COSC 3360/6310
Wednesday, January 27

Announcements

- Trivial change to assignment 1 specifications
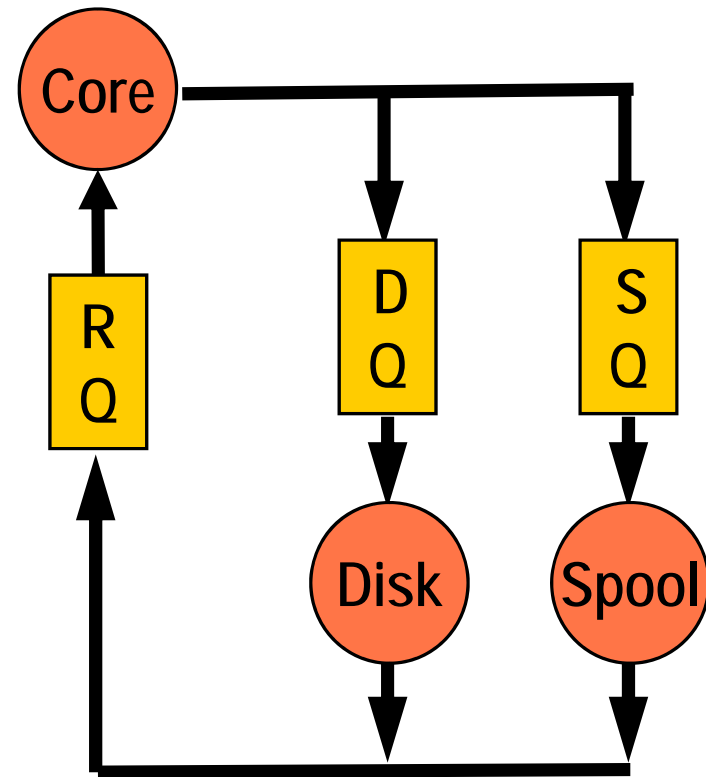 - ☐ Sample input now has first job ending with a CORE step

The first spring 2021 assignment explained

Jehan-François Pâris

jfparis@uh.edu

The model

We have

- One single core CPU
- One disk
- One print spooler
- Three queues
 - CPU (Ready queue)
  - Disk
  - Spooler

A deck of punched cards

A very simple case

- **MPL   1   `// no multiprogramming`**
  JOB   1   `// new job`
  CORE   100   `// request CORE for 100ms`
  DISK   0   `// no wait disk request`
  CORE   30   `// request CORE for 30ms`
  DISK   7   `// request DISK for 7ms`
  CORE   20   `// request CORE for 20s`
  PRINT   1000   `// print spooler request`
  CORE   20   `// request CORE for 30ms`

Job/process states

A job can be

- **_Running_**
  - ❑ It *o*ccupies a core

- **_Ready_**
  - ❑ It waits for a core

- **_Blocked_**
  - ❑ It waits for the completion of a system request
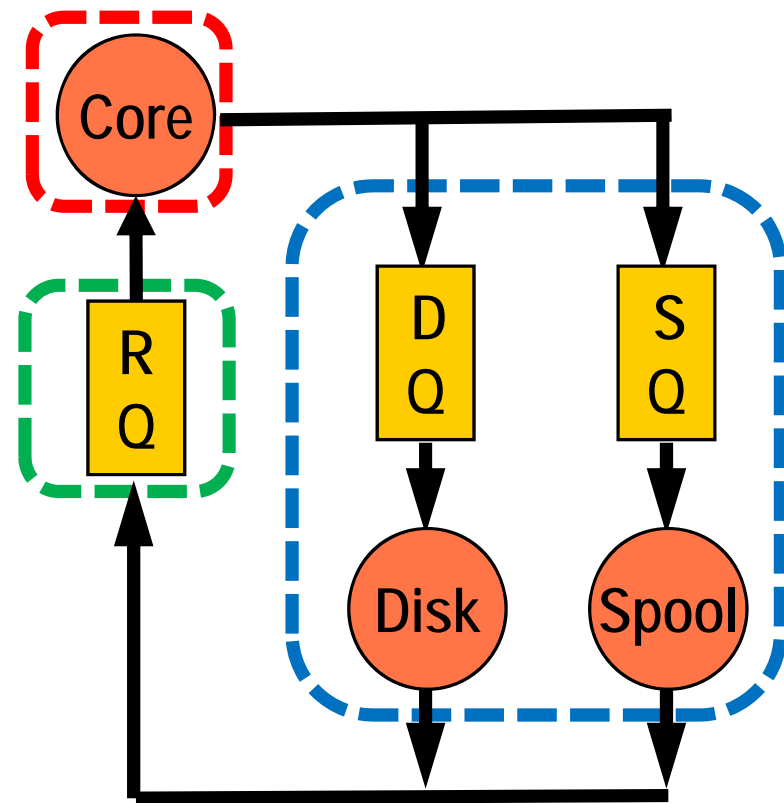
Starting the simulation

- **MPL   1**
  - ☐ Jobs are processed one by one
  - ☐ Set (simulated) time to zero

- **JOB 1**
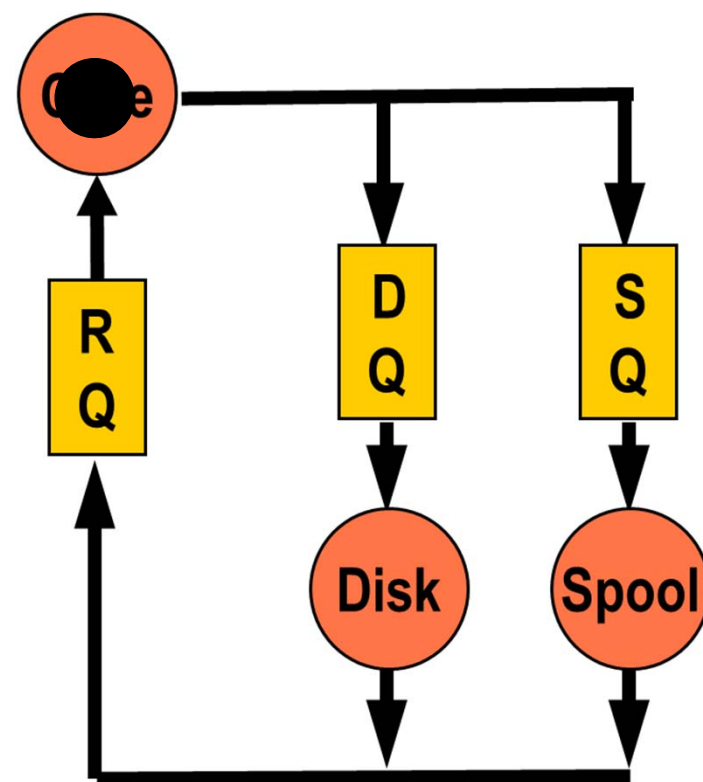 - ☐ Fetch and start processing job

Output

- **Job 1 starts at time 0 ms**
  Job Table:
  There are no other active jobs

Allocate core to job 1 for 100ms at $t = 0$ms

- MPL   1
  JOB   1
  CORE   100
  DISK   0
  CORE   30
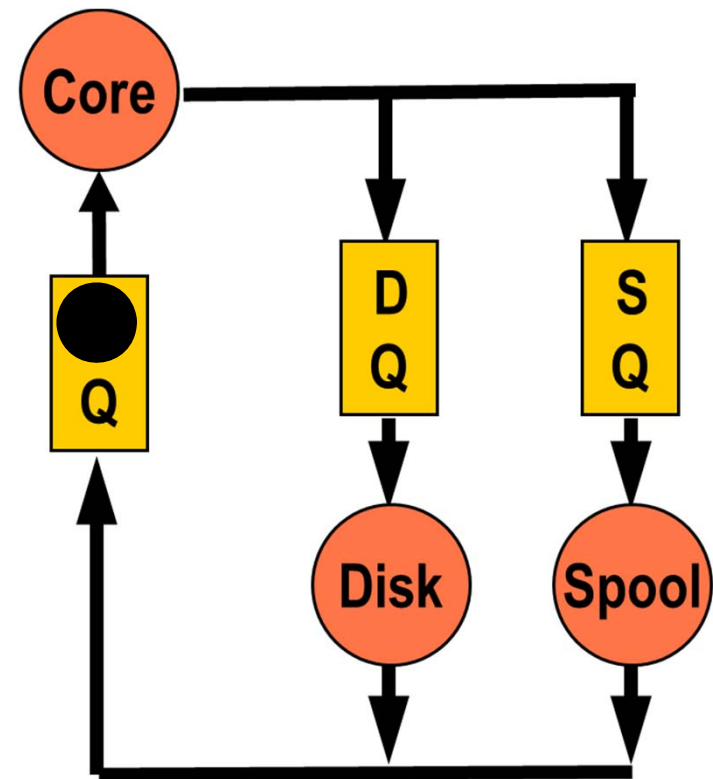 DISK   7
  CORE   20
  PRINT   1000
  CORE   20

What's next?

- Must wait end of core step at $t = 100$ms
- Can then perform the next step

- **Set $t = 100$ms**

Zero-time disk access at $t = 100$ms

- MPL   1
  JOB   1
  CORE   100
  __DISK   0__
  CORE   30
  DISK   7
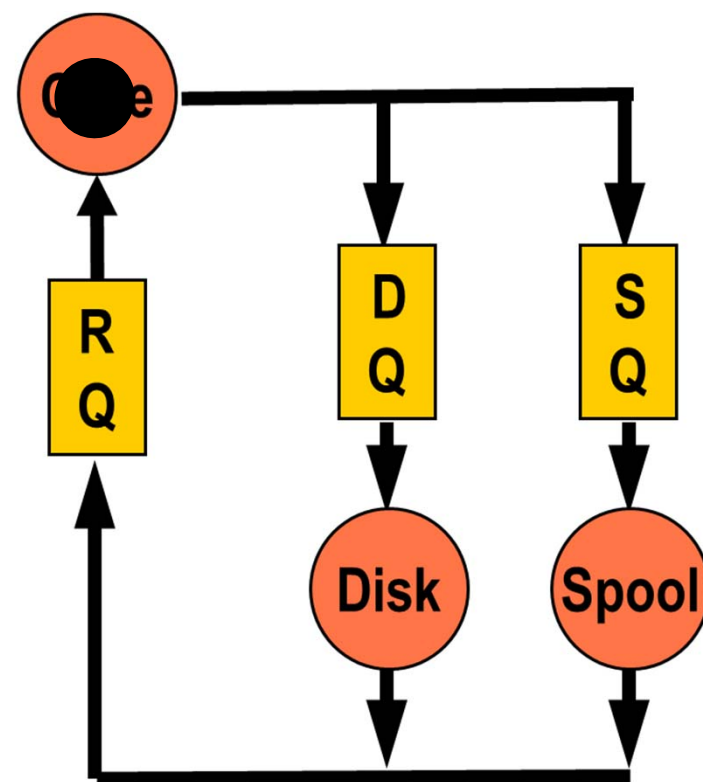 CORE   20
  PRINT   1000
  CORE   20

What's next?

- Immediately perform the next step
- At t = **100ms**

> A zero-delay disk request (DISK 0)
> ❑ *Always bypasses* the disk
> ❑ *Never waits* for it

Allocate core to job 1 for 30ms at t = 100ms

- MPL 1
  JOB 1
  CORE 100
  DISK 0
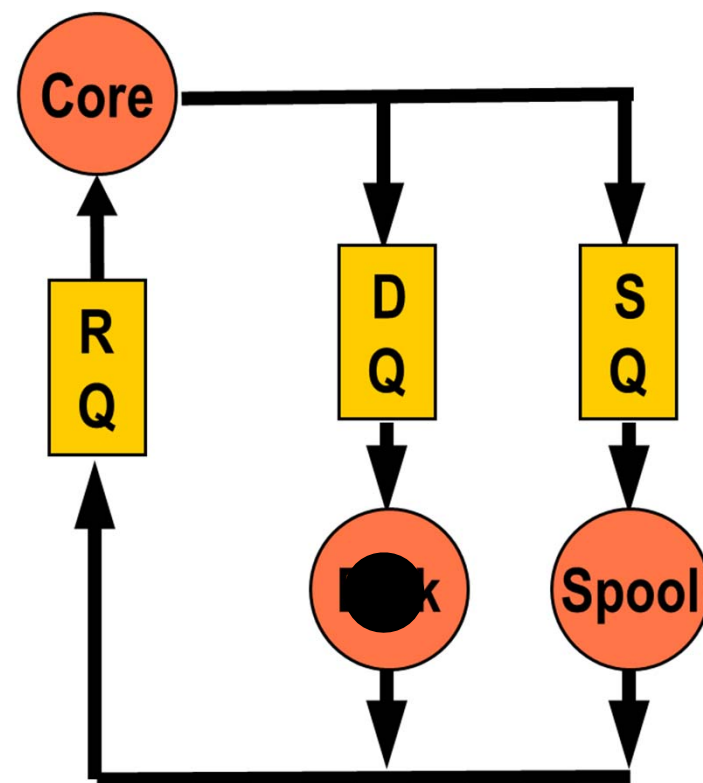 <u>CORE 30</u>
  DISK 7
  CORE 20
  PRINT 1000
  CORE 20

What's next?

- Must wait end of core step at $t = 100ms + 30ms = 130ms$
- Can then perform the next step

- **Set $t = 130ms$**

Allocate disk to job 1 for 7ms at t = 130ms

- MPL   1
  JOB   1
  CORE   100
  DISK   0
  CORE   30
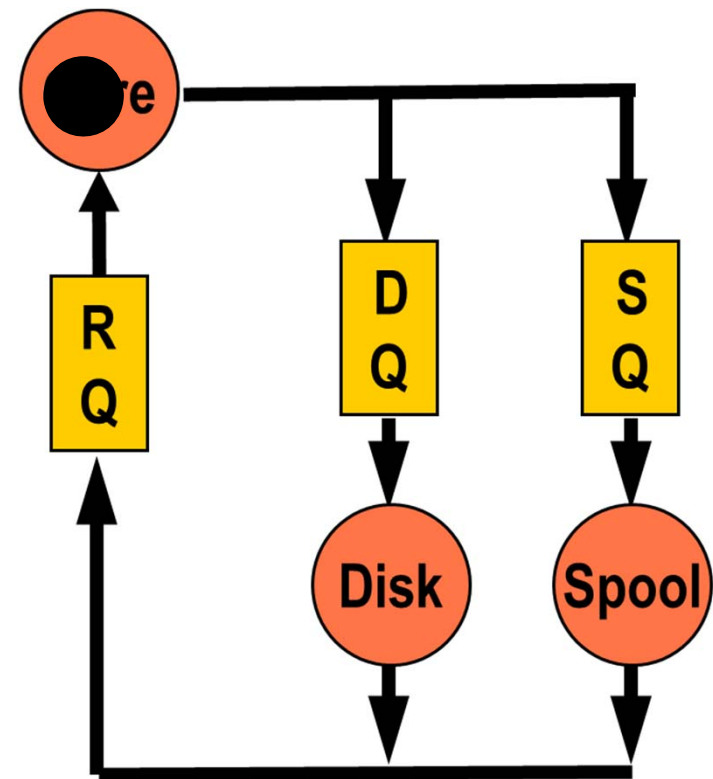 DISK   7
  CORE   20
  PRINT   1000
  CORE   20

What's next?

- Must wait end of disk step at $t = 130$ms $+ 7$ms $= \underline{\textbf{137ms}}$
- Can then perform the next step

- **Set $t = 137$ms**

Allocate core to job 1 for 20ms at t = 137ms

- MPL   1
  JOB   1
  CORE   100
  DISK   0
  CORE   30
  DISK   7
  CORE   20
  PRINT   1000
  CORE   20

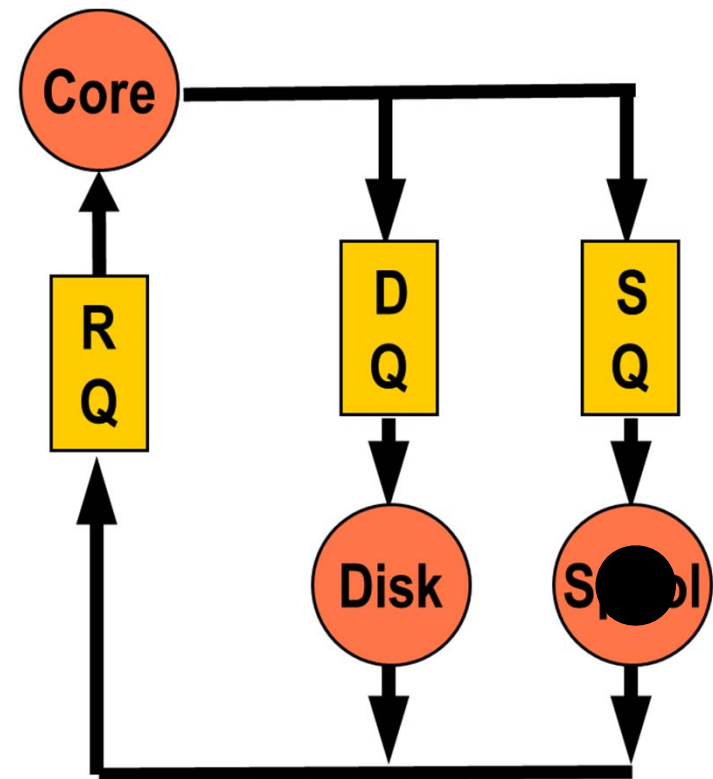# What's next?

- Must wait end of core step at $t = 137\text{ms} + 20\text{ms} = \textbf{157ms}$
- Can then perform the next step

☐**Set $t = \textbf{157ms}$**

Allocate spooler to job 1 for 1000ms at t = 157ms

- MPL   1
  JOB   1
  CORE   100
  DISK   0
  CORE   30
  DISK   7
  CORE   20
  PRINT   1000
  CORE   20

What's next?

- Must wait end of spooler step at t = 157ms + 1000ms = **<u>1157ms</u>**
- Can then perform the next step

- **Set t = 1157ms**

Allocate core to job 1 for 20ms at t = 1157ms

- MPL   1
  JOB   1
  CORE   100
  DISK   0
  CORE   30
  DISK   7
  CORE   20
- **PRINT   1000**
  <u>**CORE   20**</u>
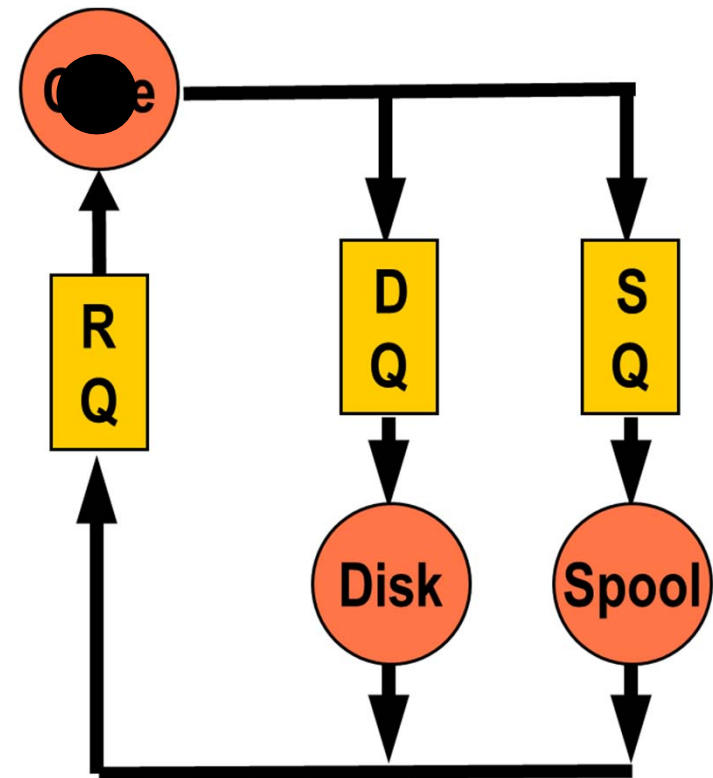
What's next?

- Job 1 has terminated
- Print the required output

Output

- Job 1 terminates at time 1177ms
  Job Table:
  There are no active jobs

- SUMMARY:
  Total elapsed time: 1177ms
  Number of jobs that have completed: 1
  Total number of disk accesses: 2
  Core utilization: 0.13

Things become interesting

- **MPL   2**   // memory can hold 2 jobs
  JOB   1   // new job
  CORE   100   // request CORE for 100ms
  DISK   9   // request DISK for 9ms
  CORE   30   // request CORE for 30ms
  JOB   5   // new job
  CORE   20   // request CORE for 20s
  DISK   8   // request DISK for 8ms
  CORE   20   // request CORE for 30ms

Starting the simulation

- **MPL   2**
  - ☐ Two jobs now compete for system resource
    - ■ Better utilization of CPU, disk, …
    - ■ Price to pay is **queuing delays**
  - ☐ Set (simulated) time to zero
- **JOB 1**
  - ☐ Fetch and start processing job 1
- **JOB 5**
  - ☐ Fetch and put on hold job 5 in ready queue

Output

- **Job 1 starts at time 0 ms**
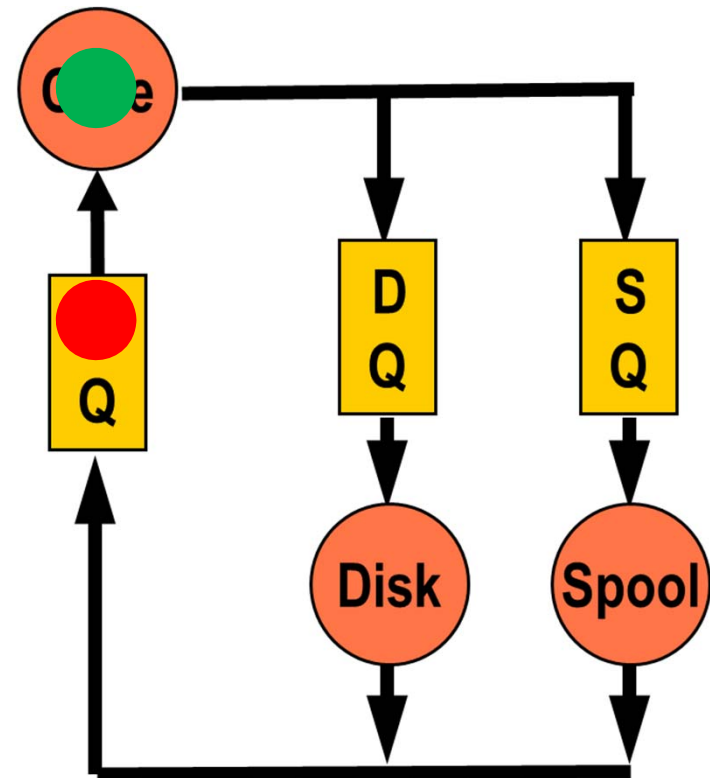 **Job Table:**
  There are no other active jobs

- **Job 5 starts at time 0 ms**
  Job Table:
  Job 1 is RUNNING (Will also accept **Job 1 is READY**)

Allocate core to job 1 for 100ms at $t = 0$ms
Note job 5 requests core

- MPL   2
  JOB   1
  CORE   100
  DISK   9
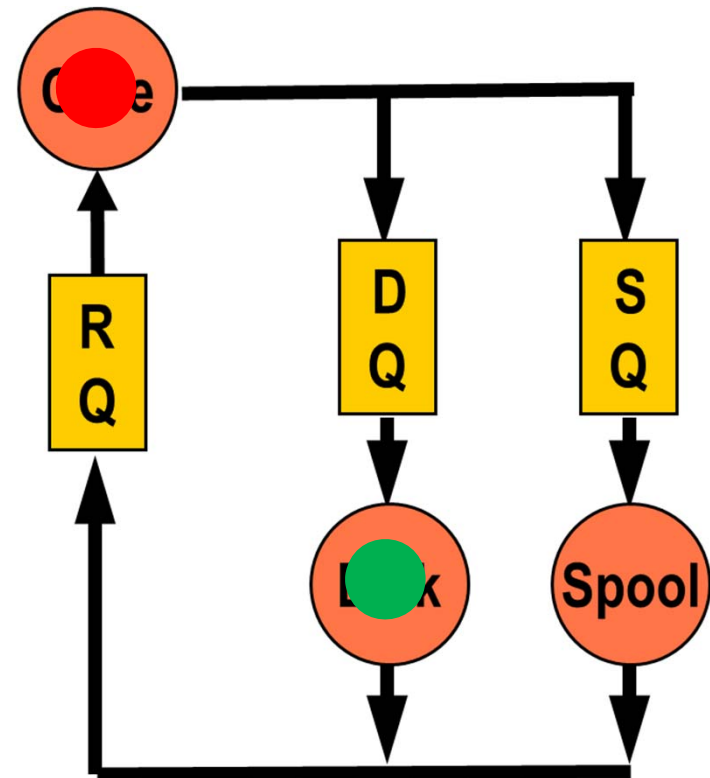 CORE   30
  JOB   5
  CORE   20
  DISK   8
  CORE   20

What's next?

- Must wait end of core step at $t = 0ms + 100ms = \textbf{100ms}$
- Can then perform the next step

Set $t = \textbf{100ms}$

Allocate disk to job 1 for 9ms at $t = 100$ms
Allocate core to job 5 for 20ms

- MPL   2
  JOB   1
  CORE   100
  DISK   9
  CORE   30
  JOB   5
  CORE   20
  DISK   8
  CORE   20

What's next

- Must wait for the first task to complete:
  - Disk I/O for job 1 ending at $t = 100ms + 9ms =$ **109ms**
  - CORE request from job 5 ending at $t = 100ms + 20ms = 120ms$

- Disk I/O for job 1 completes first
  - **Set t = 109ms**

Event list
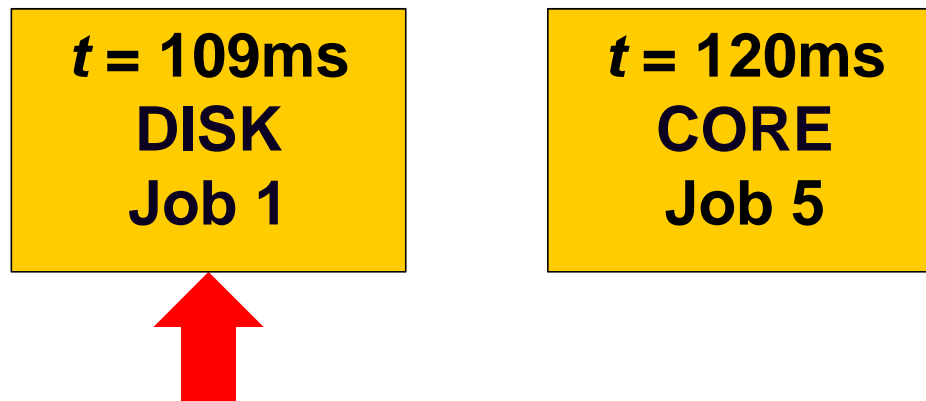
- Tells your program decide what step to take next
- Each record describes a completion event
  - Expected time
  - Type of event
  - Job ID

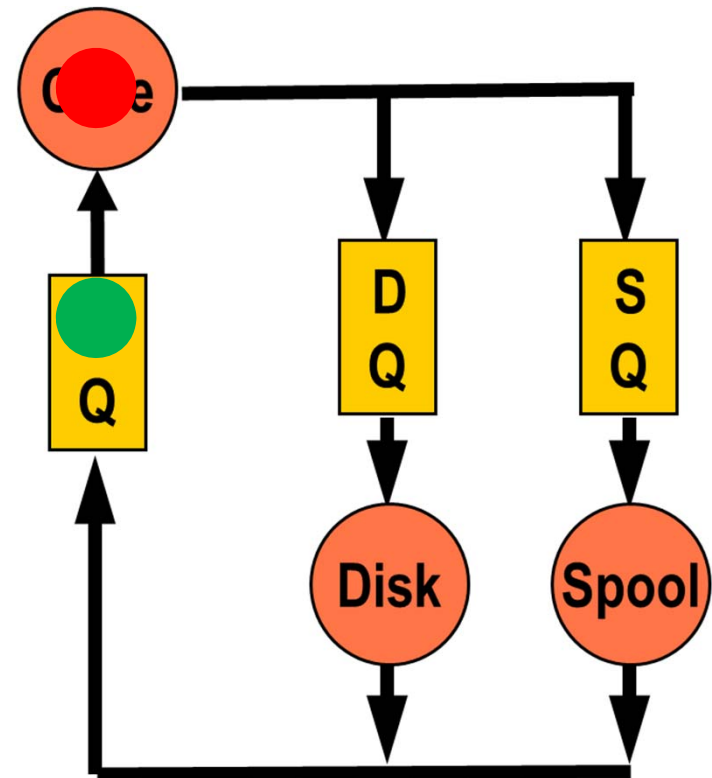|	
t = 109ms DISK Job 1	t = 120ms CORE Job 5

Using the event list

- When deciding which step to take,
  must always pick the one associated with the next event

t = 109ms DISK Job 1	t = 120ms CORE Job 5

Job 1 requests a core at $t = 109$ms

- MPL   2
  JOB   1
  CORE   100
  DISK   9
  CORE   30
  JOB   5
  CORE   20
  DISK   8
  CORE   20

The new event list
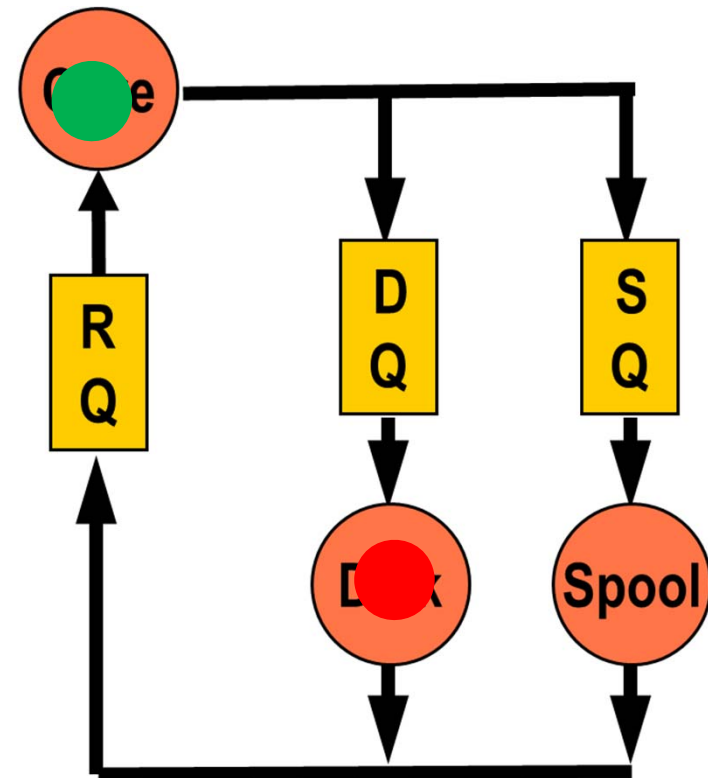
- Only one entry

- **Set t = 120ms**

t = 120ms
CORE
Job 5

Allocate core to job 1 for 30ms at $t = 120$ms
Allocate disk to job 5 for 8ms

- **MPL**   2
-   JOB   1
-   CORE   100
-   DISK   9
-   **CORE**   **30**
-   JOB   5
-   CORE   20
-   **DISK**   **8**
-   **CORE**   **20**

What's next

- Must wait for the first task to complete:
  - ☐ Disk I/O for job 5 ending at t = 120ms +8ms = **<u>128ms</u>**
  - ☐ CORE request from job 1ending at t = 120ms + 30ms = 150ms

- Disk I/O for job 5 completes first
  - ☐ **Set t = 128ms**

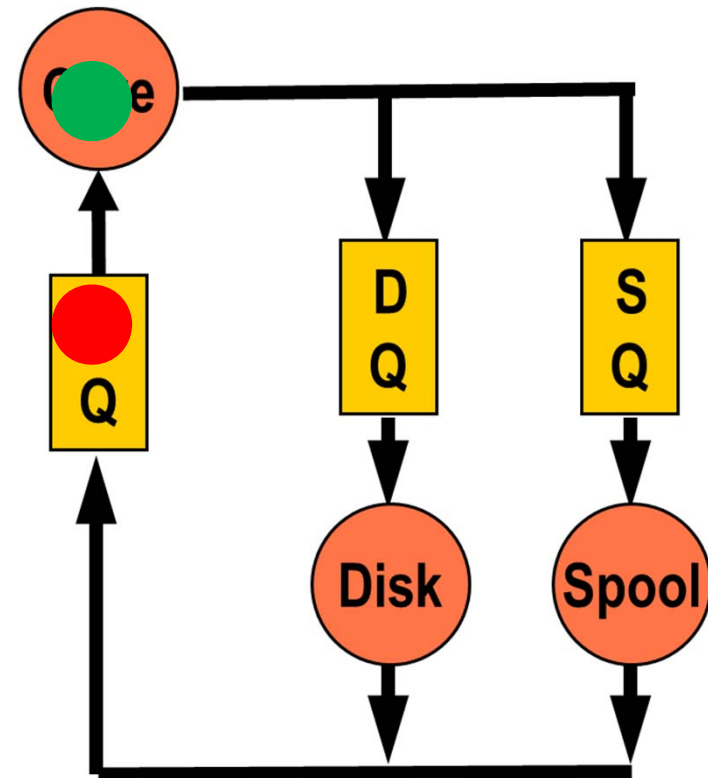# The new event list

- When deciding which step to take,
  must always pick the one associated with the next event

t = 128ms

DISK

Job 5
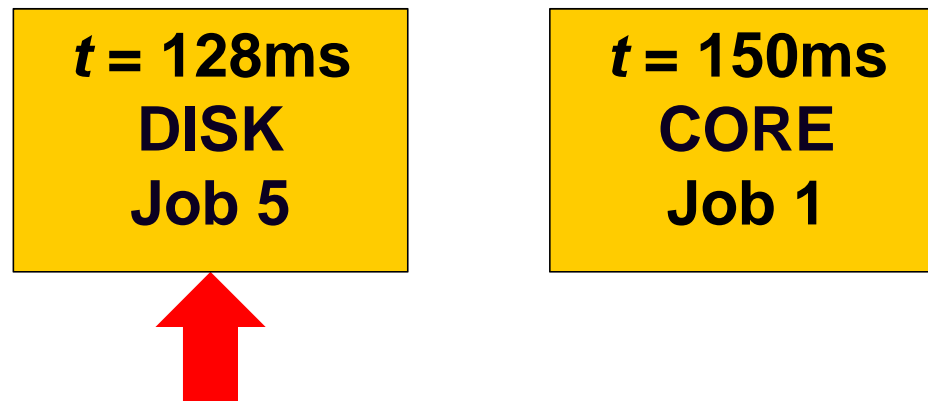
t = 150ms

CORE

Job 1

Job 5 requests CORE at $t = 128$ms

- MPL   2
  JOB   1
  CORE   100
  DISK   9
  CORE   30
  JOB   5
  CORE   20
  DISK   8
  CORE   20

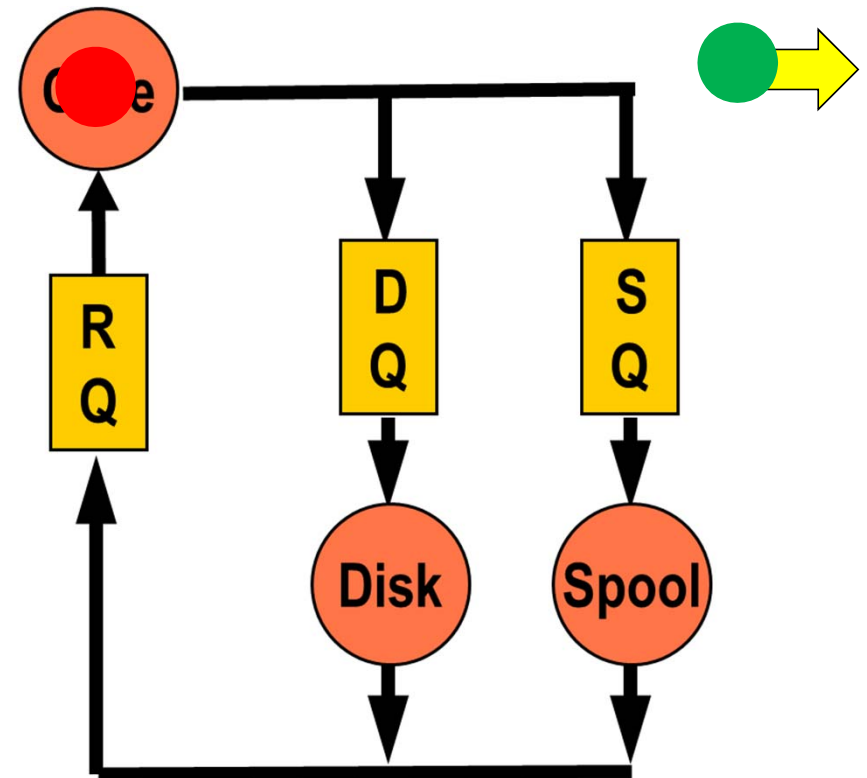# The new event list

t = 150ms
CORE
Job 1

Job 1 terminates at $t = 150$ms
Allocate core to job 5 for 20ms

- MPL   2
  JOB   1
  CORE   100
  DISK   9
  CORE   30
  JOB   5
  CORE   20
  DISK   8
  CORE   20

Output

- Job 1 terminates at time 150ms
  Job Table:
  Job 5 is RUNNING

Output

- Job 1 terminates at time 150ms
  Job Table:
  Job 5 is RUNNING
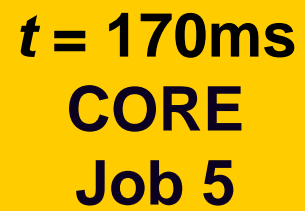
What's next

- Must wait for the completion of core request for job 5 ending at $t = $ 150ms +20ms = **<u>170ms</u>**

- **Set t = 170ms**

The new event list

_t_ = 170ms
CORE
Job 5

Job 5 terminates at $t = 170$ms

- MPL   2
  JOB   1
  CORE   100
  DISK   9
  CORE   30
  JOB   5
  CORE   20
  DISK   8
  CORE   20
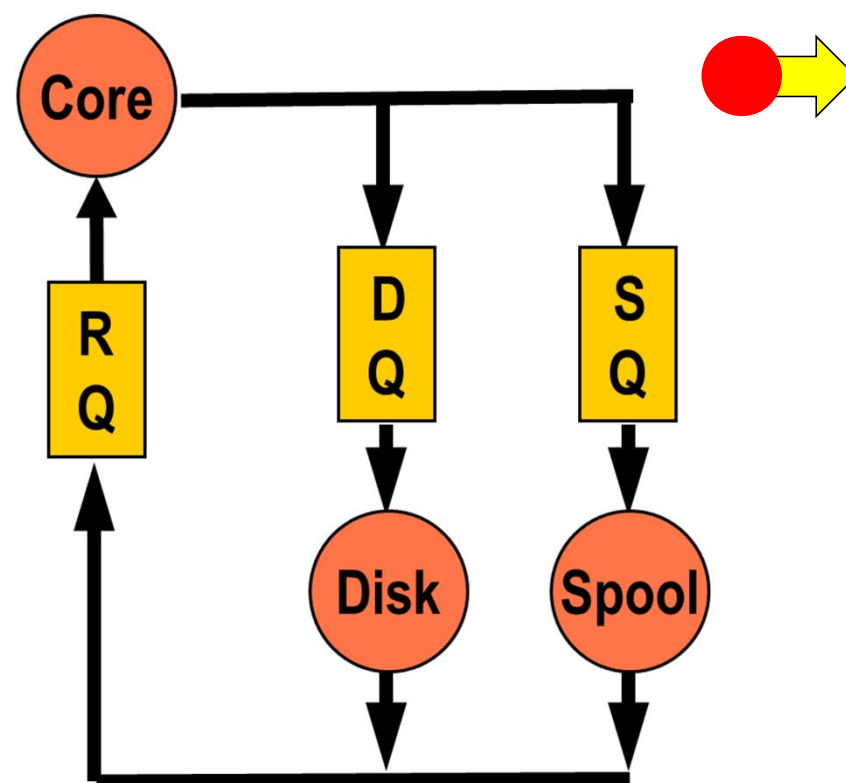
Output

- Job 5 terminates at time 170ms
  Job Table:
  There are no active jobs

- SUMMARY:
  Total elapsed time: 170ms
  Number of jobs that have completed: 2
  Number of disk accesses: 2
  Core utilization: 1.00

Computing core utilization

- $$\frac{\text{Sum of all CORE times}}{\text{Simulated elapsed time}}$$

- (100 + 30 + 20 + 20)/170 = 1.00
  - *The CPU was always busy*

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 job to wait for device
      - Ready queue, disk queue, spooler queue

The simulated time

- Imaginary clock keeping track of simulated time
  - Never ticks
  - Only updated each time we process a new event

ENGINEERING THE SIMULATION

Simulating time

- **Absolutely nothing happens to our model between two successive "events"**
- **Events are**
    - ☐ Completion of a computing step
    - ☐ Completion of a disk access
    - ☐ Completion of a spooler request
- We associate an **event routine** with each event

Organizing our program (I)

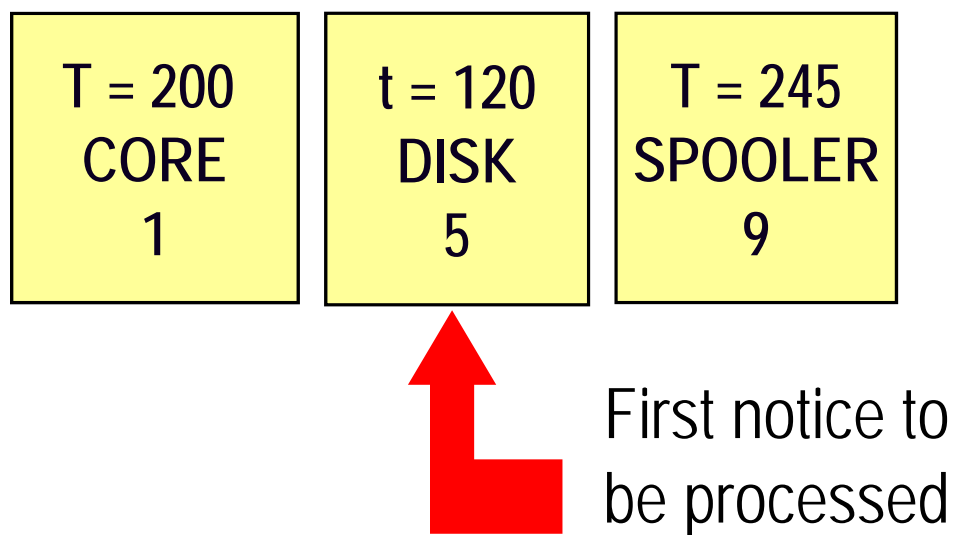- Most steps of simulation involve scheduling future completion events
- Associate with each completion event an event notice
  - *Time of event*
  - *Device (core, disk, spooler)*
  - *Job ID*

Organizing our program (II)

- Process all event notices in chronological order

T = 200 CORE 1	t = 120 DISK 5	T = 245 SPOOLER 9

First notice to be processed

Organizing our program (III)

- Overall organization of main program

```
read in input file
start first MPL jobs
while (event list is not empty) {
        process next event in list
} // while
print simulation results
```

Organizing our program (IV)

- Processing next event in list

```
pop event from list
clock = event.time
if (event.type is core) {
    core(event.time, event.jobID)
else if (event.type is disk) {
    disk(event.time, event.jobID)
```
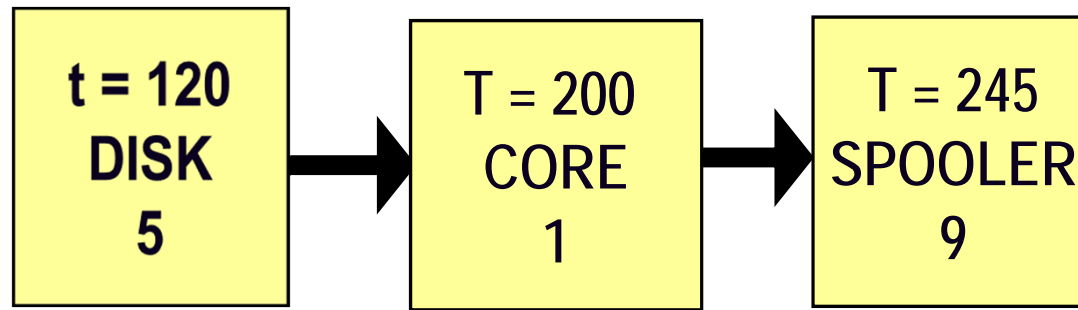
Organizing our event list (I)

- As a priority queue

- Associating a completion time
  - With each core request
  - With each disk request
  - With each spooler request

Organizing our event list

- Process all event notices in time order

First notice to be processed
is at the head of the list

Core request routine

```
core_request(how_long, jobID){
    if (core == FREE) {
        core = BUSY;
        schedule CORE completion at time
        current_time + how_long for job jobID;
    } else {
        queue jobID in readyQueue
    }  // if
} // core_request
```

Core completion routine

```
core_release (jobID){
    if (readyQueue is not empty) {
        pop first core request in readyQueue
        schedule its completion at
        current_time + how_long
    } else {
        core = FREE;
    } //if
    process next job request for job jobID
} // core_release
```

Disk request routine

```
disk_request(how_long, jobID){
    if (how_long == 0) {
        perform next job request
    } else if (disk == FREE) {
        disk = BUSY;
        schedule DISK completion event at time
        current_time + how_long for job jobID
    } else {
        queue job jobID in diskQueue;
    }  // if
} // disk_request
```

Disk completion routine

```
disk_release (jobID){
    if (disk queue is not empty) {
        pop first request in disk queue
        schedule its completion at
        current_time + how_long;
    } else {
        disk = FREE;
    } // if
    process next job request for job jobID
} // disk_release
```

Spooler request routine

```
spooler_request(how_long, jobID){
     if (spooler == FREE) {
          disk = BUSY;
          schedule SPOOLER completion event at time
          current_time + how_long for job jobID
     } else {
          queue job jobID in spoolerQueue;
     }  // if
} // spooler_request
```

Spooler completion routine

```
spooler_release (jobID){
    if (spooler queue is not empty) {
        pop first request in spooler queue
        schedule its completion at
        current_time + how_long;
    } else {
        spooler = FREE;
    } // if
    process next job request for job jobID
} // disk_release
```

Overview (I)

- **Input module**
  - Read in all input data
    - Store them in a jobList
  - Start first MPL jobs
- **Main loop**
  - Pops next event from event list
    - CORE completion
    - DISK completion
    - SPOOLER completion

Overview (II)

- ***Starting a job***
  - Will always be a Core request

- ***Handling job termination***

```
if (jobList is empty) {
      break;  //  then print summary
  } else  {
      pop next job from jobList
  } // if
```

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
    - DISK or SPOOLER

Overview (IV)

- ***Disk request***
  - ☐ If disk is free
    - Schedules an DISK completion event

- ***Disk completion event***
  - ☐ May schedule a DISK completion event
  - ☐ Starts next request
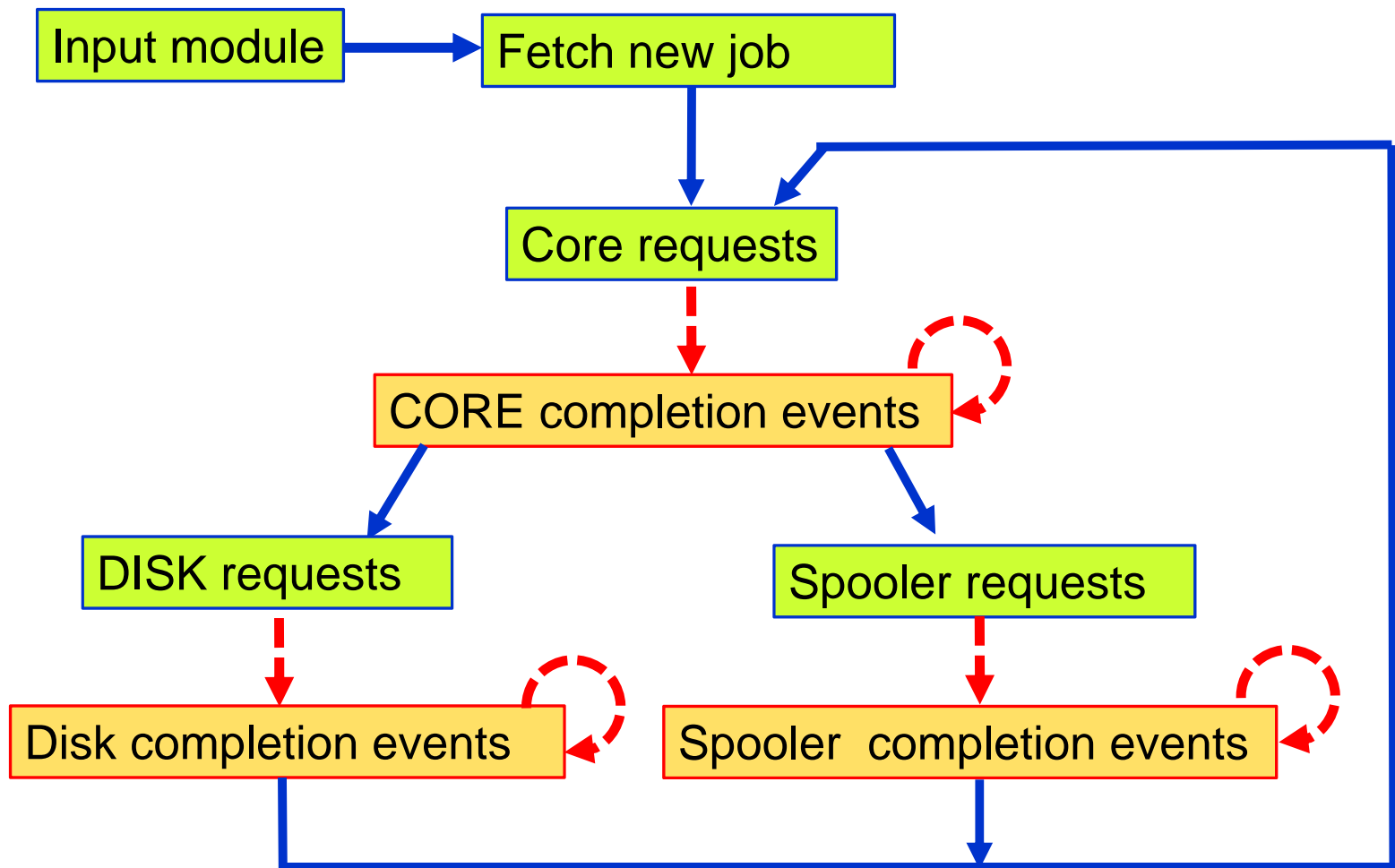   - Always a CORE request

Overview (V)

- ***Spooler request***
  - ☐ Schedules a SPOOLER completion event

- ***Spooler completion event***
  - ☐ May schedule a SPOOLER completion event
  - ☐ Starts next request
    - Always a CORE request

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 events

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** among the times
  - □ The CPU will be done with the current job
  - □ The disk will complete a disk I/O
  - □ The spooler will be done with the current job

AN IMPLEMENTATION

- My main data structures would include:

  - ☐ An input table
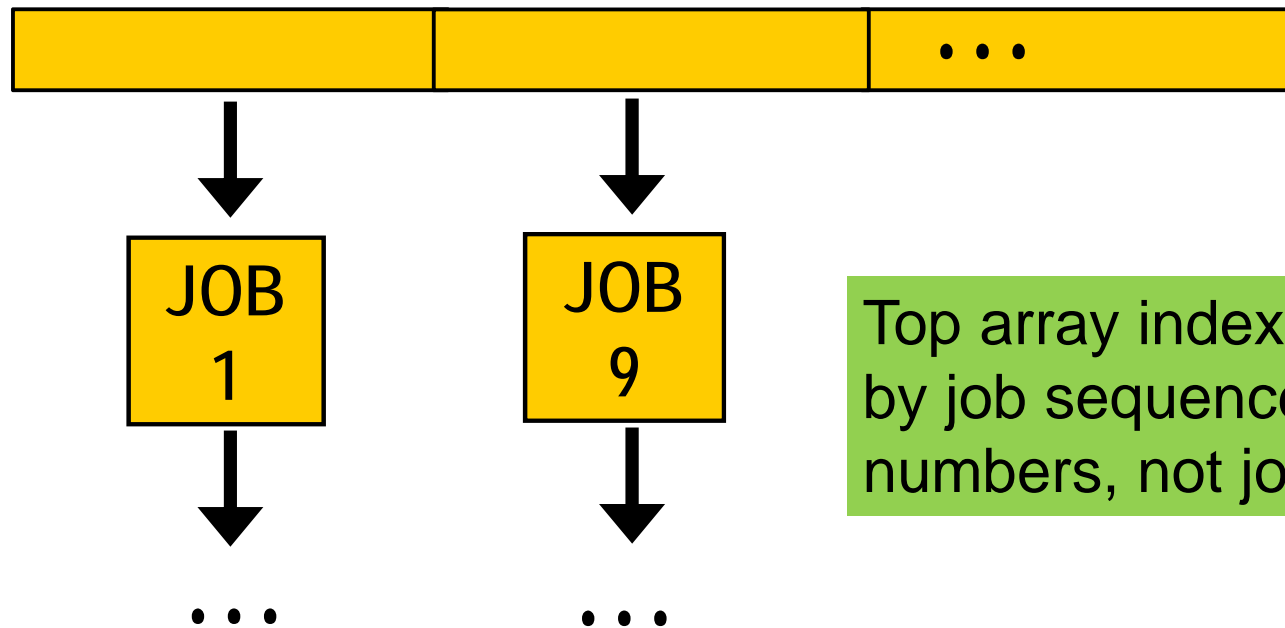
  - ☐ A job/process table

  - ☐ A device table

The input table

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

Operation	Parameter
MPL	5
JOB	3
CORE	20
DISK	0
CORE	20
JOB	42
CORE	4
…	…

A more elegant input table

Top array indexed by job sequence numbers, not job ID

The job table/process table (I)

Job ID	First Line	Last Line	Current Line	State
3	1	4	varies	varies
4	5	…	…	
	…	…	…	

The job table/process table (II)

- One line per job
  - □ *Line index is job sequence number*!
- First column has start time of job
- 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 total
CPU	P0	15
disk	-	--

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 disk
- 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 reading 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
    - `while(scanf(…)) { … }`

- If you use `cin`
  - `cin` returns `0` once it reaches the end of data
    - `while (cin >> keyword >> argument)  { … }`