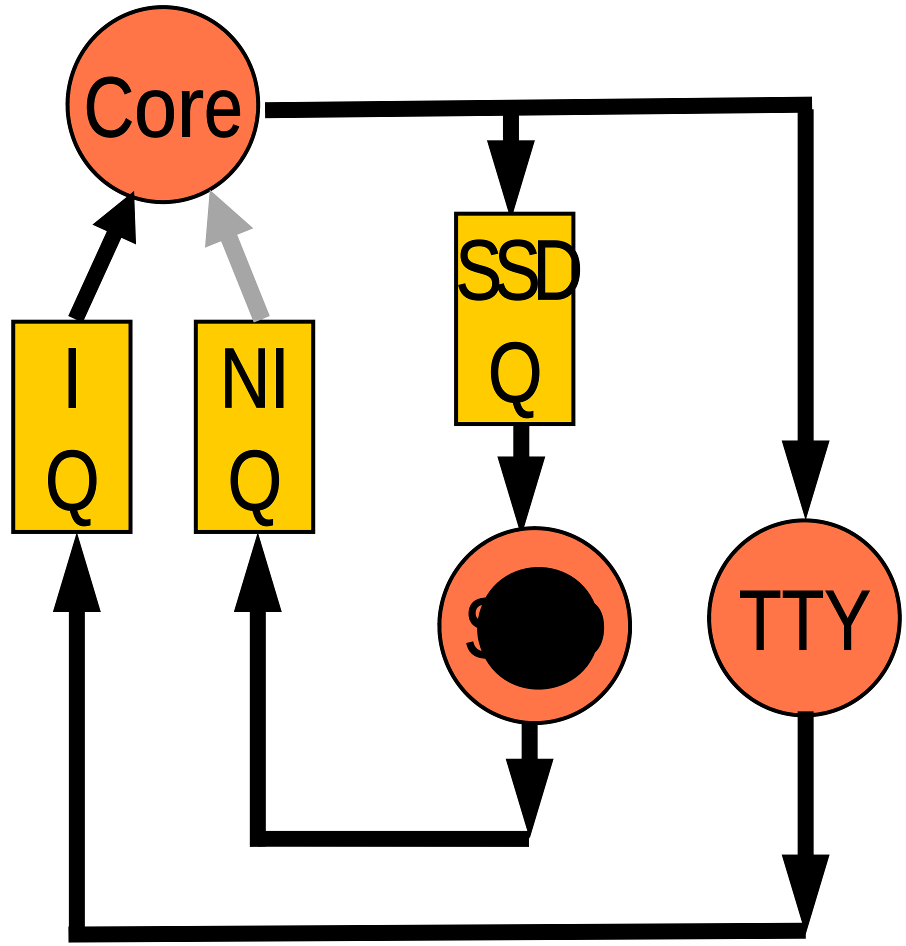
<u>T=5,220ms</u> Process 23 gets core until = 5,300ms

NCORES 1
START 120
PID 23
CORE 100
TTY 5000
<u>CORE 80</u>
SSD 1
CORE 30
SSD 1
CORE 20



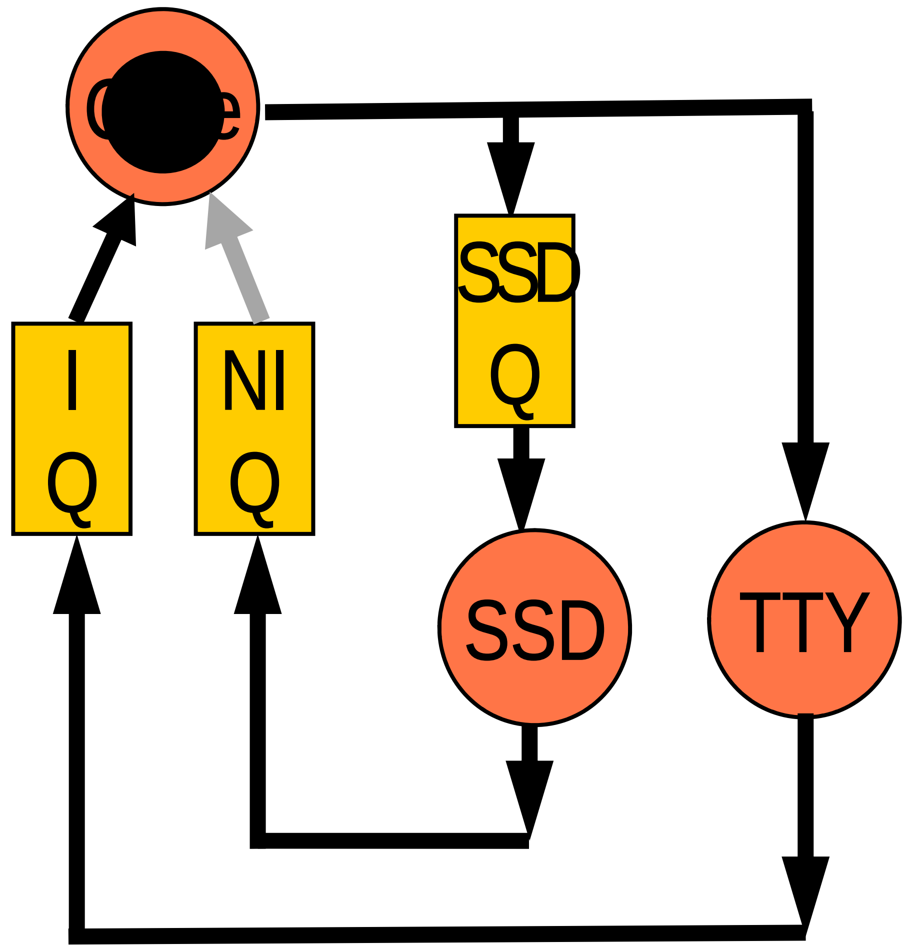
<u>T=5,300ms</u> Process 23 releases core
Gets SSD until = 5,301ms

```
NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20
```



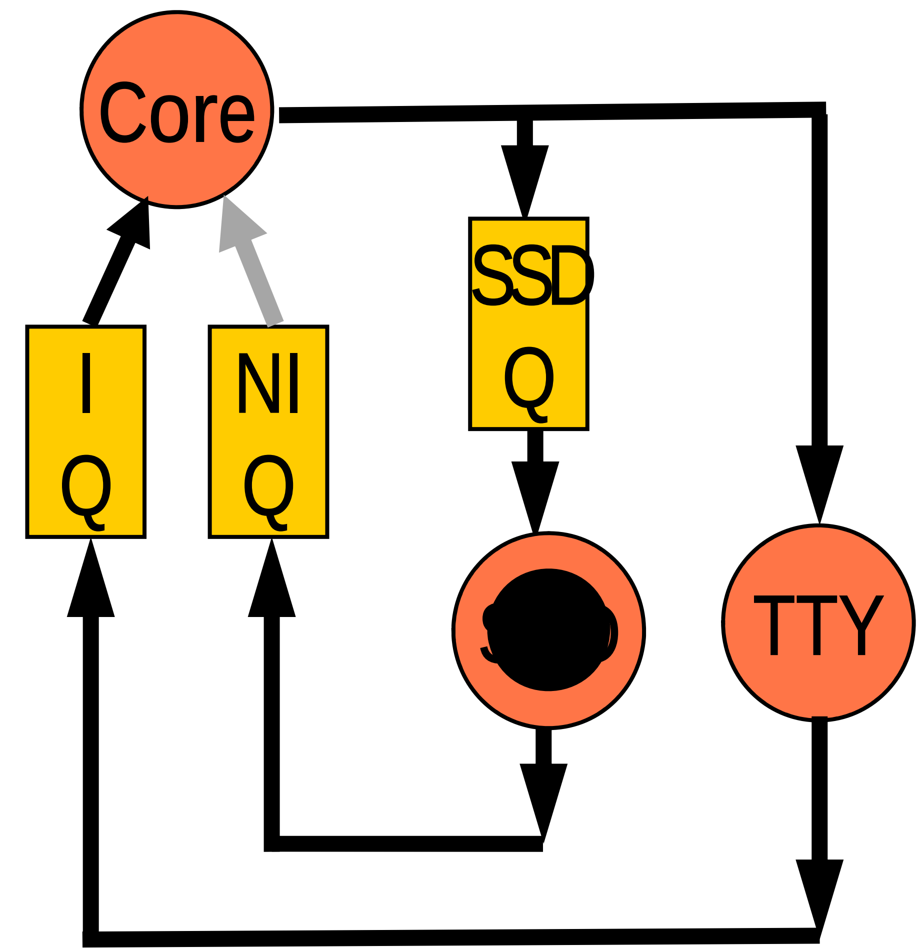
<u>T=5,301ms</u> Process releases SSD
Gets core until = 5,331ms

```
NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20
```



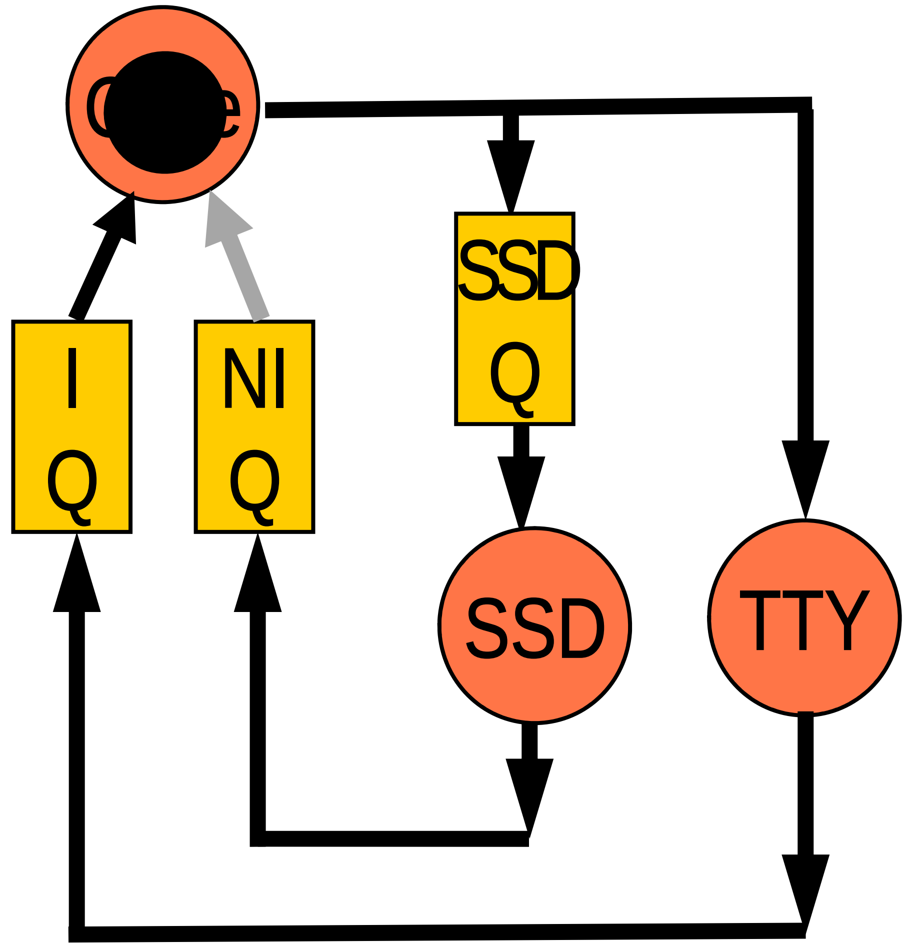
T=5,331ms Process releases core
Gets SSD until = 5,332ms

NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20



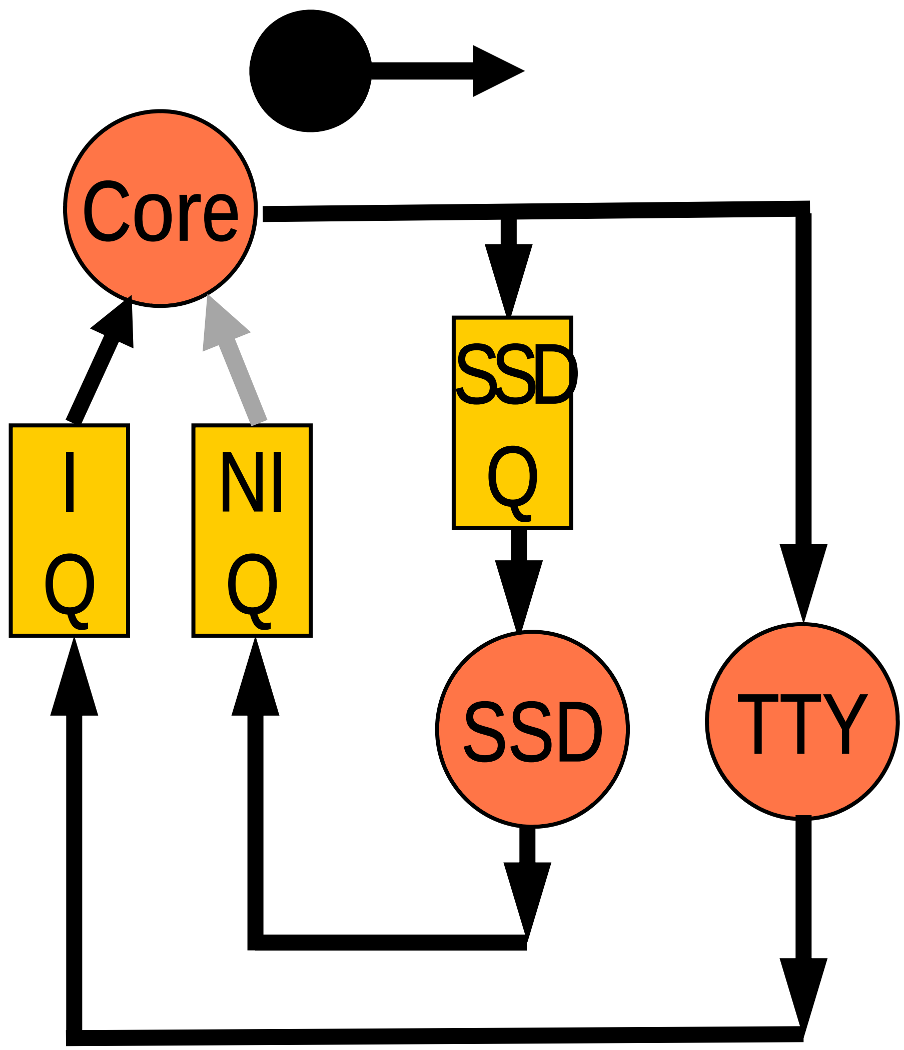
T=5,332ms Process releases SSD
Gets core until = 5,352ms

```
NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20
```



T=5,352ms Process terminates

```
NCORES 1
START 120
PID 23
CORE 100
TTY 5000
CORE 80
SSD 1
CORE 30
SSD 1
CORE 20
```



The completion events

- Completions are events
  - Mark end of a step
  - Notify it is time to move to next step
    - Next allocation step
- Scheduling is trivial when there is only one process
  - Not true otherwise

Handling two processes

```
NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START  5
PID 1
CORE 20
SSD 0
CPU 40
```

Process 0

Process 1

The solution (I)

Time	Command	Action(s)
0	`NEW 0`	Process 0 starts
0	`CORE 10`	P0 gets core until T=10ms
5	`NEW 5`	Process 1 starts
5	`CORE 20`	*Core is busy:*
		P1 enters NI queue
		P1 is in READY state
10	`SSD 1`	P0 releases the CPU
		Gets SSD until T = 11 ms
		P1 gets CPU until t = 30ms
		P1 is in RUNNING state

The solution (II)

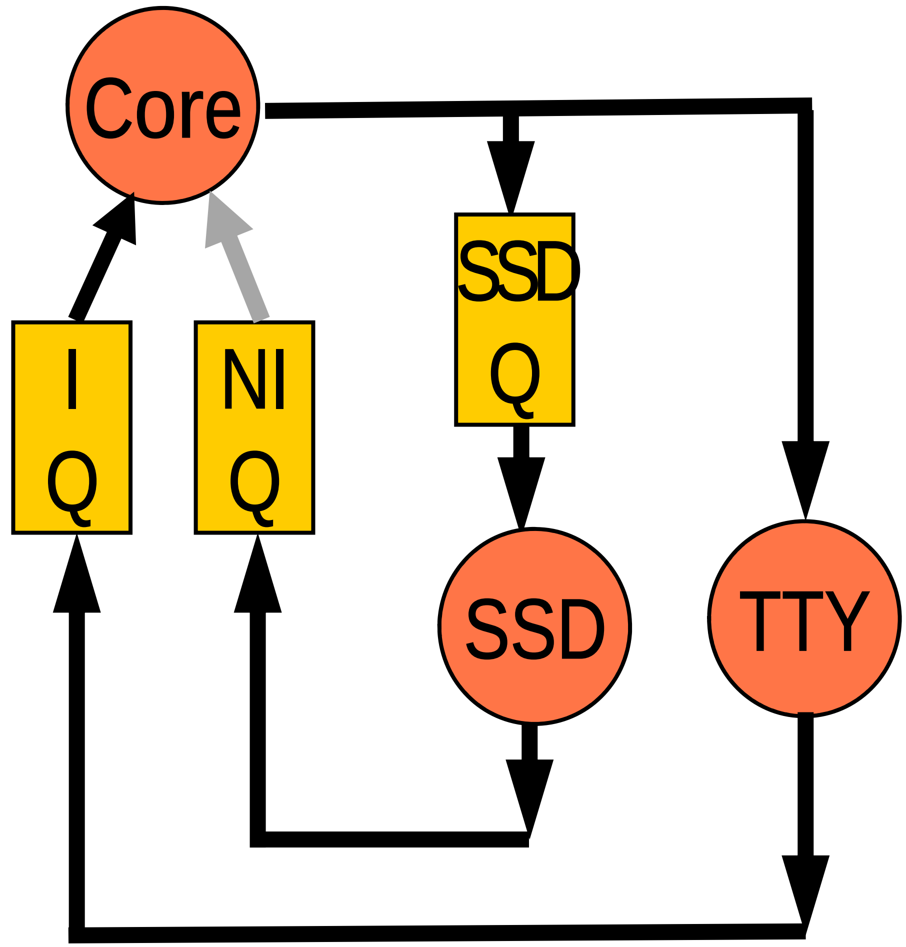
Time	Command	Action(s)
11	`CORE 30`	*CPU is busy:*
		P0 enters NI queue
		P0 is in READY state
30	`SSD 0`	P1 releases the CPU
		Gets SSD until T=30 ms
		P0 gets CPU until T=60ms
30	`CORE 40`	*CPU is busy:*
		P1 enters NI queue
		P1 is in READY state

The solution (III)

Time	Command	Action(s)
60		P0 releases CPU and terminates P1 gets CPU until T=100ms
100		P1 releases CPU and terminates

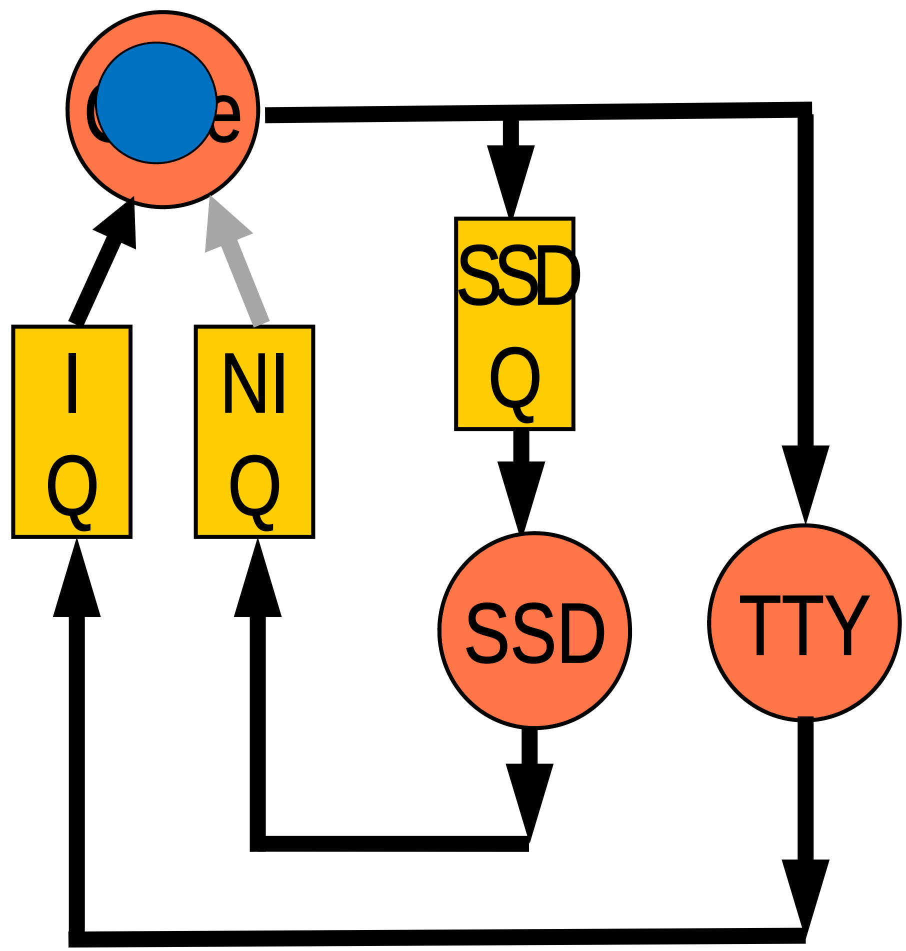
Another way to look at it

```
NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START  5
PID 1
CORE 20
SSD 0
CPU 40
```



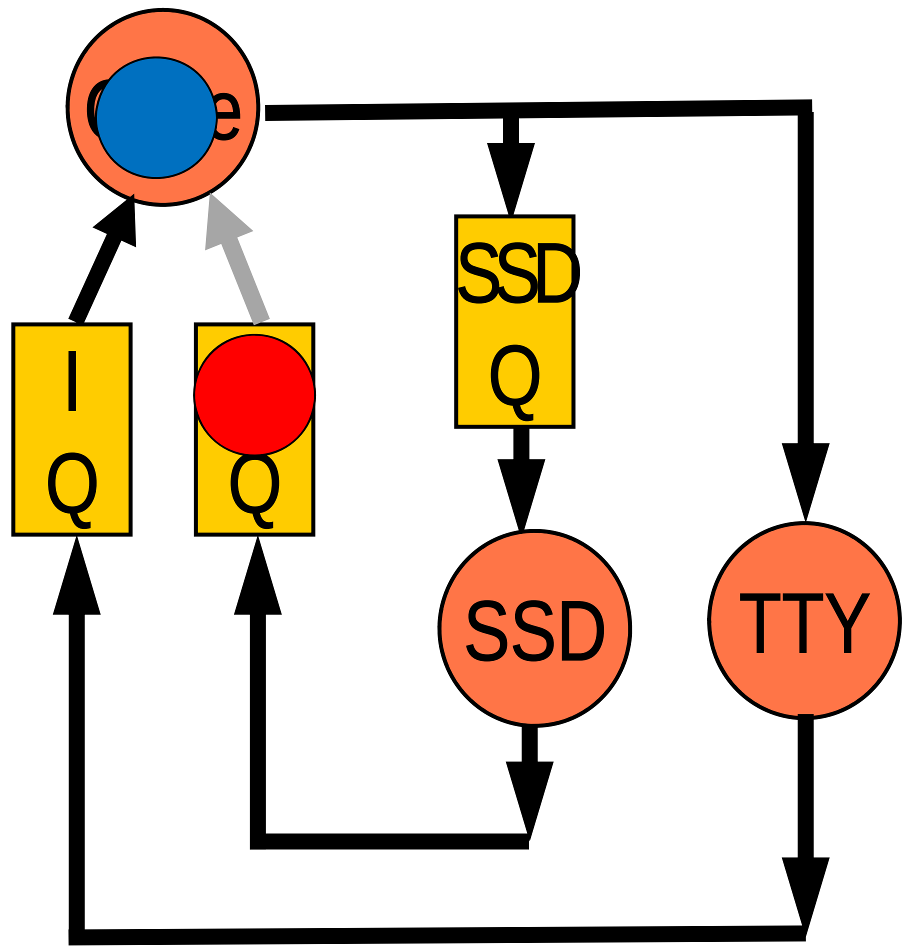
T= 0ms P0 gets core until T = 10ms

```
NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START  5
PID 1
CORE 20
SSD 0
CPU 40
```



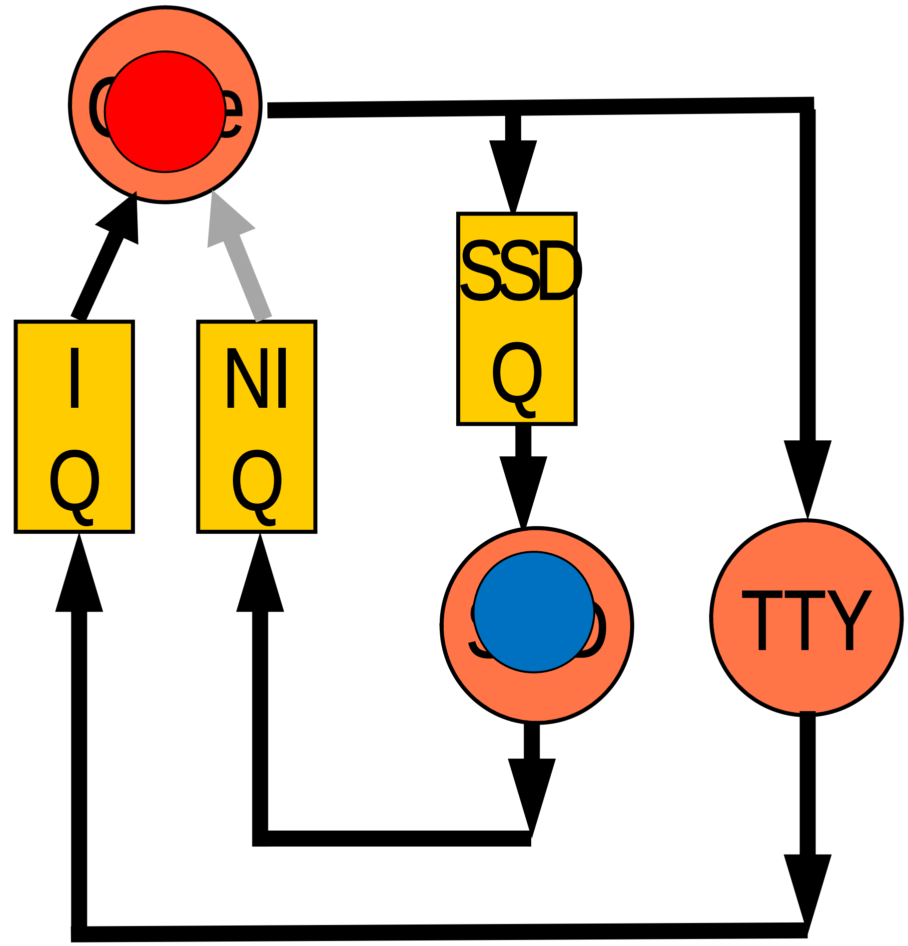
T=5ms P1 waits for P0 to release core at T=30ms

NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START 5
PID 1
CORE 20
SSD 0
CPU 40



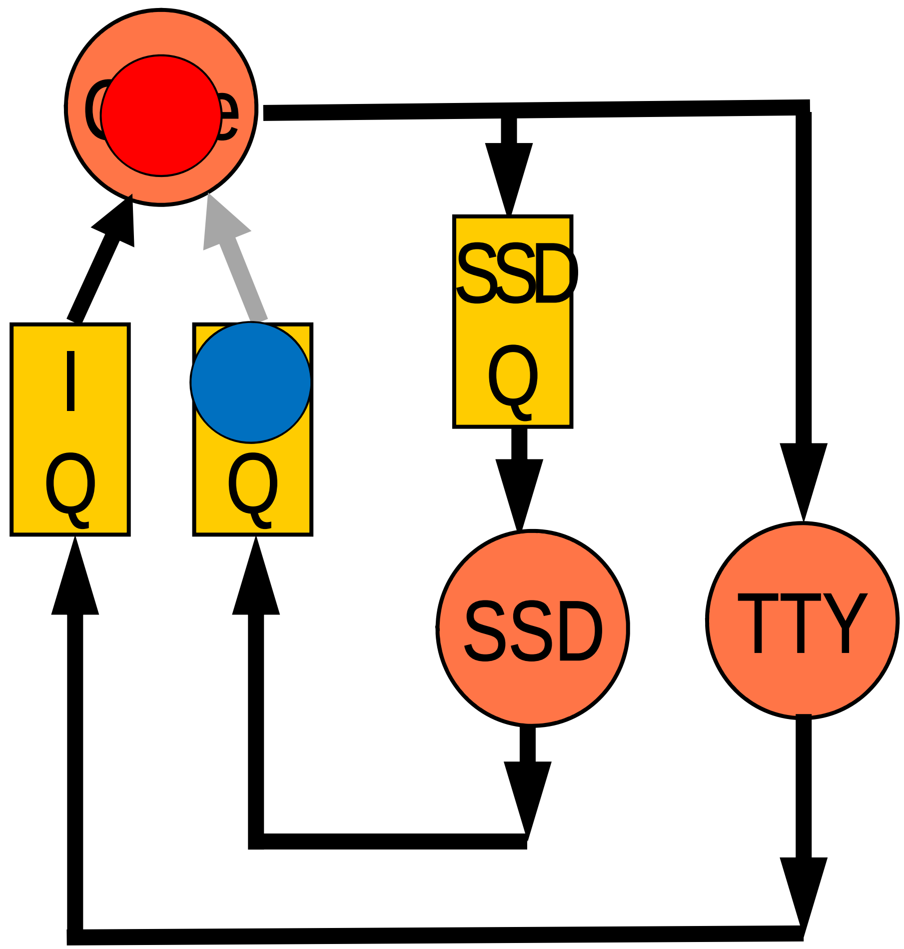
T=10ms P0 gets SSD until T=11ms
P1 gets core until T=30ms

```
NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START 5
PID 1
CORE 20
SSD 0
CPU 40
```



T=11ms P0 waits for P1 to release core at T=30ms

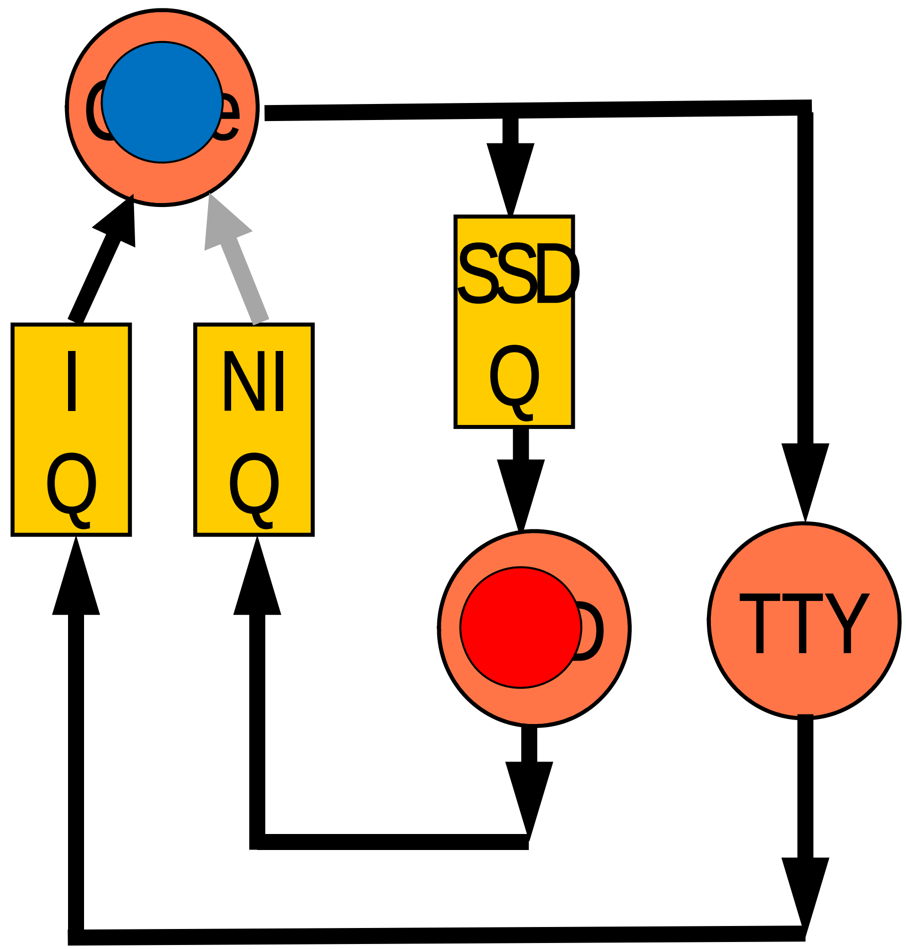
NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START 5
PID 1
CORE 20
SSD 0
CPU 40



<u>T=30ms</u> P1 gets SSD until T=30ms

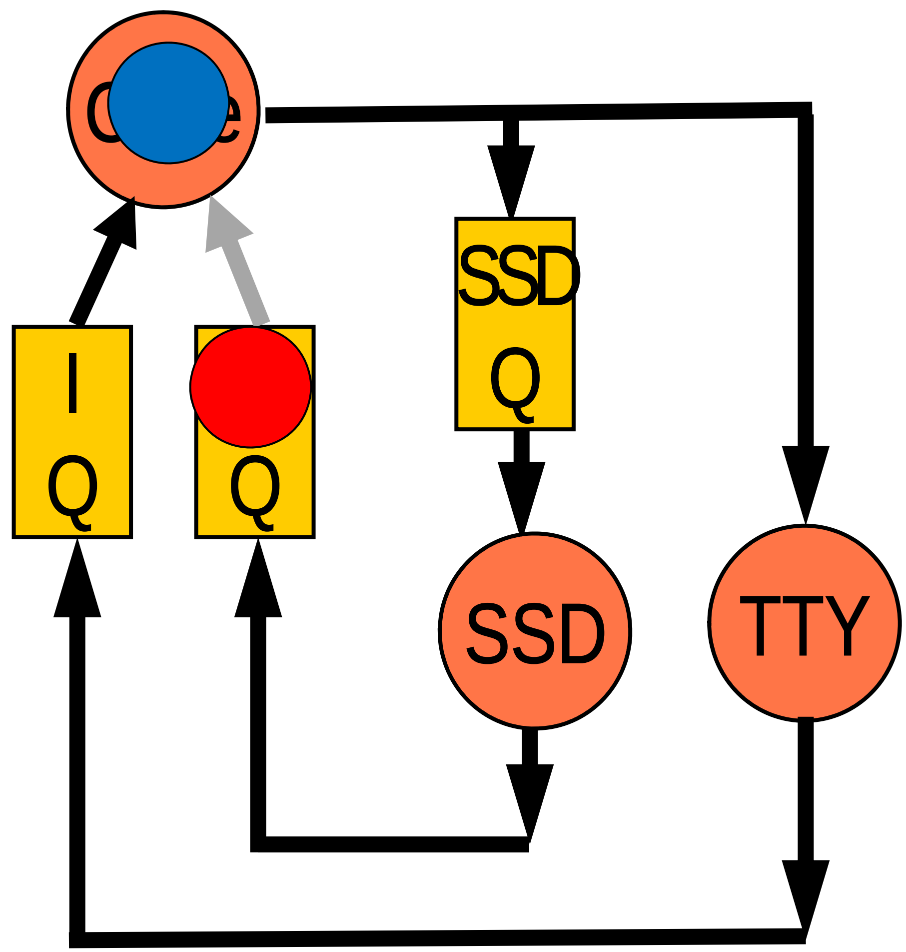
P0 gets core until T=60ms

NCORES 1
START 0
PID 0
CORE 10
SSD 1
<u>CORE 30</u>
START 5
PID 1
CORE 20
<u>SSD 0</u>
CPU 40



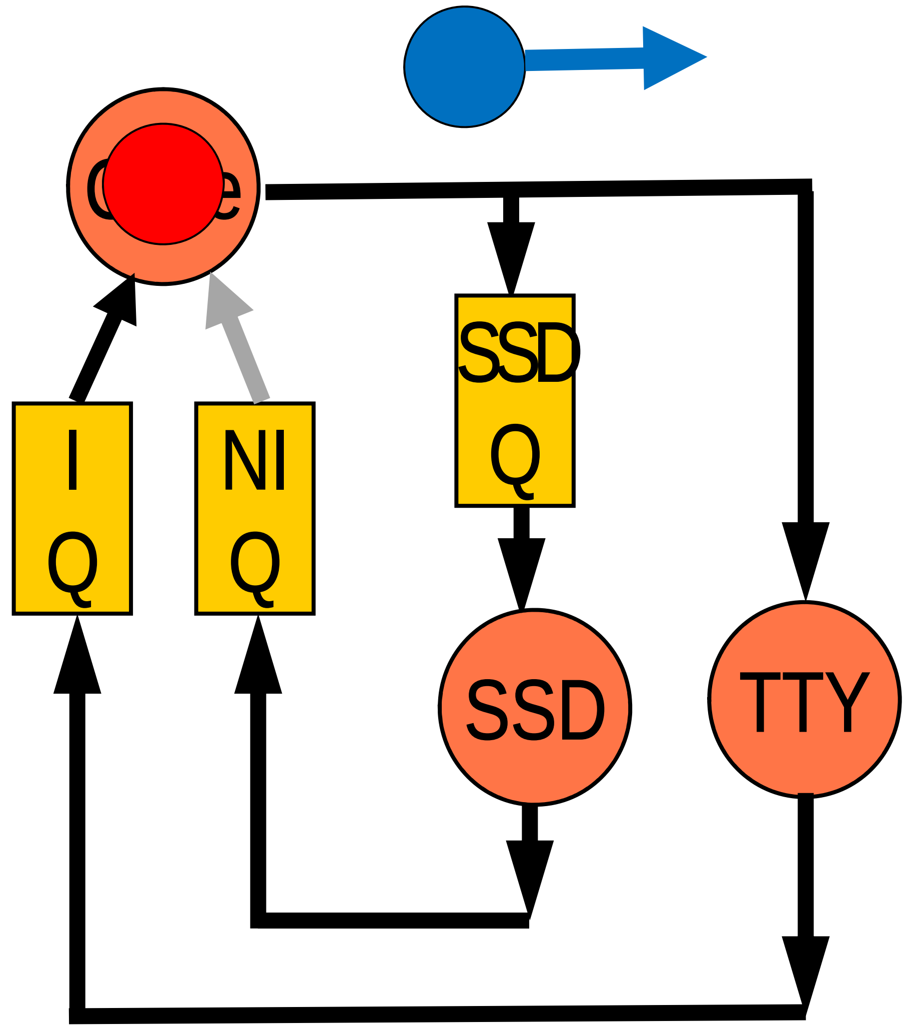
<u>**T=30ms**</u> P1 waits for P0 to release core at T=60ms

```
NCORES 1
  START 0
  PID 0
  CORE 10
  SSD 1
  CORE 30
  START 5
  PID 1
  CORE 20
  SSD 0
  CPU 40
```



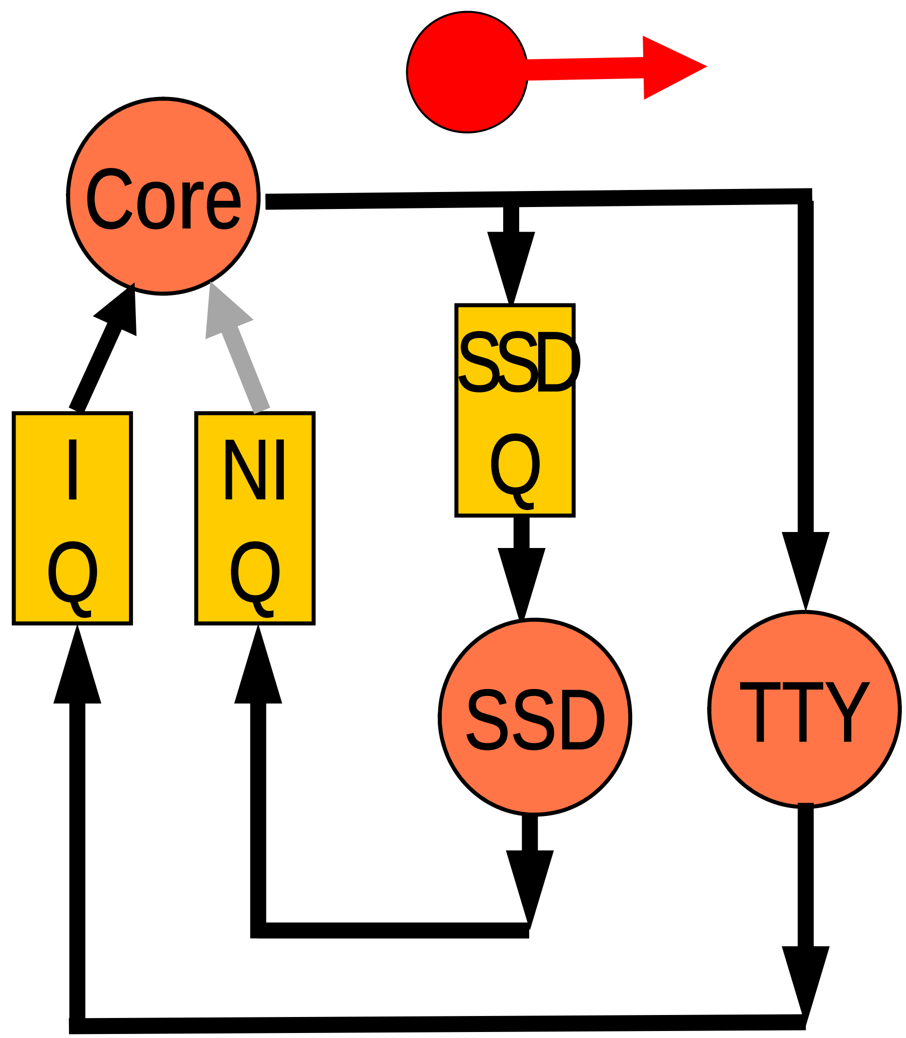
T=60ms P0 to release core and terminates
P1 gets core until T=100ms

```
NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START 5
PID 1
CORE 20
SSD 0
CPU 40
```



T=100ms P1 releases core and terminates

```
NCORES 1
START 0
PID 0
CORE 10
SSD 1
CORE 30
START 5
PID 1
CORE 20
SSD 0
CPU 40
```

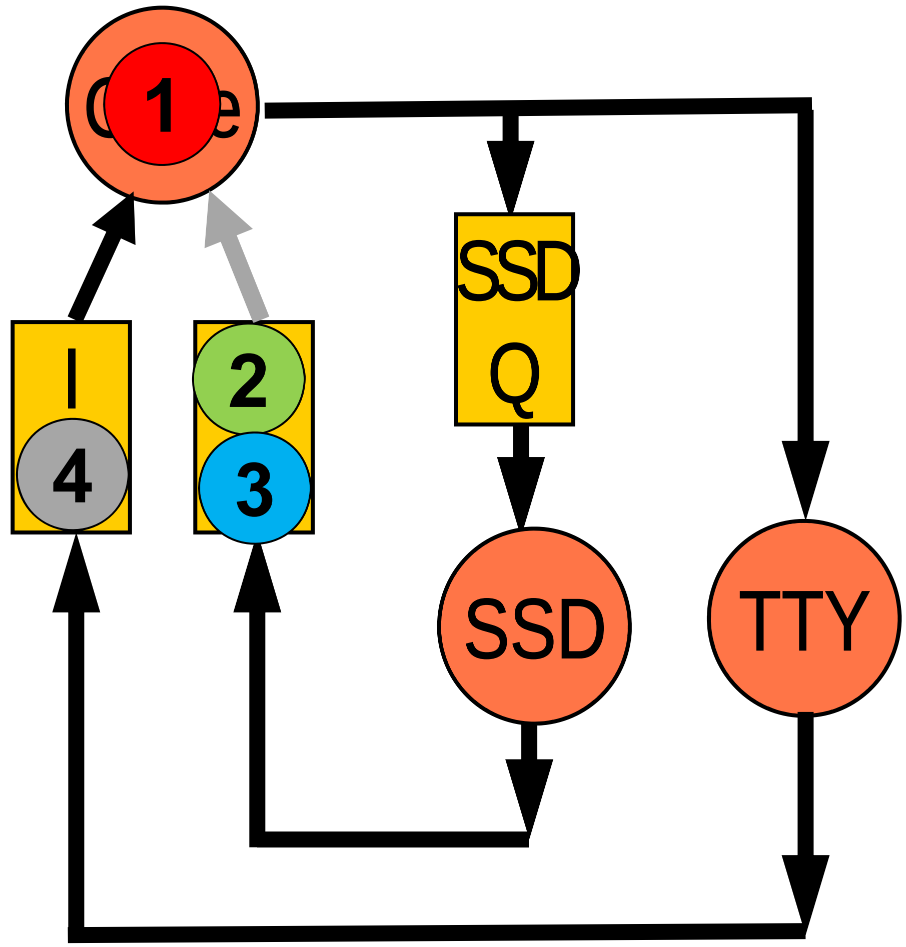


Scheduling the CPU

- Have two queues
  - Interactive queue
    - Contains processes that have just completed a user interaction
    - Higher priority
  - Interactive queue
    - Contains all other processes
    - Lower priority

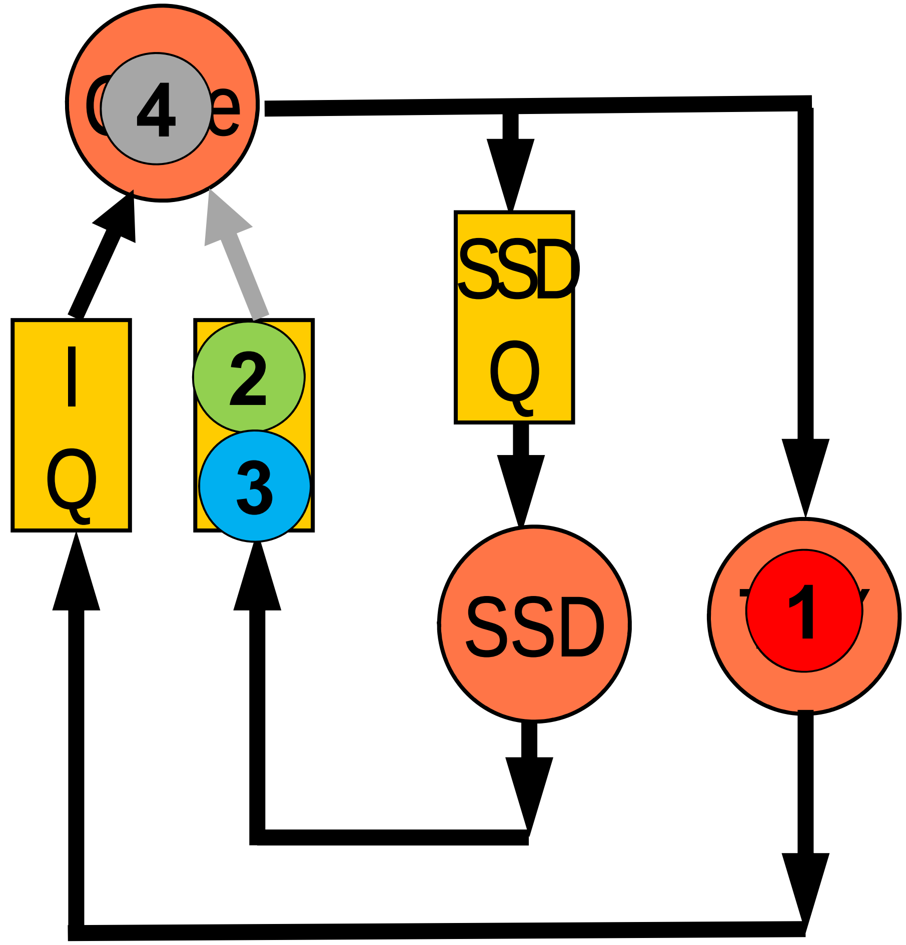
Example (I)

- When process 1 releases a core, process 4 will get it ahead of 2 and 3.



Example (II)

- If process 1 returns to READY state before 4 releases its core, it will get this core ahead of 2 and 3
- Otherwise process 2 will get the core



Handling parallel activities

- We only need to consider start times and completion times of each computational step
- Completion times are the most important
  - Release a device
  - Initiate the next request
    - Can be immediately satisfied if requested device is free
    - May require process to wait for device
      - Ready queue or disk queue

ENGINEERING THE SIMULATION

Simulating time

- ***Absolutely nothing happens to our model between two successive "<u>events</u>"***
- ***Events are***
  - Arrival of a new process
  - Completion of a computing step
  - Completion of a SSD access
  - Completion of a user interaction
- We associate an ***event routine*** with each event

Organizing our program (I)

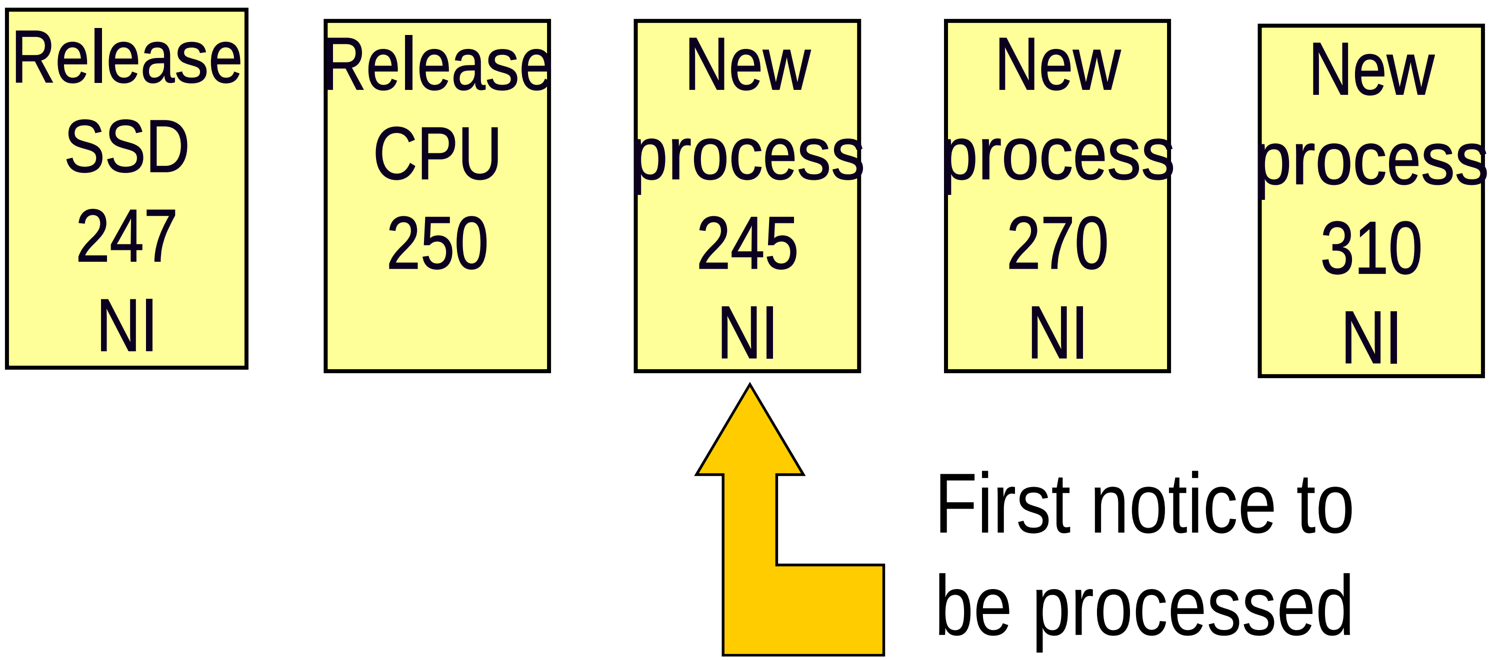
- Most steps of simulation involve scheduling future completion events
- Associate with each completion event an event notice
  - **Time of event**
  - **Device**
  - **Process ID**
  - **Interactive/Non-interactive bit**

Organizing our program (II)

- Do the same with process starts
  - **Time of event**
  - **Process start**
  - **Process ID**
  - **Interactive/non-interactive bit**
    - **Set to non-interactive**

Organizing our program (III)

- Process all event notices in chronological order



Organizing our program (IV)

- Overall organization of main program

```
read in input file
schedule all process starts
while (event list is not empty) {
      process next event in list
} // while
print simulation results
```

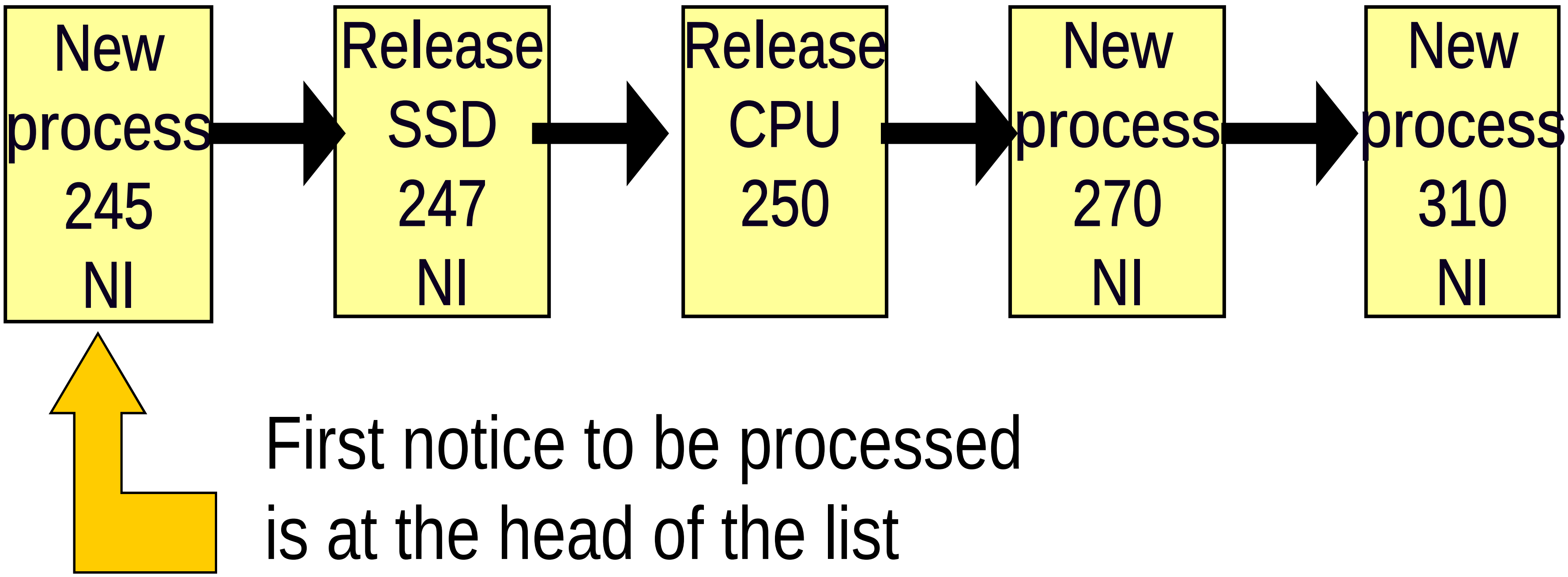
Organizing our event list (I)

- As a priority queue

- Associating a completion time
  - With each core request
  - With each SSD request
  - With each user interaction
  - With the each new process arrival

Organizing our event list

- Process all event notices in chronological order



Arrival event routine

```
arrival(time, proc_id) {
    process first request of new process
}  // arrival
```

Core request routine

```
core_request(how_long, proc_id, isinteract){
     if (nfreecores > 0) {
          nfreecores--;
          schedule completion at time
          current_time + how_long
          for process proc_id;
     } else {
          if (isinteract == interactive) {
               queue proc_id in i_queue
          } else {
               queue proc_id in ni_queue
          } // inner if
     }  // outer if
} // core_request
```

Core completion routine

```
core_release (proc_id){
     if (i_queue is not empty) {
          pop first request in i_queue
         schedule its completion at
          current_time + how_long;
     } else if (ni_queue is not empty {
          pop first request in ni_queue
          schedule its completion at
          current_time + how_long;
     } else {
          nfreecores++;
     } //if
     process next process request;
} // core_release
```

SSD request routine

```
ssd_request(how_long, proc_id){
     if (ssd == FREE) {
          ssd = BUSY;
          schedule completion at time
          current_time + how_long
          for process proc_id;
     } else {
          queue process proc_id in
          ssd queue;
     }  // if
} // ssd_request
```

SSD completion routine

```
ssd_release (proc_id, &isinteract){
     isinteract = non_interactive;
     if (ssd queue is not empty) {
          pop first request in ssd queue
          schedule its completion at
          current_time + how_long;
     } else {
          ssd = FREE;
     } // if
     process next process request;
} // ssd_release
```

User request routine

```
user_request (how_long, proc_id){
     schedule completion at time
     current_time + how_long
     for process process_id;
} // user_request
```

User completion routine

```
user_release (proc_id, &isinteract){
     isinteract = interactive;
     process next process request;
} // user_release
```

Overview (I)

- ***Input module***
  - Schedules all ARRIVAL events
- ***Main loop***
  - Pops next event from event list
    - ARRIVAL
    - CORE completion
    - SSD completion
    - TTY completion

Overview (II)

- ***ARRIVAL event***
  - Starts a core request

Overview (III)

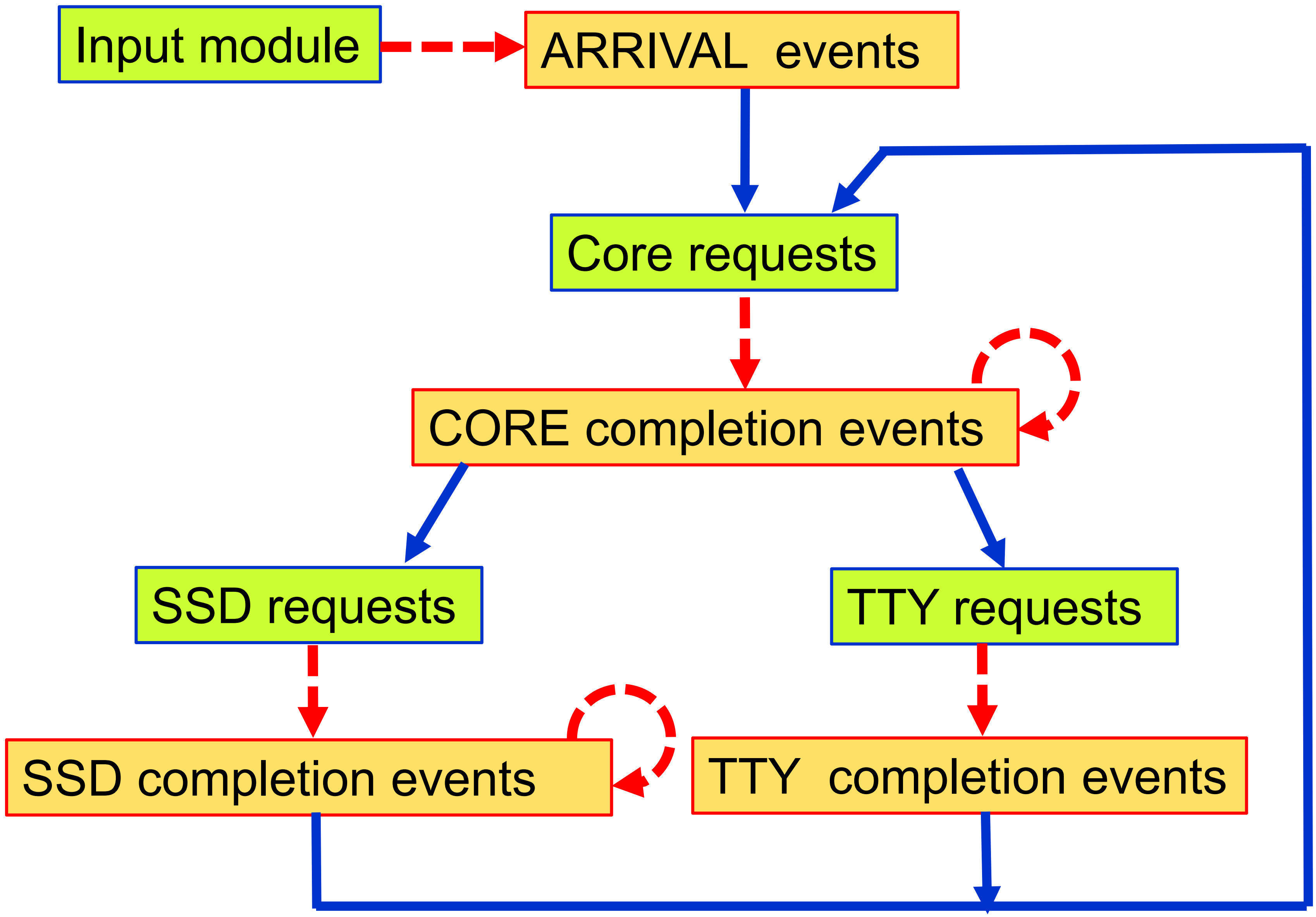
- ***Core request***
  - If a core is free
    - Schedules a CORE completion event
- ***CORE completion event***
  - May schedule a CORE completion event
  - Starts next request
    - SSD or TTY

Overview (IV)

- ***SSD request***
  - If SSD is free
    - Schedules an SSD completion event
- ***SSD completion event***
  - May schedule a SSD completion event
  - Starts next request
    - CORE

Overview (V)

- ***TTY request***
  - Schedules an TTY completion event
- ***TTY completion event***
  - Starts next request
    - CORE


Input module
ARRIVAL events
Core requests
CORE completion events
SSD requests
TTY requests
SSD completion events
TTY completion events

Explanations

- Green boxes represent conventional functions
- Amber boxes represent events and their associated functions
- Continuous blue arrows represent regular function calls
- Red dashed lines represent the scheduling of specific eventc

Finding the next event

- If you do not use a priority list for your events, you can find the next event to process by searching the **lowest value** in
  - The process start times in the process table
  - The completion times in the device table
  - The display completion timed in the process table

AN IMPLEMENTATION

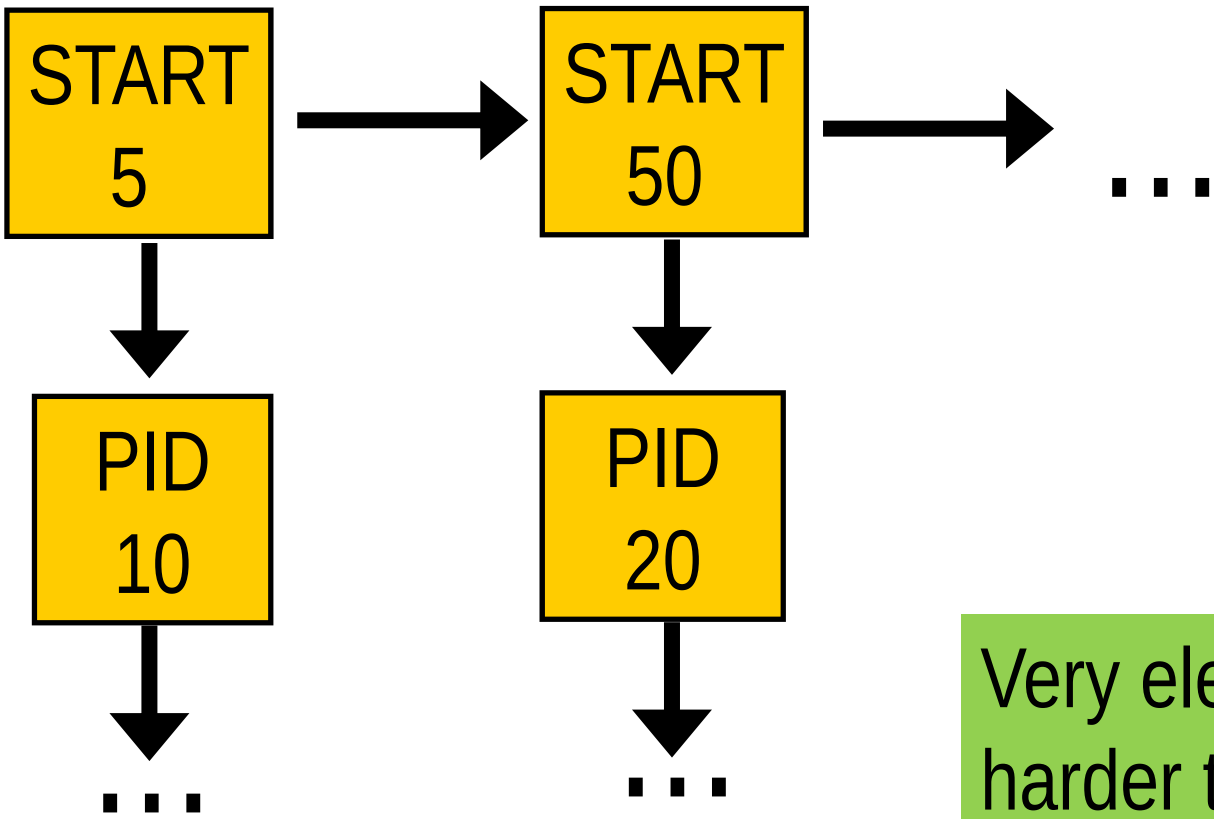
- My main data structures would include:
  - An input table
  - A process table
  - A device table

The input table

- Stores the input data
- Line indices are used in process table

Operation	Parameter
START	5
PID	10
CORE	20
SSD	0
CORE	20
START	50
PID	20
…	…

A full list implementation of the input table



Very elegant but harder to debug

The process table (I)

PID	Start Time	First Line	Last Line	Current Line	State
10	5	0	4	varies	varies
20	50	5	…	…	
	…	…	…	…	

The process table (II)

- One line per process
  - ***Line index is process sequence number*!**
- First column has start time of process
- First line, last line and current line respectively identify first line, last line and current line of the process in the input table
- Last column is for the current state of the process (READY, RUNNING or BLOCKED)

The device table (I)

Device	Status	Busy times
CPU	P0	15
SSD	-	-

The device table (II)

- One line per device
  - ***Line index identifies the device***
- First column has status of device
  - Number of free cores for CPU
  - Free/busy for SSD
- Last column is for the total of all busy times

READING YOUR INPUT

- You *must* use I/O redirection
  - **`assign1 < input_file`**
- ***Advantages***
  - Very flexible
  - Programmers write their code as if it was read from standard input
    - No need to mess with **`fopen()`**, **`argc`** and **`argcv`**

Detecting the end of data

- The easiest ways to do it
- If you use **`scanf()`**
  - **`scanf(…)`** returns **`0`** once it reaches the end of data
    - **`if(scanf(…))`**
- If you use **`cin`**
  - **`cin`** returns **`0`** once it reaches the end of data
    - **`while (cin >> keyword >> time )`**