



The first spring 2024 assignment explained

Jehan-François Pâris
jfparis@uh.edu



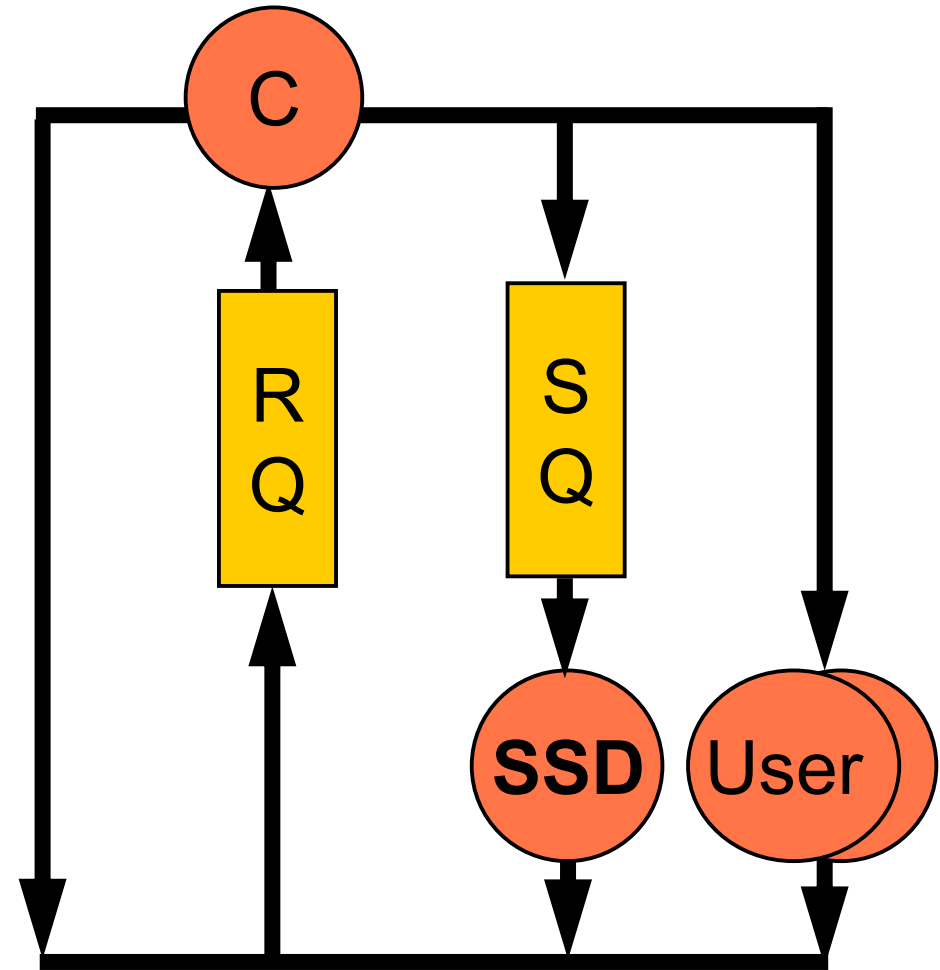
Assignment objectives

- We will simulate how processes circulate in a time-sharing system when
 - They wait for and finally get a core
 - They perform read and write operations on the SSD
 - They interact with the users

The model

A system with

- A single processor
- One SSD
- One user window for each process
- Plenty of RAM

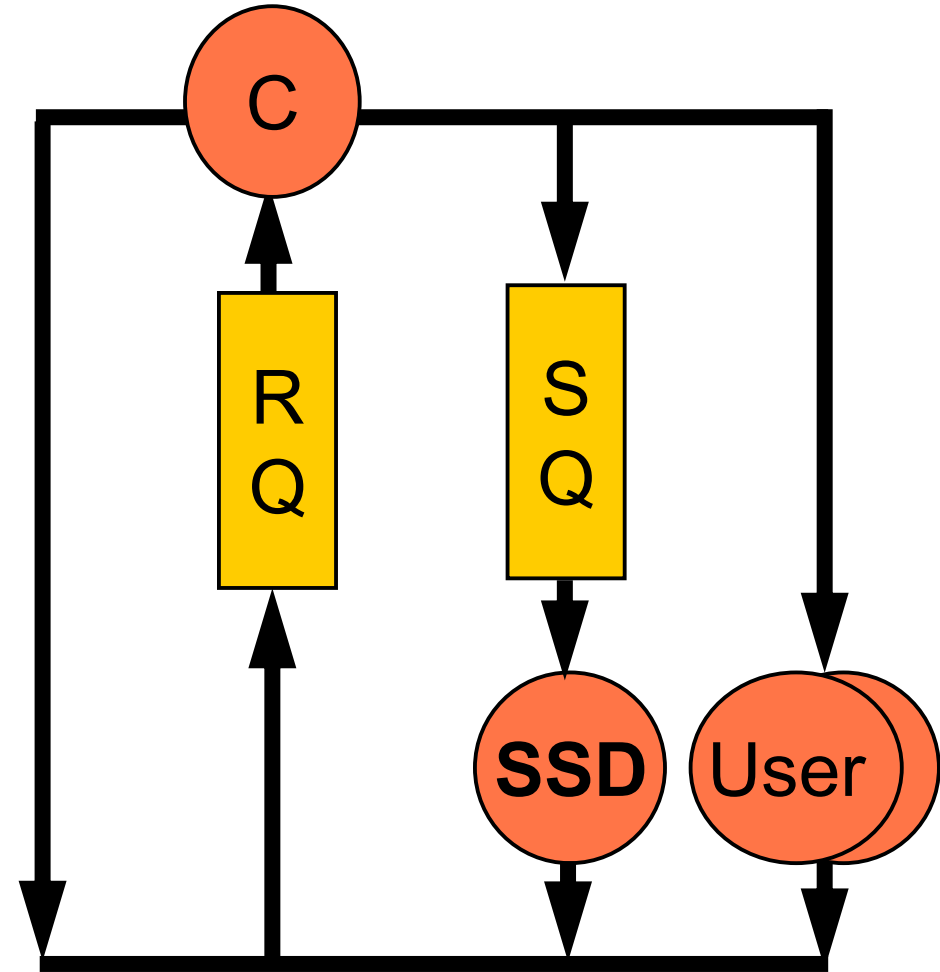


The queues (I)

Processes have to wait for

- The core
- The SSD

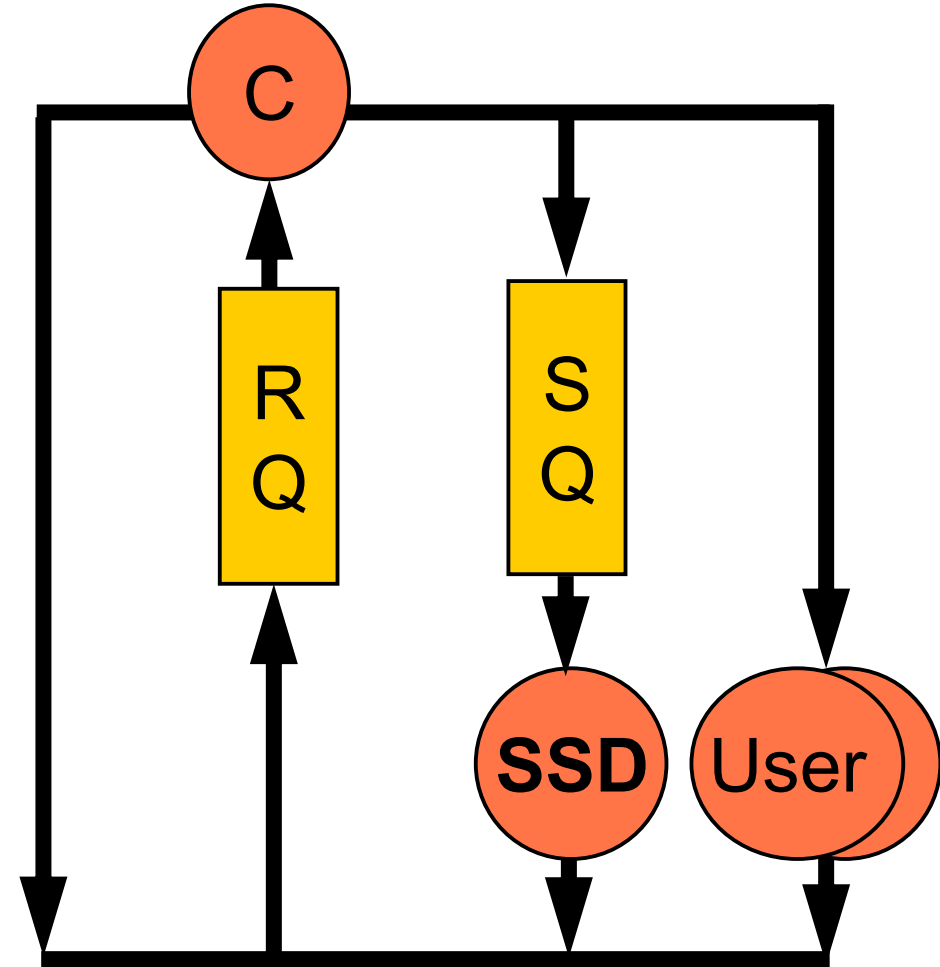
***No queues, and no delays,
for all user interactions***



The queues

We will have

- One *ready queue* for the CPU
- One SSD queue





Managing the queues

- Will be First-Come First-served (FCFS)
 - All queues will be FIFO queues
- Totally unrealistic for the ready queue
 - Actual systems use much more sophisticated scheduling policies
 - Too hard to implement in an assignment

Process states

A job can be

☐ **Running**

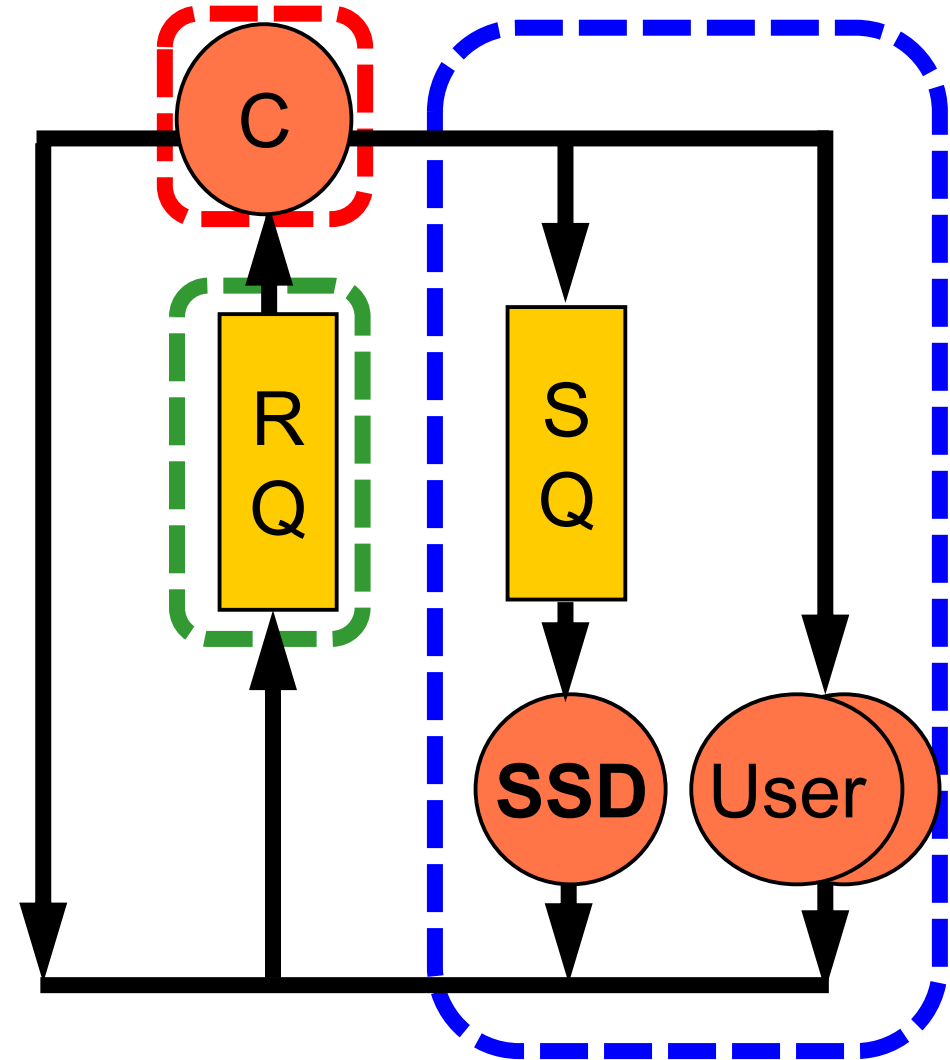
❖ It occupies the single core


☐ **Ready**

❖ It waits for a core

☐ **Blocked**

❖ It waits for the completion of an I/O request

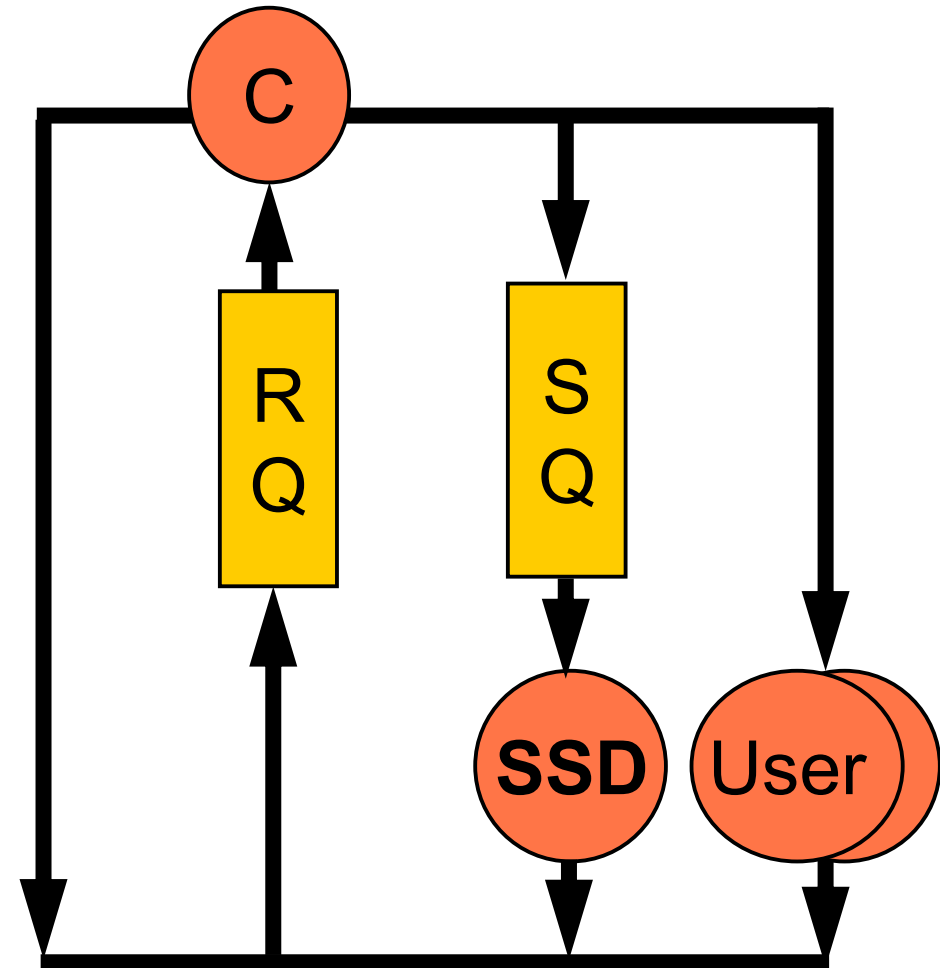




A very simple example

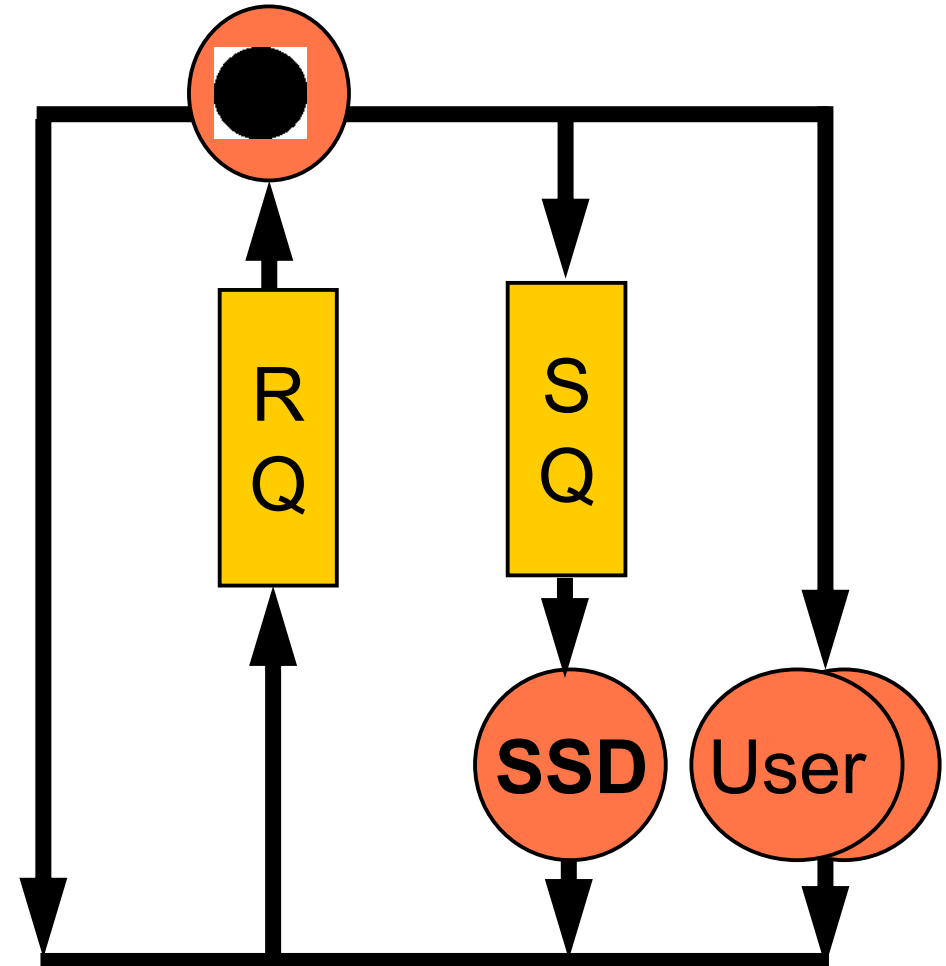
Process P_0 arrives at $t = 0\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



P_0 requests and gets the CPU until
 $t = 0 + 200 = 200\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



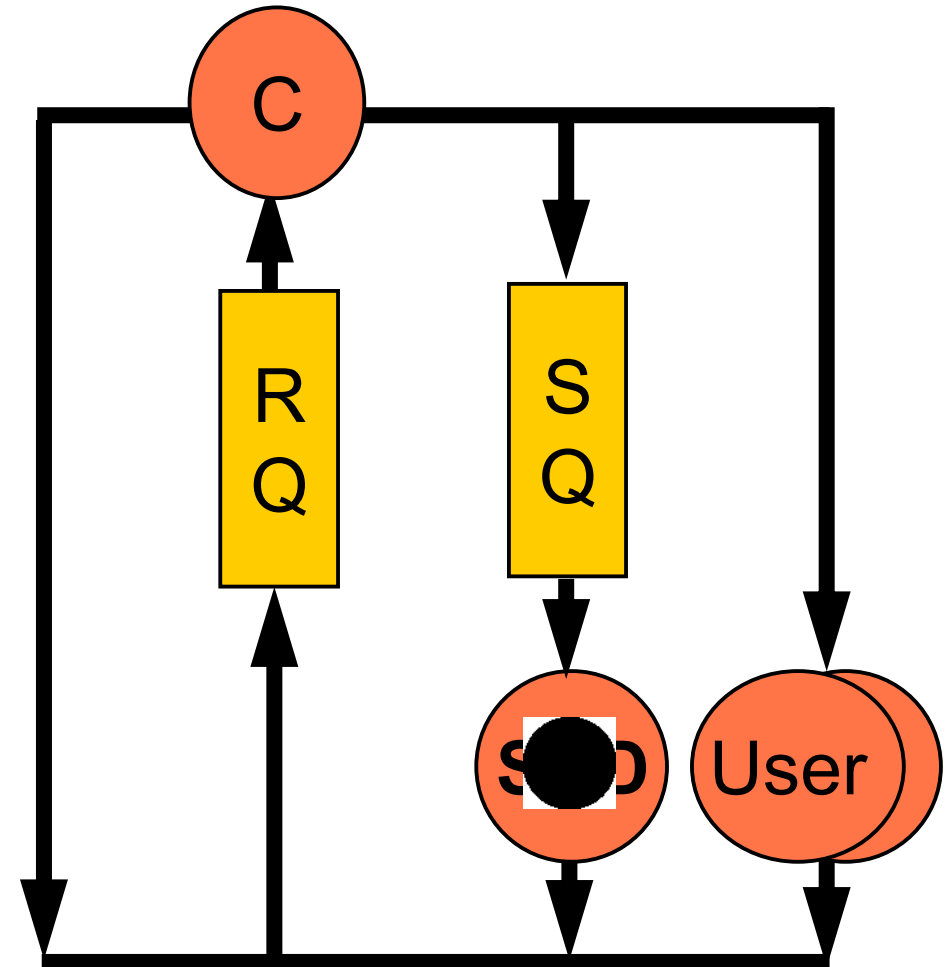


Our actions

- Set our virtual—simulation—clock time to **0ms**
- Allocate the core to P_0
- Note request will complete at **$t = 0 + 200 = 200\text{ms}$**

A $t = 200\text{ms}$, P_0 releases the CPU and gets the SSD until $t = 200 + 0.1 = 200.1\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30





Performing the read

- Start with an empty buffer
- P_0 requests the next 256 bytes of its file
- Kernel fetches **BSIZE** bytes = 4,096 bytes
- $4,096 - 256 = 3,840$ bytes remain in the buffer

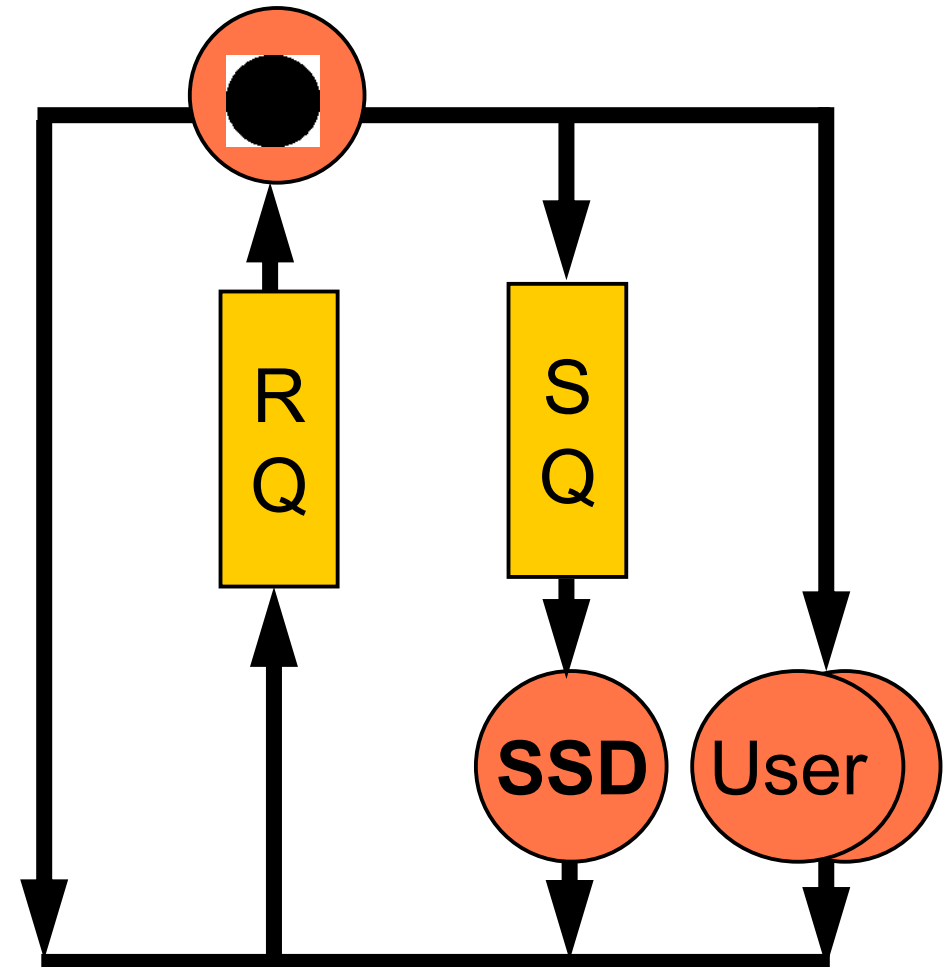


Our actions

- Set our virtual—simulation—clock time to **200ms**
- Release the core.
- Allocate the SSD to P_0
- Note request will complete at **$t = 200 + 0.1 = 200.1\text{ms}$**

A $t = 200.1\text{ms}$, P_0 releases the SSD and gets the core until $t = 200.1 + 30 = 230.1\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30

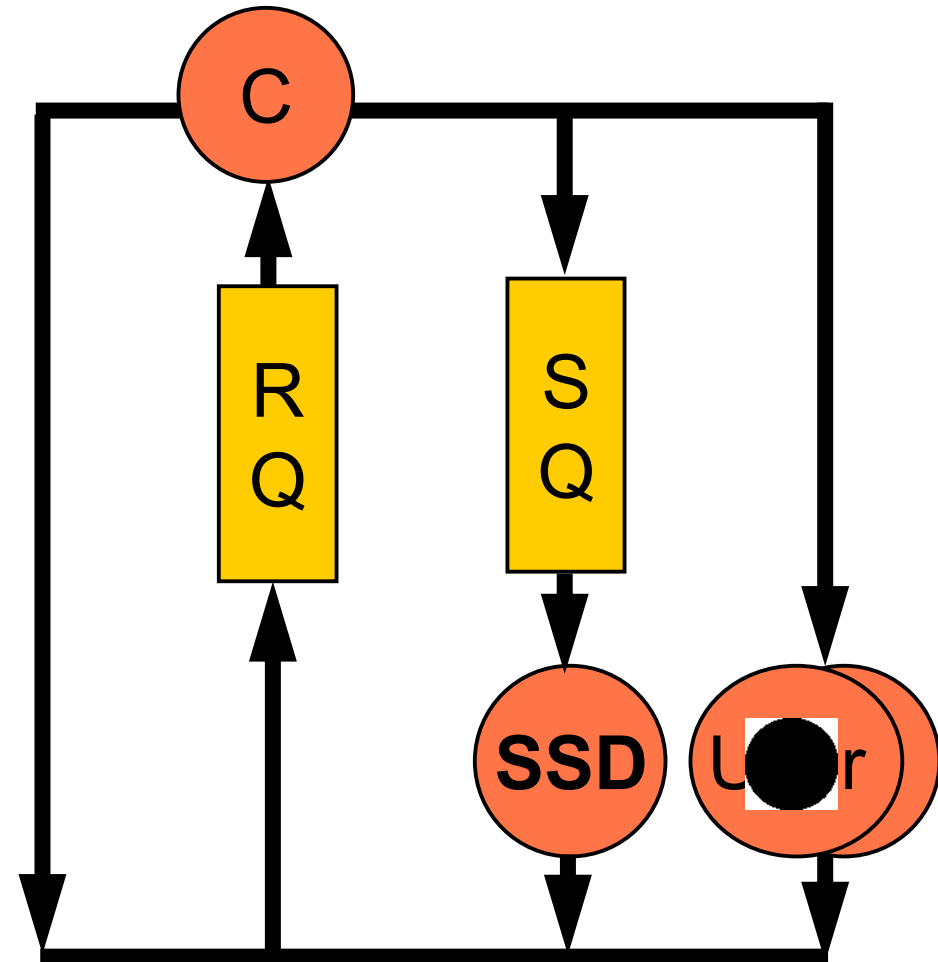


Our actions

- Set our virtual—simulation—clock time to **200.1ms**
- Release the SSD.
- Allocate the CPU to P_0
- Note request will complete at **$t = 200.1 + 30 = 200.1\text{ms}$**

A $t = 230.1\text{ms}$, P_0 releases the CPU and display info until $t = 230.1 + 100 = 330.1\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



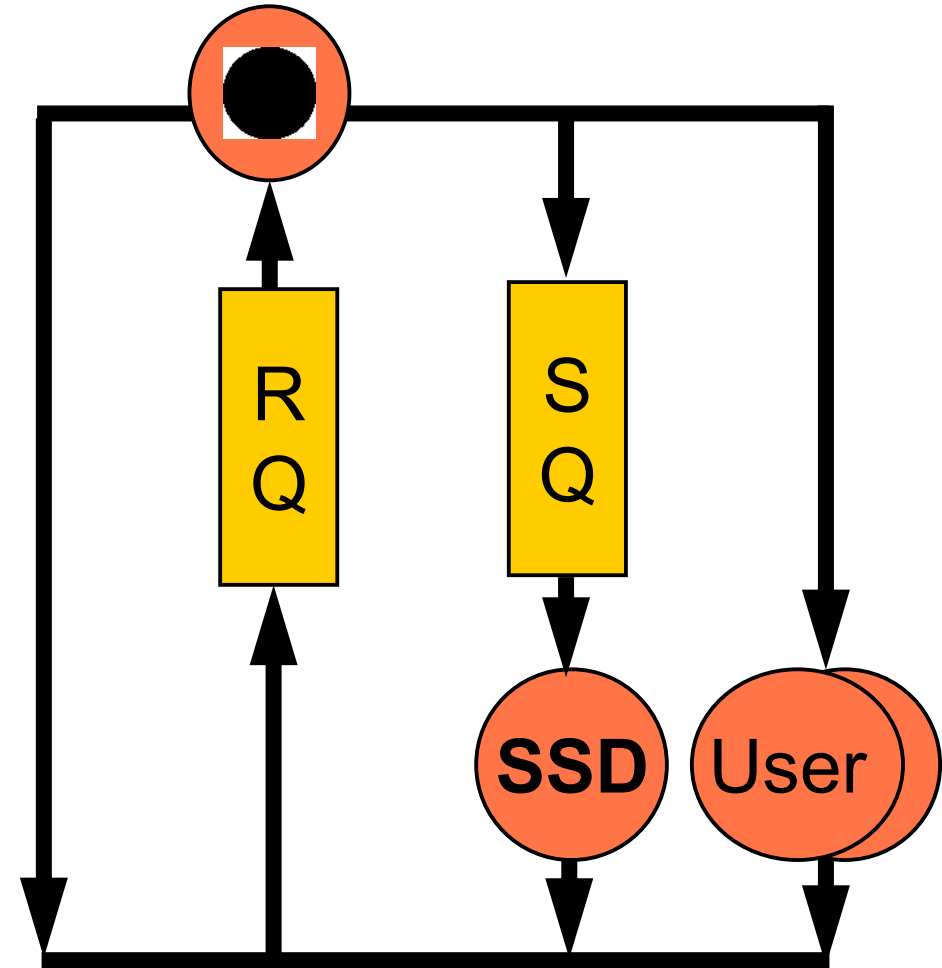


Our actions

- Set our virtual—simulation—clock time to **230.1ms**
- Release the CPU.
- Note request will complete at **$t = 230.1 + 100 = 330.1\text{ms}$**

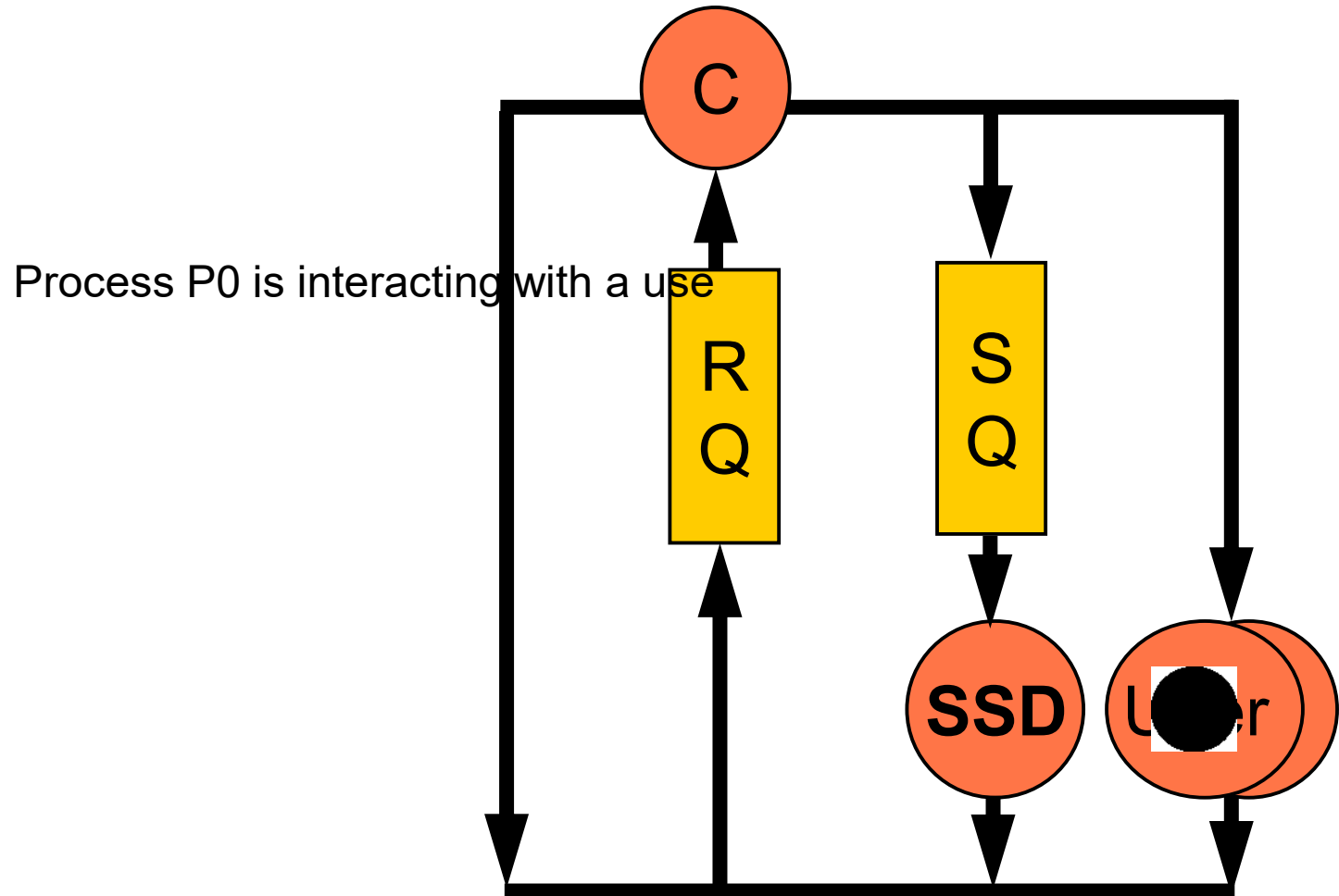
A $t = 330.1\text{ms}$, P_0 gets the CPU until
 $t = 330.1 + 10 = 340.1\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



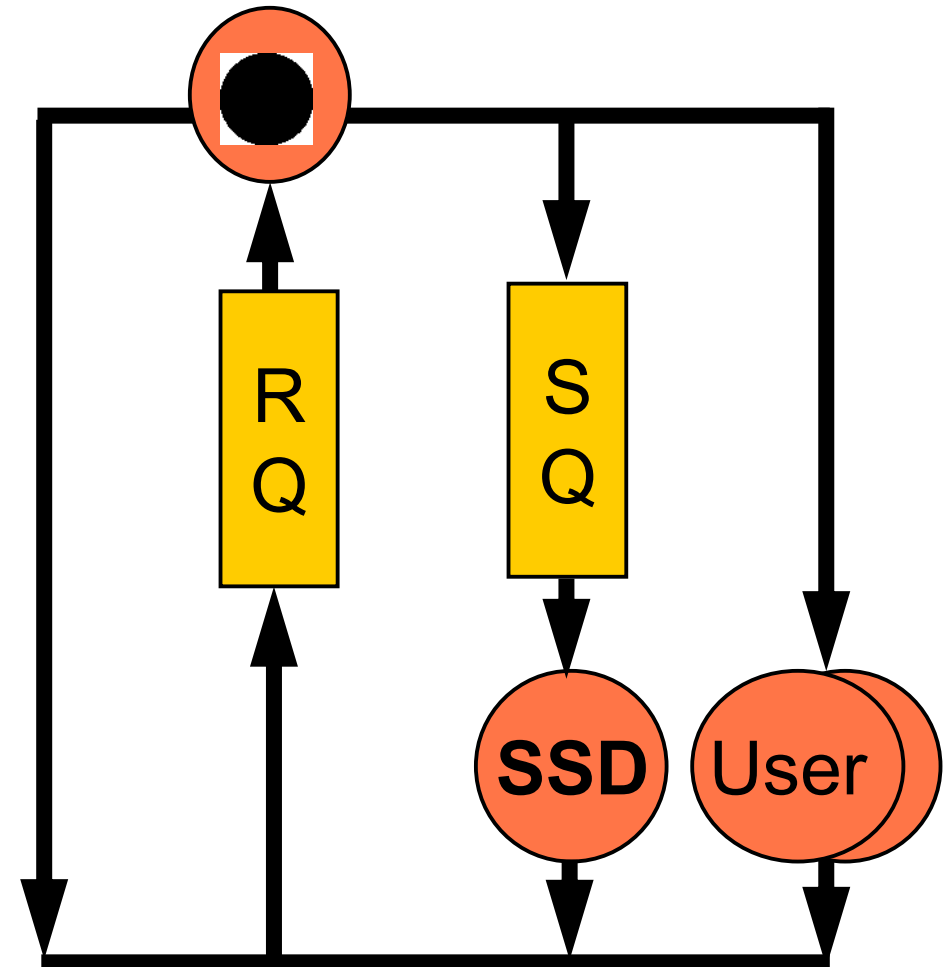
A $t = 340.1\text{ms}$, P_0 releases the CPU and gets user input until $t = 340.1 + 900 = 1240.1\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



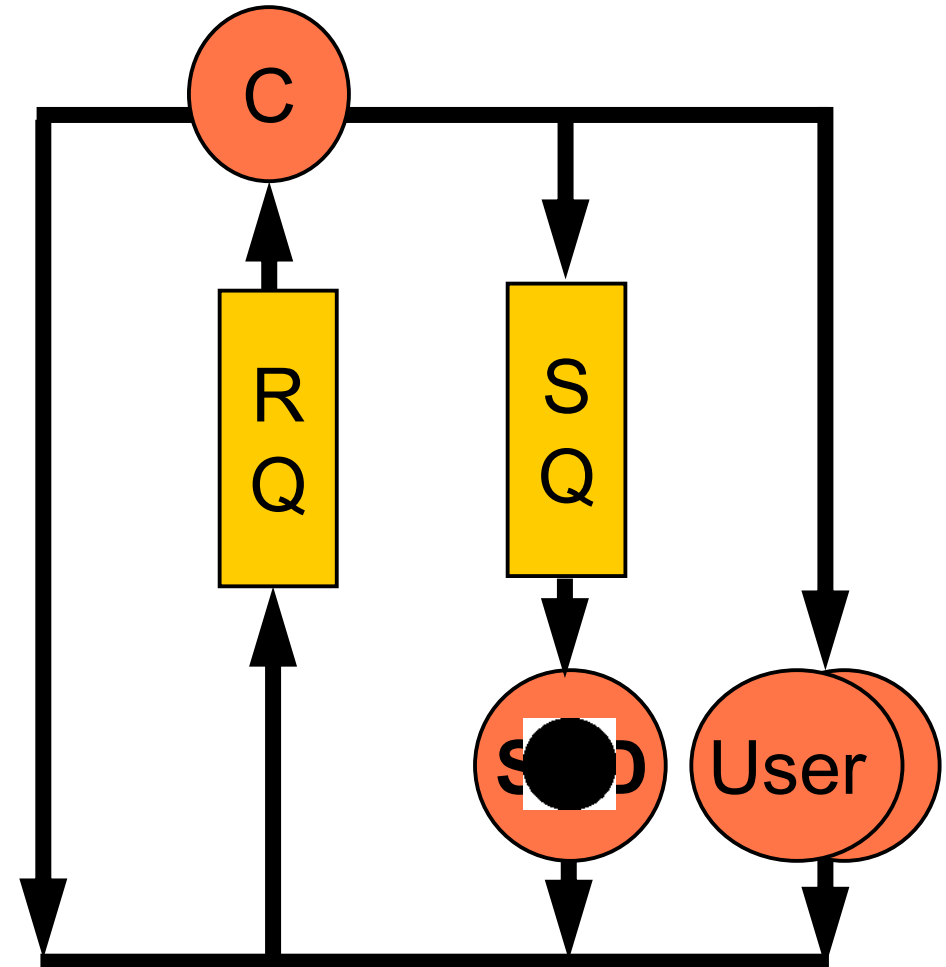
A $t = 1240.1\text{ms}$, P_0 gets the CPU
until $t = 1240.1 + 10 = 1250.1\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



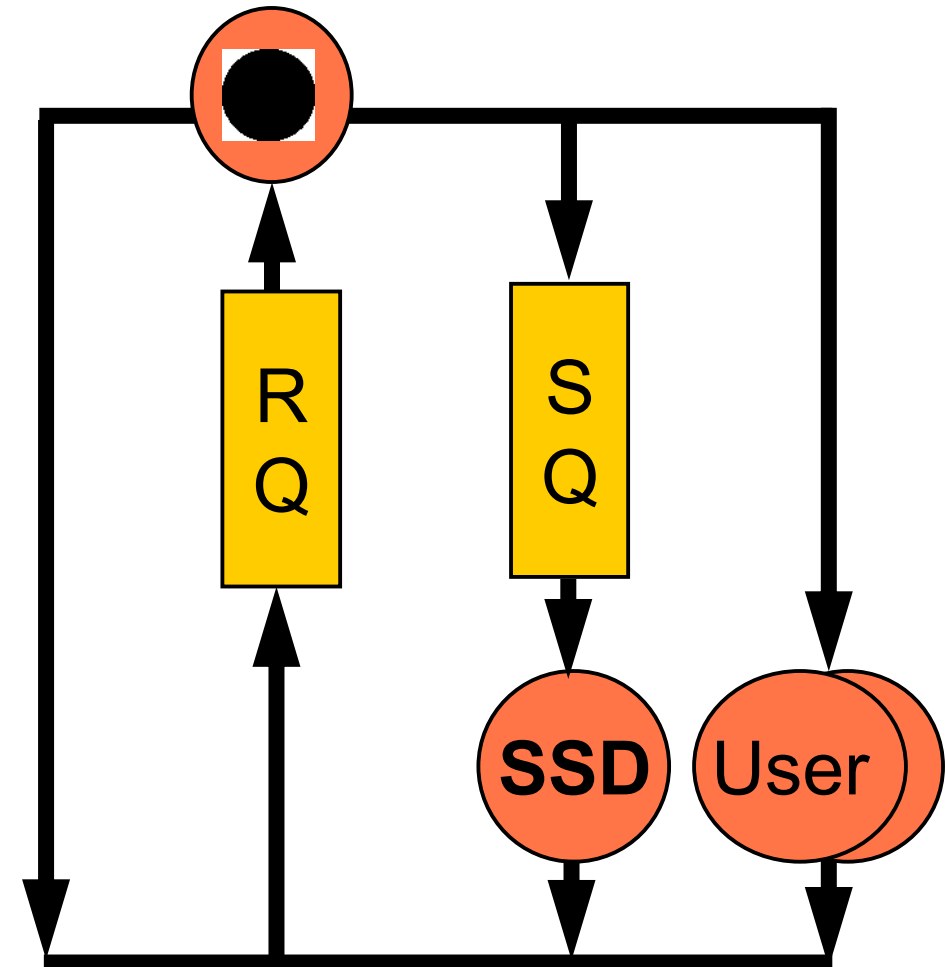
A $t = 1250.1\text{ms}$, P_0 releases the CPU and gets the SSD until $t = 1250.1 + 0.1 = 1250.2\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



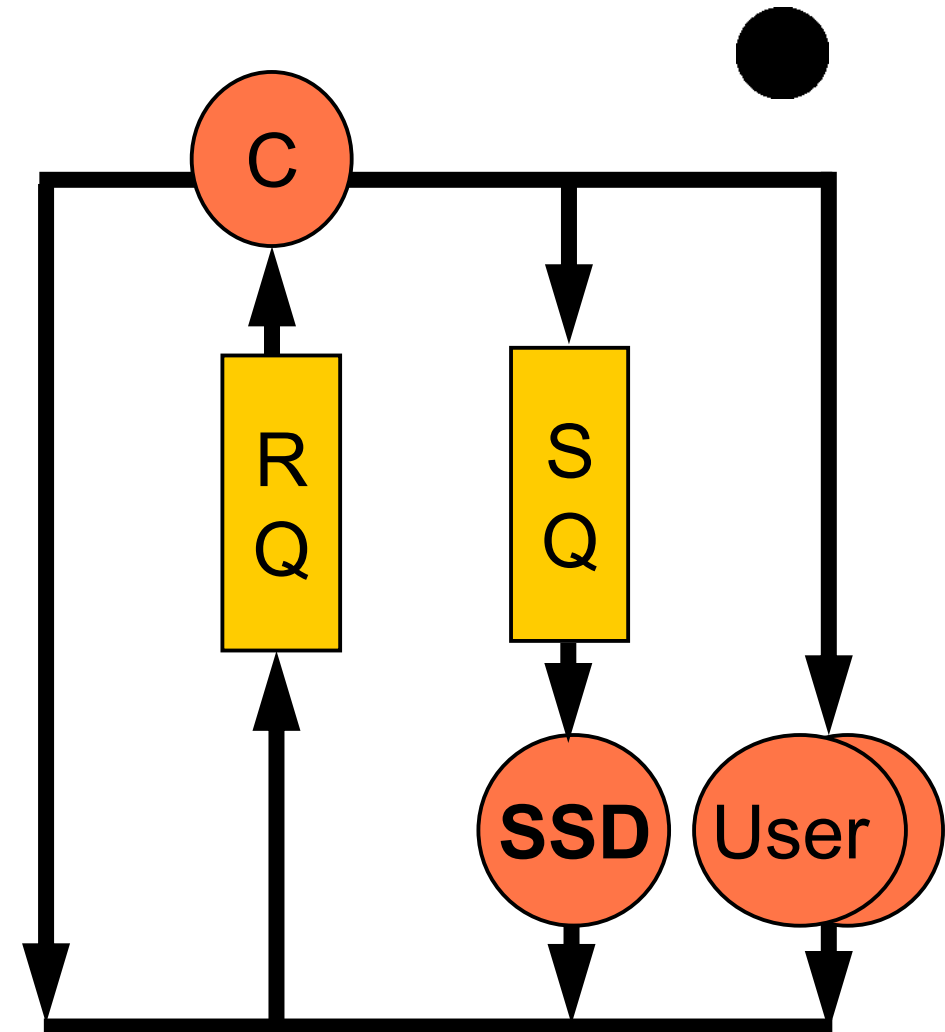
A $t = 1250.2\text{ms}$, P_0 releases the SSD and gets the core until $t = 1250.2 + 30 = 1280.2\text{ms}$

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30



$A t = 1280.2\text{ms}$, P_0 terminates

BSIZE 4096
START 0
CORE 200
READ 256
CORE 30
DISPLAY 100
CORE 10
INPUT 900
CORE 10
WRITE 256
CORE 30





Required output

Process 0 terminates at $t = 1280.2\text{ms}$.

It performed 1 physical read(s), 0 logical read(s), and 1 physical write(s).

Process states:

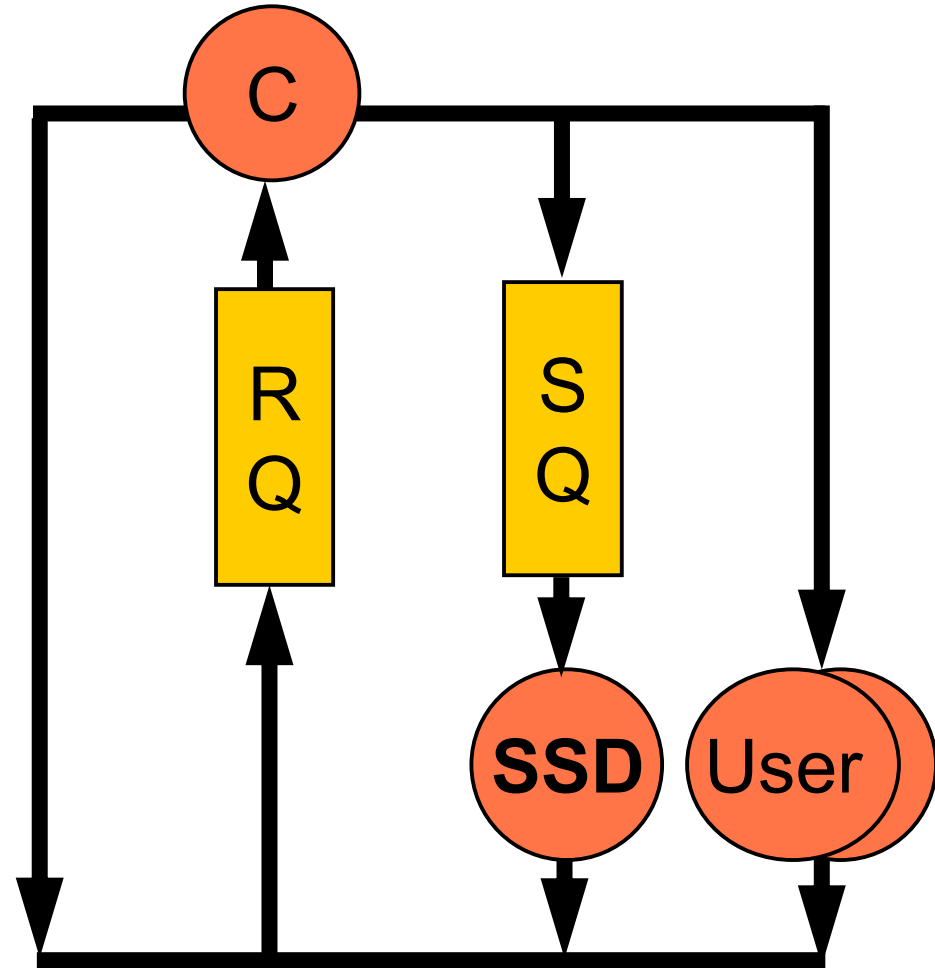
0 TERMINATED



Adding buffered reads

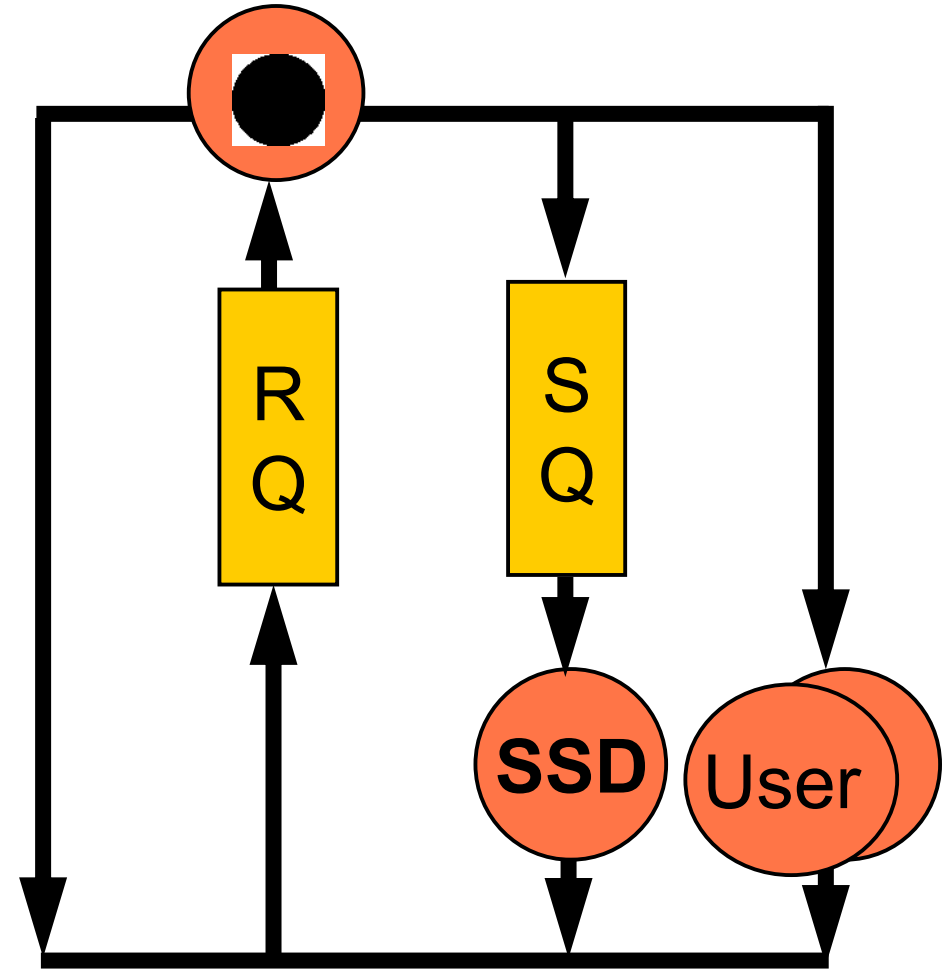
Process P_0 arrives at $t = 40\text{ms}$

BSIZE 4096
START 40
CORE 200
READ 2048
CORE 30
READ 1024
CORE 10
READ 2048
CORE 40
...



P_0 requests and gets the CPU until
 $t = 40 + 200 = 240\text{ms}$

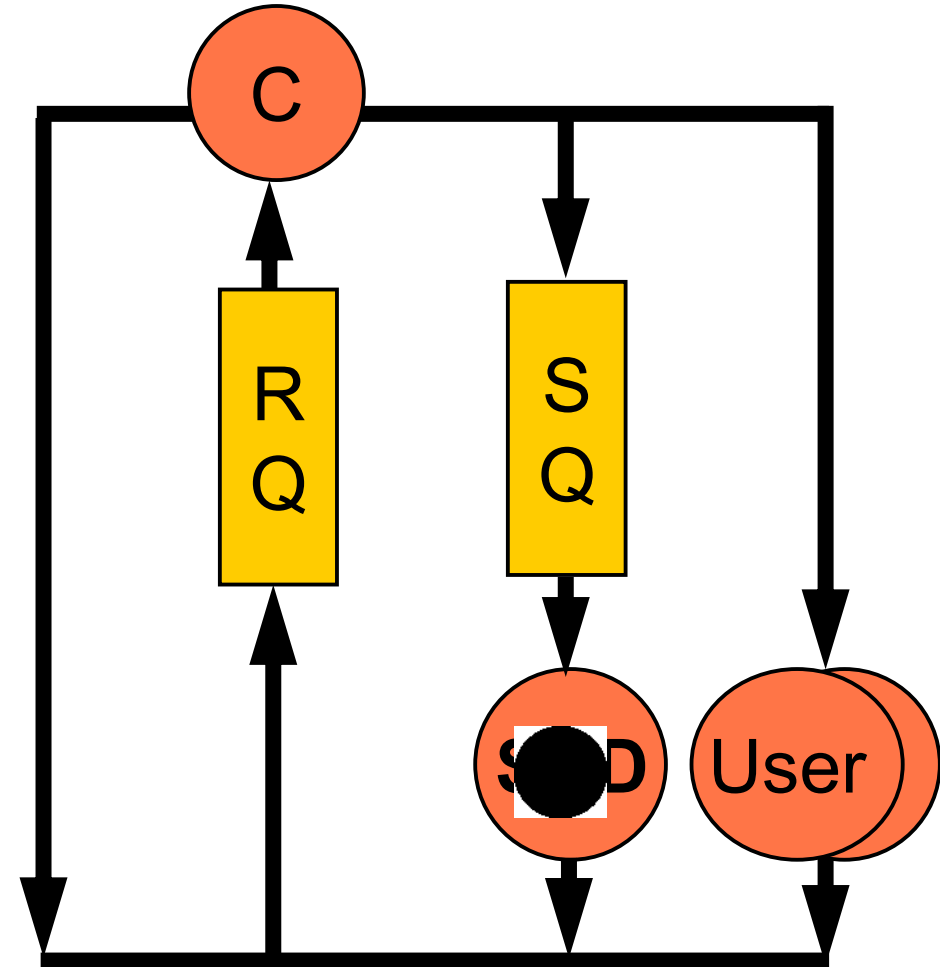
BSIZE 4096
START 40
CORE 200
READ 2048
CORE 30
READ 1024
CORE 10
READ 2048
CORE 40
...



P_0 releases the CPU and performs an SSD access until $t = 240 + 0.1 = 240.1\text{ms}$

BSIZE 4096
START 40
CORE 200
READ 2048

CORE 30
READ 1024
CORE 10
READ 2048
CORE 40
...



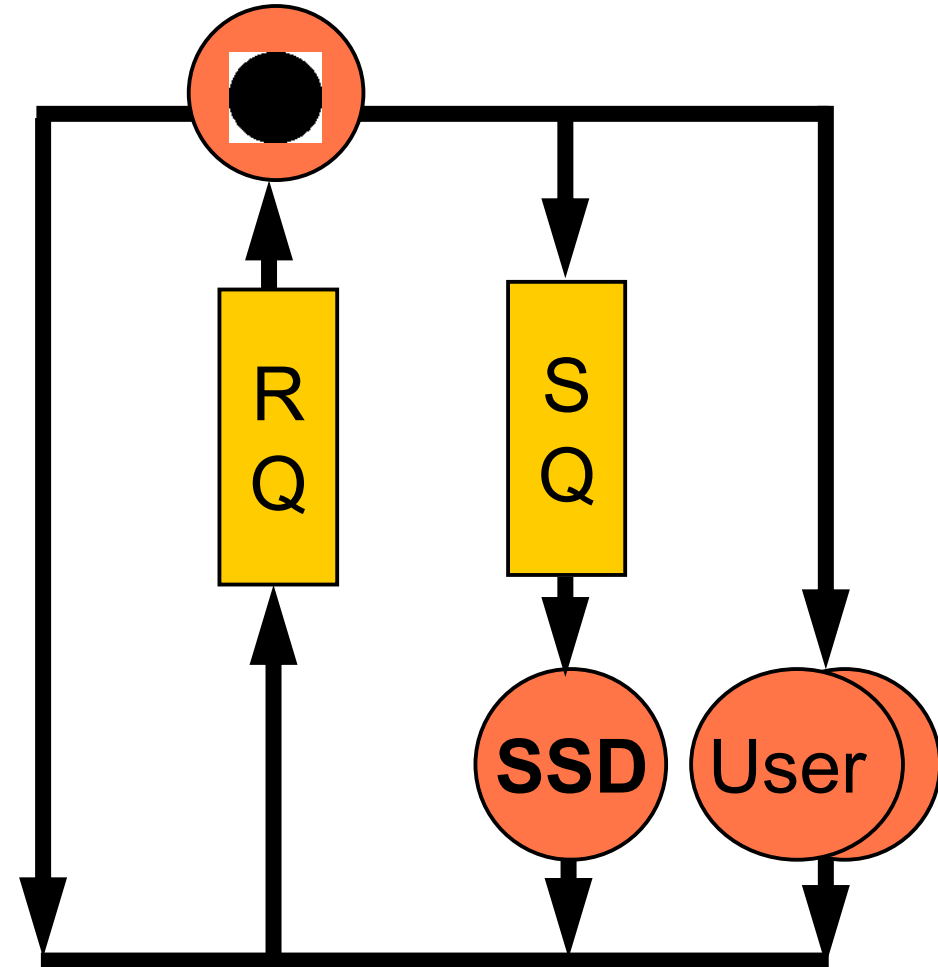


The SSD access

- Process requests 2,048 bytes from its file
- Kernel reads 4,096 bytes
 - 2,048 unread bytes will remain in the I/O buffer.

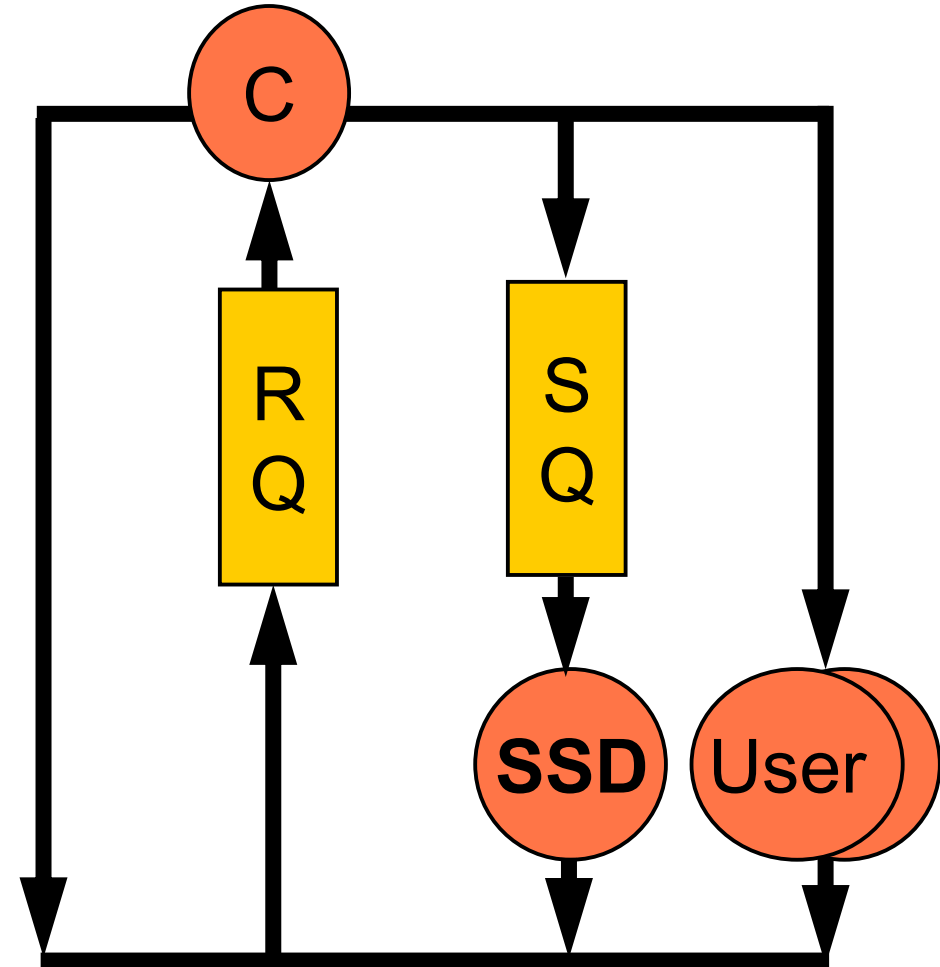
P_0 requests and gets the CPU until
 $t = 240.1 + 30 = 270.1\text{ms}$

BSIZE 4096
START 40
CORE 200
READ 2048
CORE 30
READ 1024
CORE 10
READ 2048
CORE 40
...



P_0 reads 1,024 bytes that are already in memory at time $t = 270.1\text{ms}$

BSIZE 4096
START 40
CORE 200
READ 2048
CORE 30
READ 1024
CORE 10
READ 2048
CORE 40
...



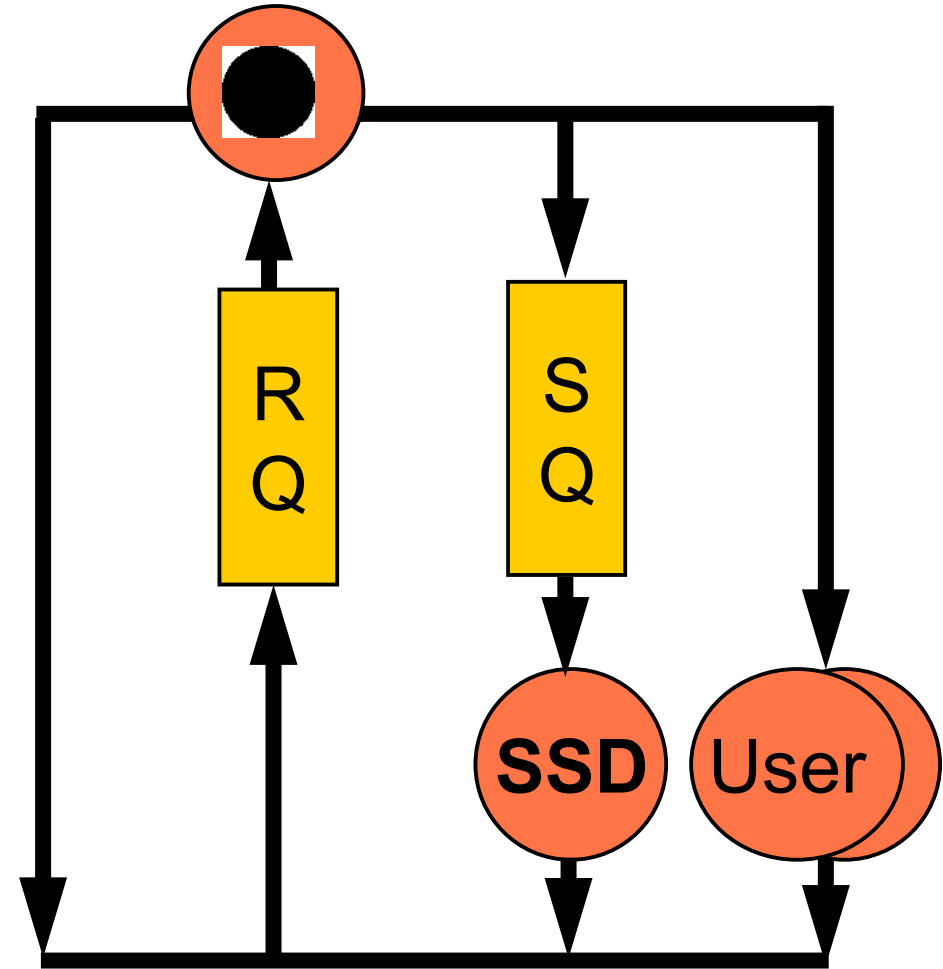


A much faster read

- Buffer contains 2,048 unread bytes
- P_0 requests 1,024 of them
 - Without requiring an SSD access
 - Much faster
 - Assume a zero delay
- Buffer will still contain 1,024 unread bytes

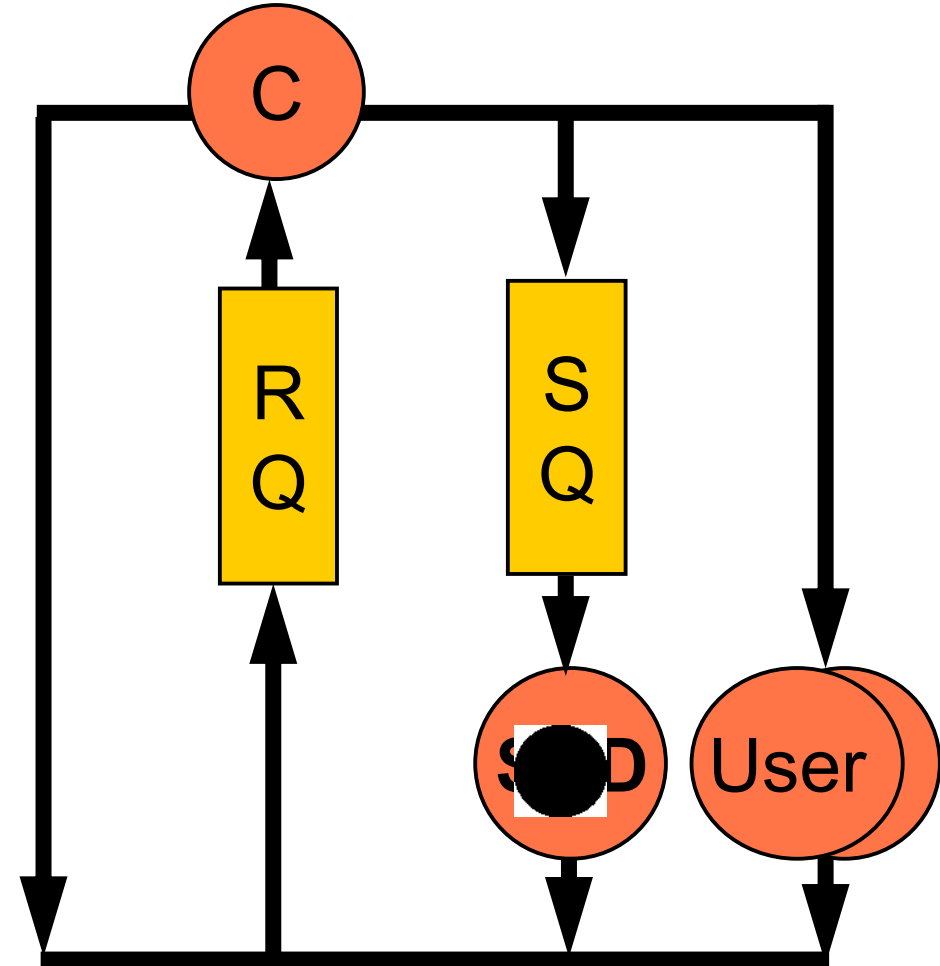
P_0 requests and gets the CPU until
 $t = 270.1 + 10 = 280.1\text{ms}$

BSIZE 4096
START 40
CORE 200
READ 2048
CORE 30
READ 1024
CORE 10
READ 2048
CORE 40
...



P_0 releases the CPU and performs an SSD access until $t = 280.1 + 0.1 = 280.2\text{ms}$

BSIZE 4096
START 40
CORE 200
READ 2048
CORE 30
READ 1024
CORE 10
READ 2048
CORE 40
...





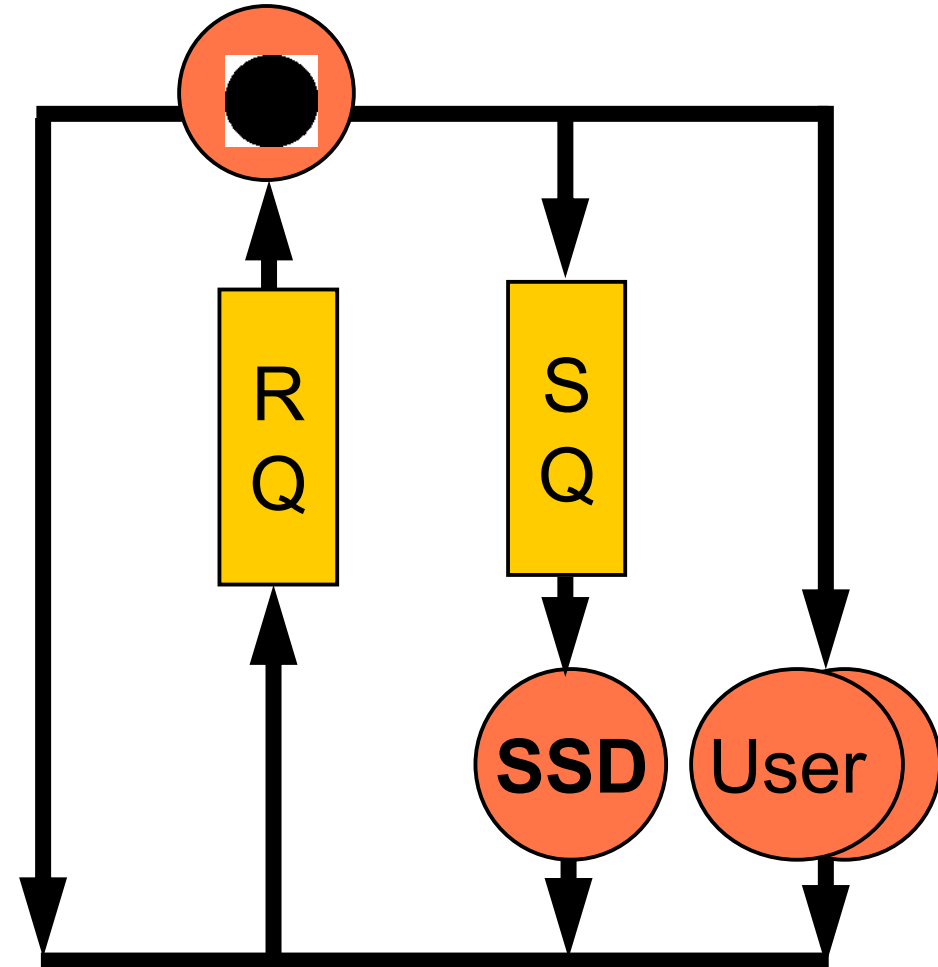
The SSD access

- P_0 requests 2,048 bytes
 - Buffer only contains 1,024 read bytes
 - P_0 will need an SSD access to get the 1,024 missing bytes
 - Will bring in 4,096 bytes
- Buffer will end with $4,096 - 1,024 = 3,072$ unread bytes

P_0 requests and gets the CPU until
 $t = 280.2 + 40 = 320.2\text{ms}$

BSIZE 4096
START 40
CORE 200
READ 2048
CORE 30
READ 1024
CORE 10
READ 2048
CORE 40

...

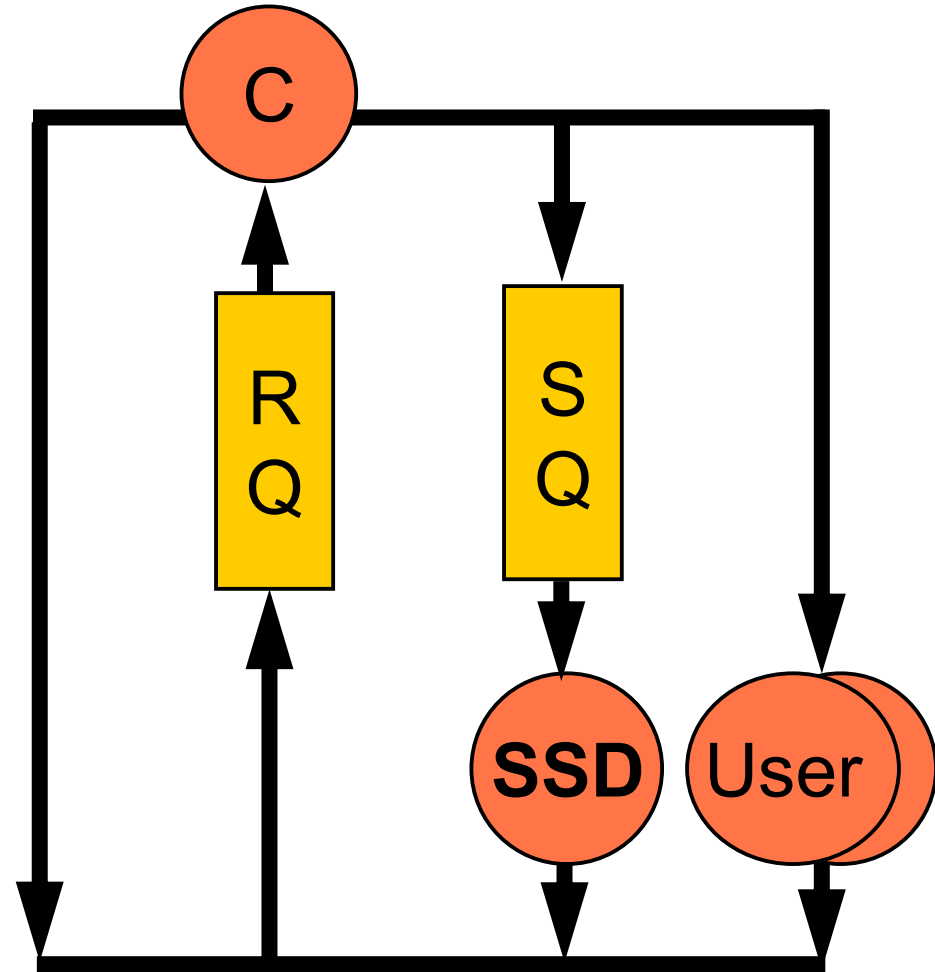




Introducing contention for resources

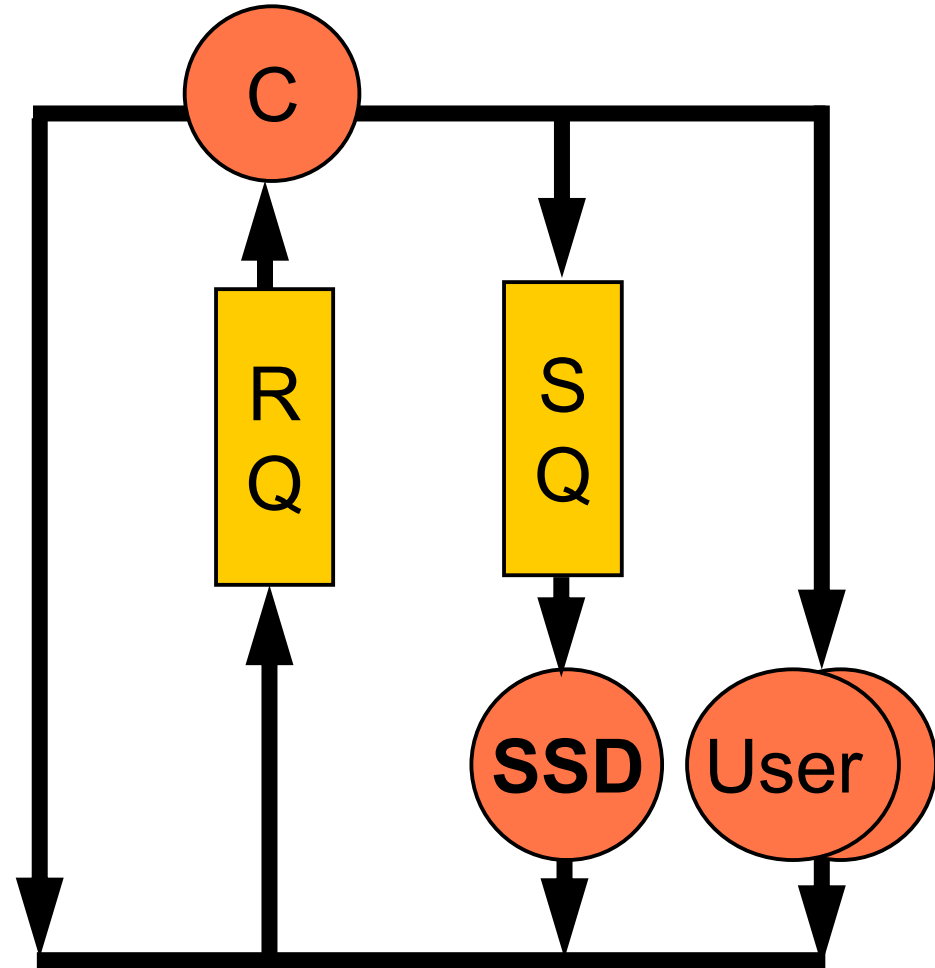
Two very short processes

BSIZE 4096
START 0
CORE 200
WRITE 4096
CORE 30
START 100
CORE 80
WRITE 4096
CORE 40
...



Process P_0 arrives at $t = 0\text{ms}$

BSIZE 4096
START 0
CORE 200
WRITE 4096
CORE 30
START 100
CORE 80
WRITE 4096
CORE 40
...



P_0 requests and gets the CPU until
 $t = 0 + 200 = 200\text{ms}$

BSIZE 4096

START 0

CORE 200

WRITE 4096

CORE 30

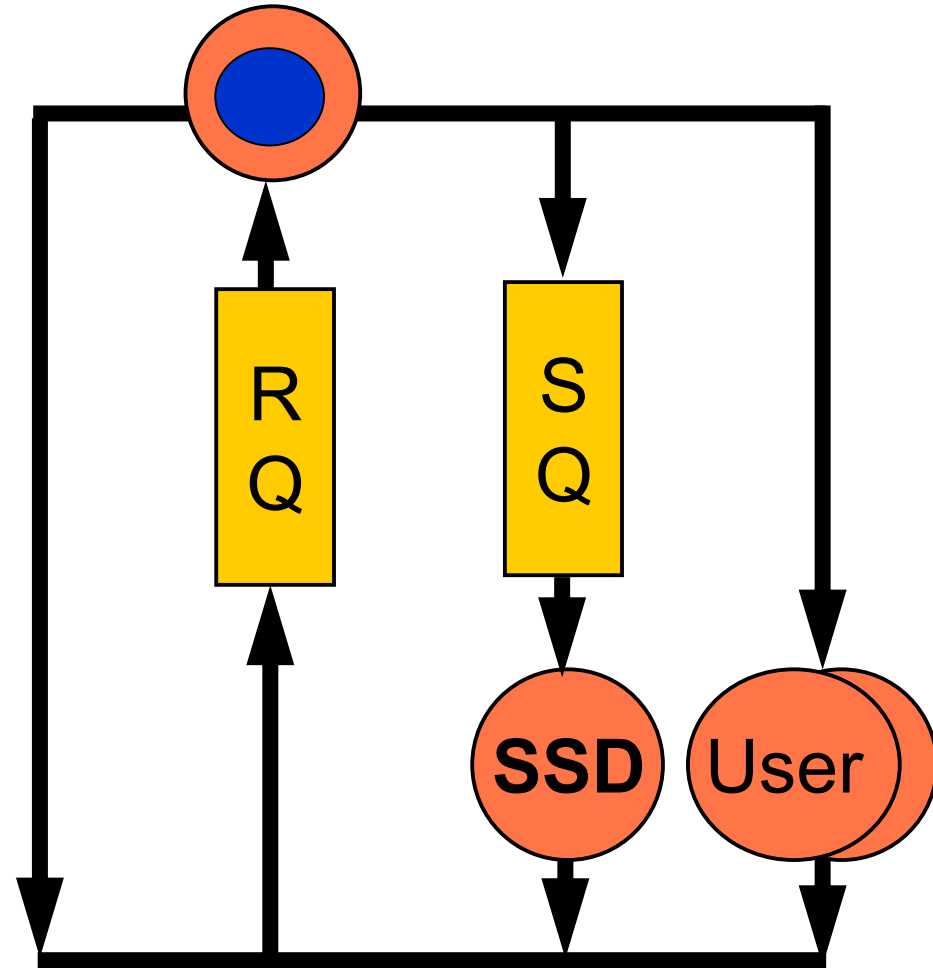
START 100

CORE 80

WRITE 4096

CORE 40

...



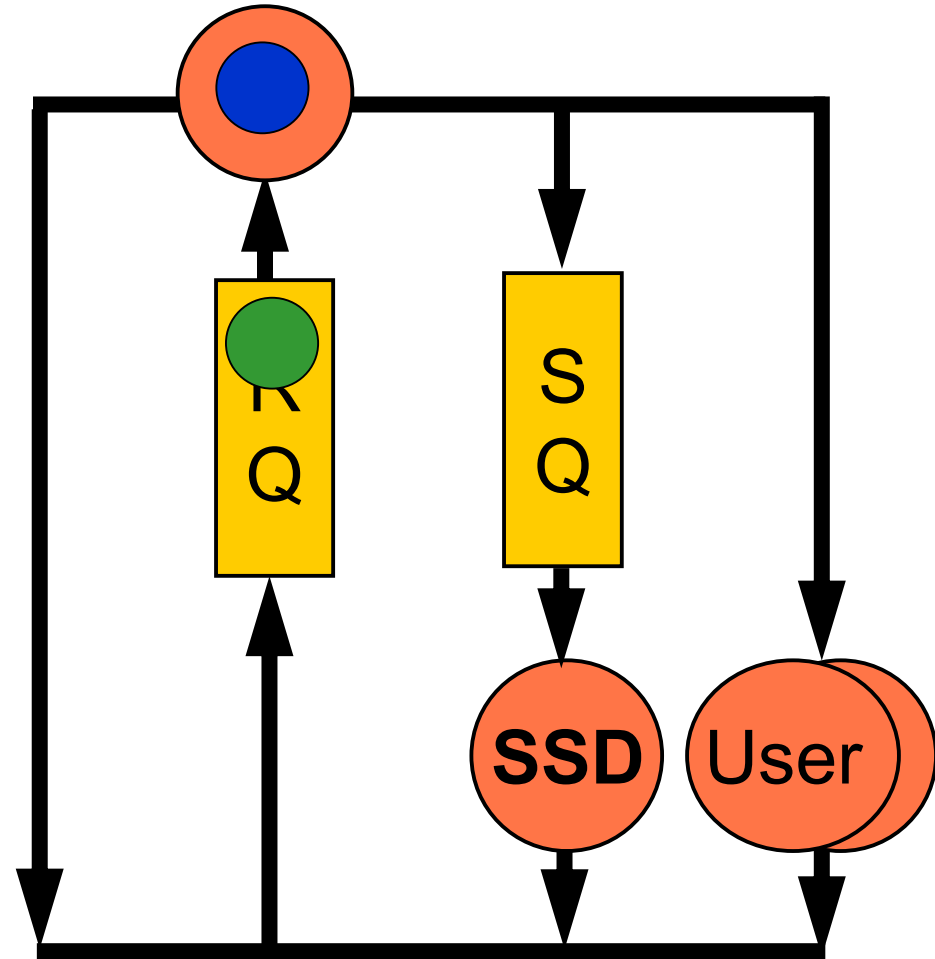


What is the next step?

- The two next events are
 - P_0 will complete its core step at $t = 200\text{ms}$
 - P_1 will arrive at $t = 100\text{ms}$
- And the first one is
 - P_1 will arrive at $t = 100\text{ms}$

P_1 waits for the CPU at $t = 100\text{ms}$

BSIZE 4096
START 0
CORE 200
WRITE 4096
CORE 30
START 100
CORE 80
WRITE 4096
CORE 40
...





What is the next step?

- The sole scheduled event is
 - P_0 will complete its core step at $t = 200\text{ms}$
- We have no choice

P_1 gets the CPU until $t = 200 + 80 = 280\text{ms}$

P_0 uses the SSD until $t = 200 + 0.1 = 200.1\text{ms}$

BSIZE 4096

START 0

CORE 200

WRITE 4096

CORE 30

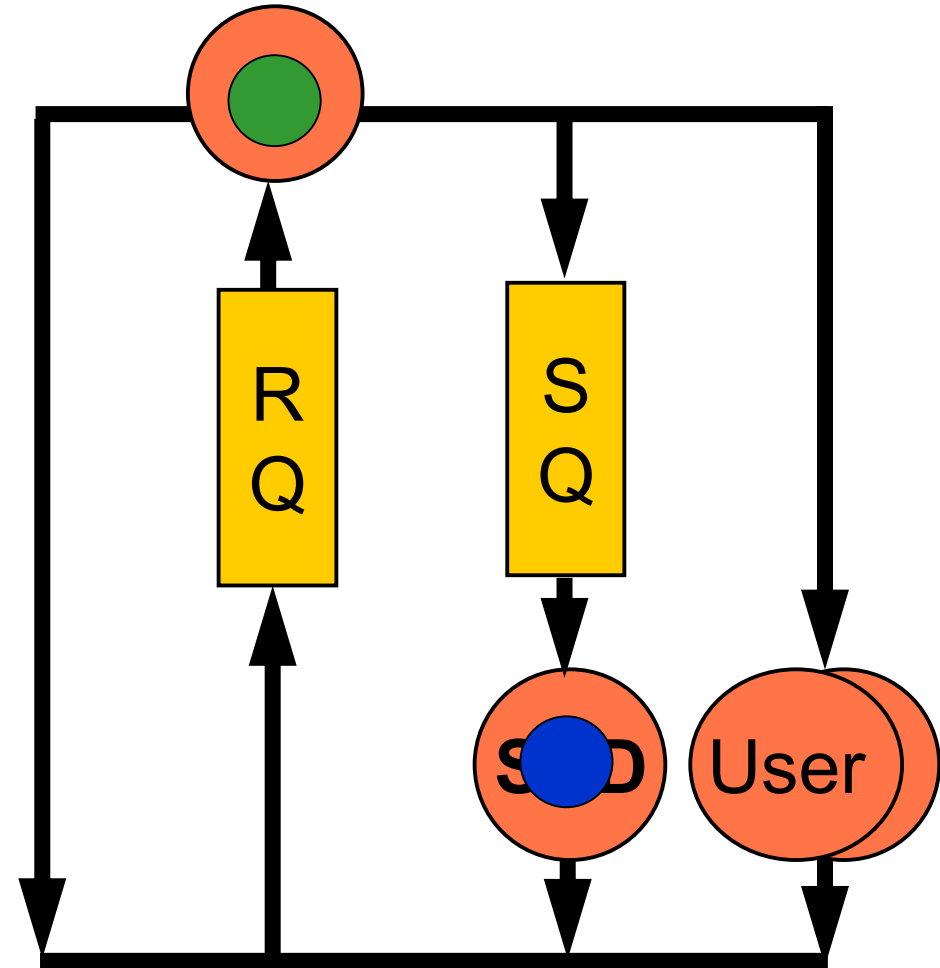
START 100

CORE 80

WRITE 4096

CORE 40

...



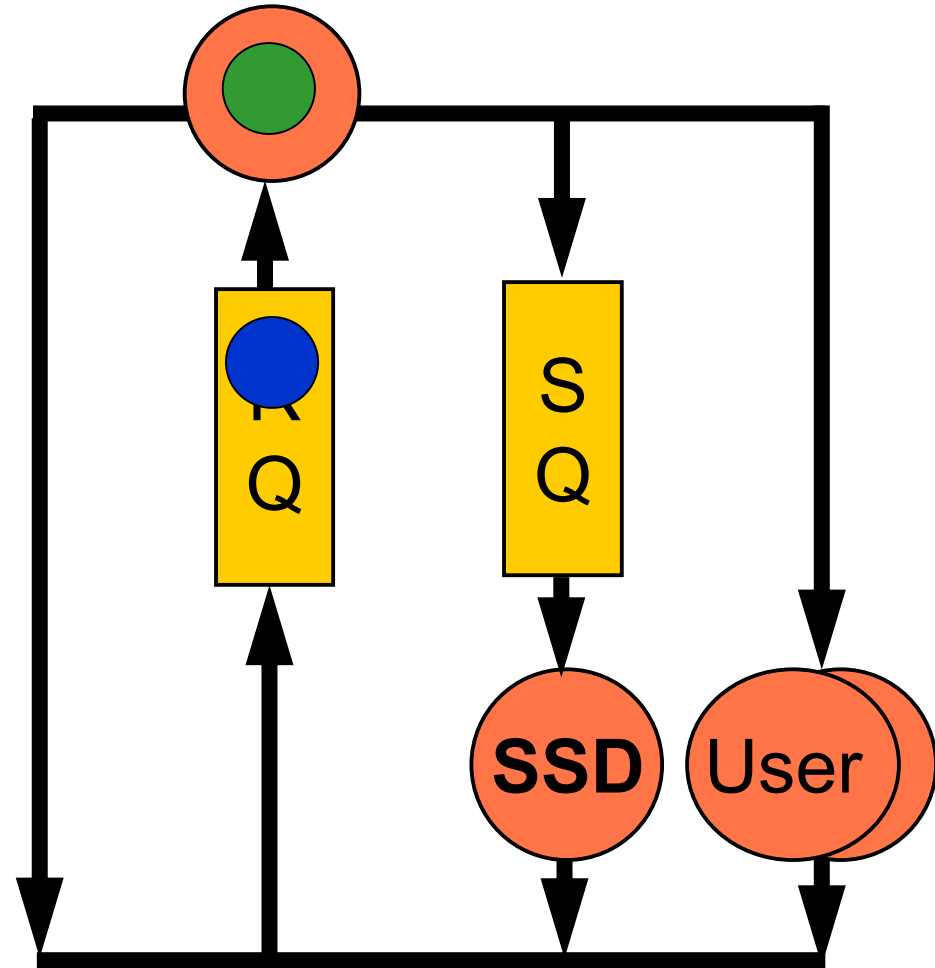


What is the next step?

- The two next events are
 - P_0 will complete its SSD write at $t = 200.1\text{ms}$
 - P_1 will complete its core step at $t = 280\text{ms}$
- And the first one is
 - P_0 will complete its SSD write at $t = 200.1\text{ms}$

P_1 has the CPU until $t = 280\text{ms}$
 P_0 must wait

BSIZE 4096
START 0
CORE 200
WRITE 4096
CORE 30
START 100
CORE 80
WRITE 4096
CORE 40
...





What is the next step?

- The sole scheduled event is
 - P_1 will complete its core step at $t = 280\text{ms}$
- We have no choice

P_0 gets the CPU until $t = 280 + 30 = 310\text{ms}$

P_1 uses the SSD until $t = 280 + 0.1 = 280.1\text{ms}$

BSIZE 4096

START 0

CORE 200

WRITE 4096

CORE 30

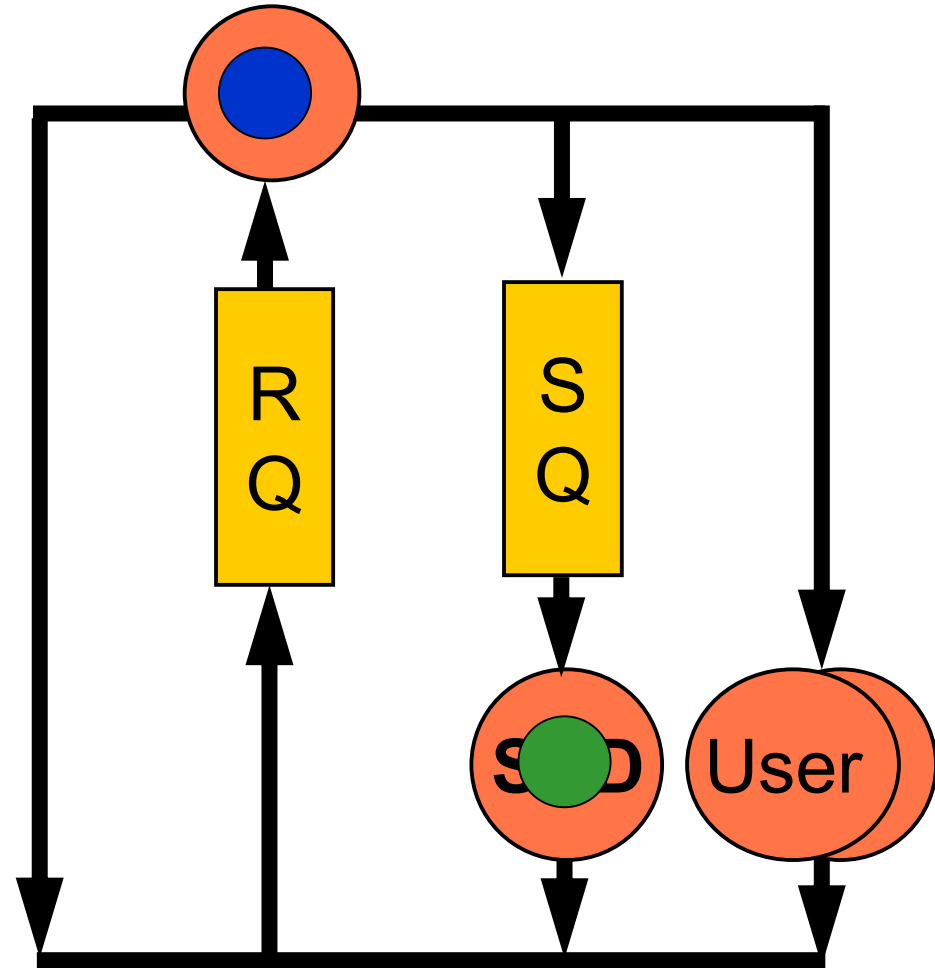
START 100

CORE 80

WRITE 4096

CORE 40

...



What is the next step?

- The two next events are
 - P_0 will complete its core step at $t = 310\text{ms}$
 - P_1 will complete its SSD write at $t = 280.1\text{ms}$
- And the first one is
 - P_1 will complete its SSD write at $t = 280.1\text{ms}$

P_0 keeps the CPU until $t = 310\text{ms}$

P_1 must wait

BSIZE 4096

START 0

CORE 200

WRITE 4096

CORE 30

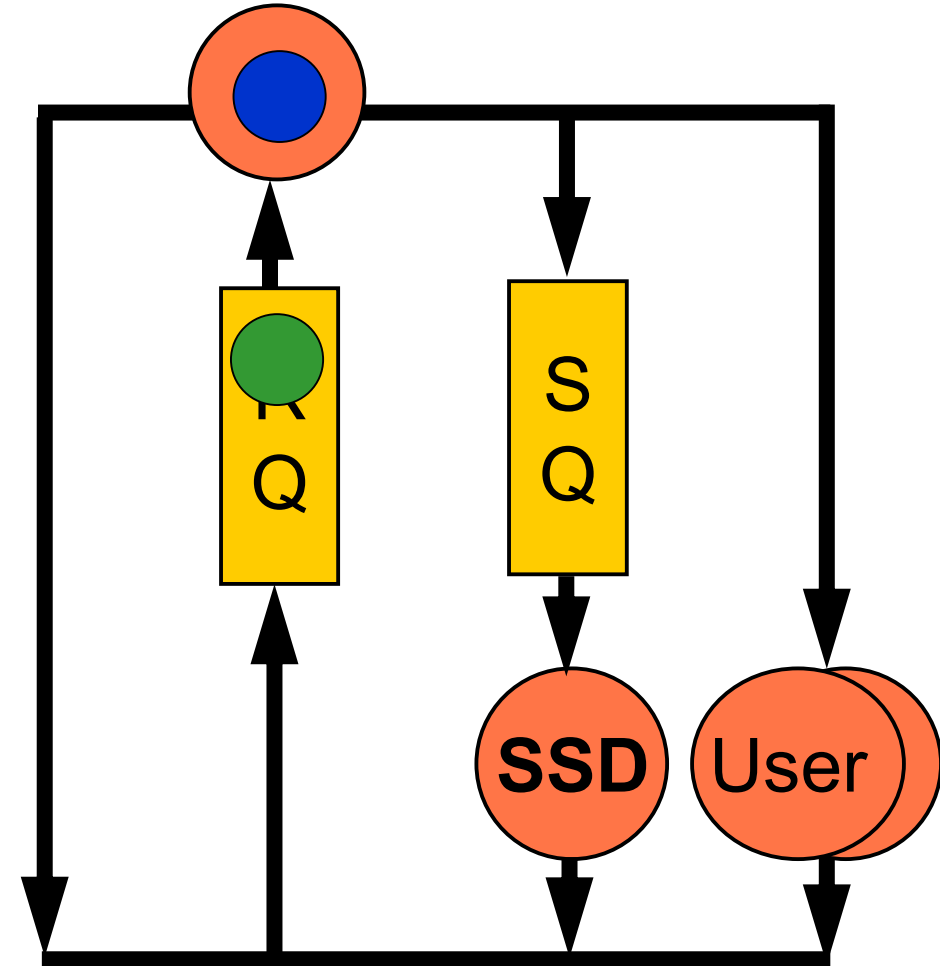
START 100

CORE 80

WRITE 4096

CORE 40

...





What is the next step?

- The sole scheduled event is
 - P_0 will complete its core step at $t = 310\text{ms}$
- We have no choice

P_0 terminates at $t = 310\text{ms}$

P_1 gets the CPU until $t = 310 + 40 = 350\text{ms}$

BSIZE 4096

START 0

CORE 200

WRITE 4096

CORE 30

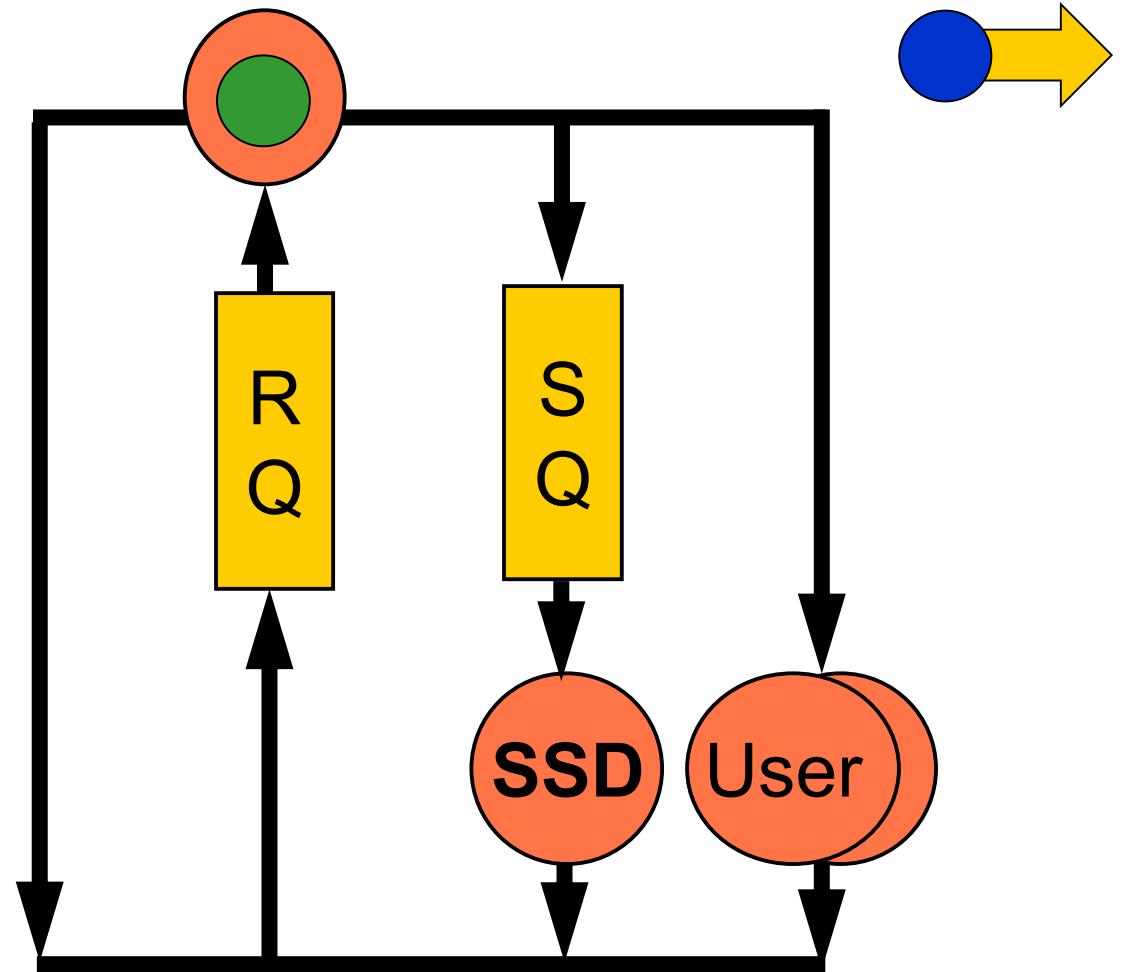
START 100

CORE 80

WRITE 4096

CORE 40

...





Required output

Process 0 terminates at $t = 310\text{ms}$.

It performed 0 physical read(s), 0 logical read(s), and 1 physical write(s).

Process states:

0 TERMINATED

1 RUNNING

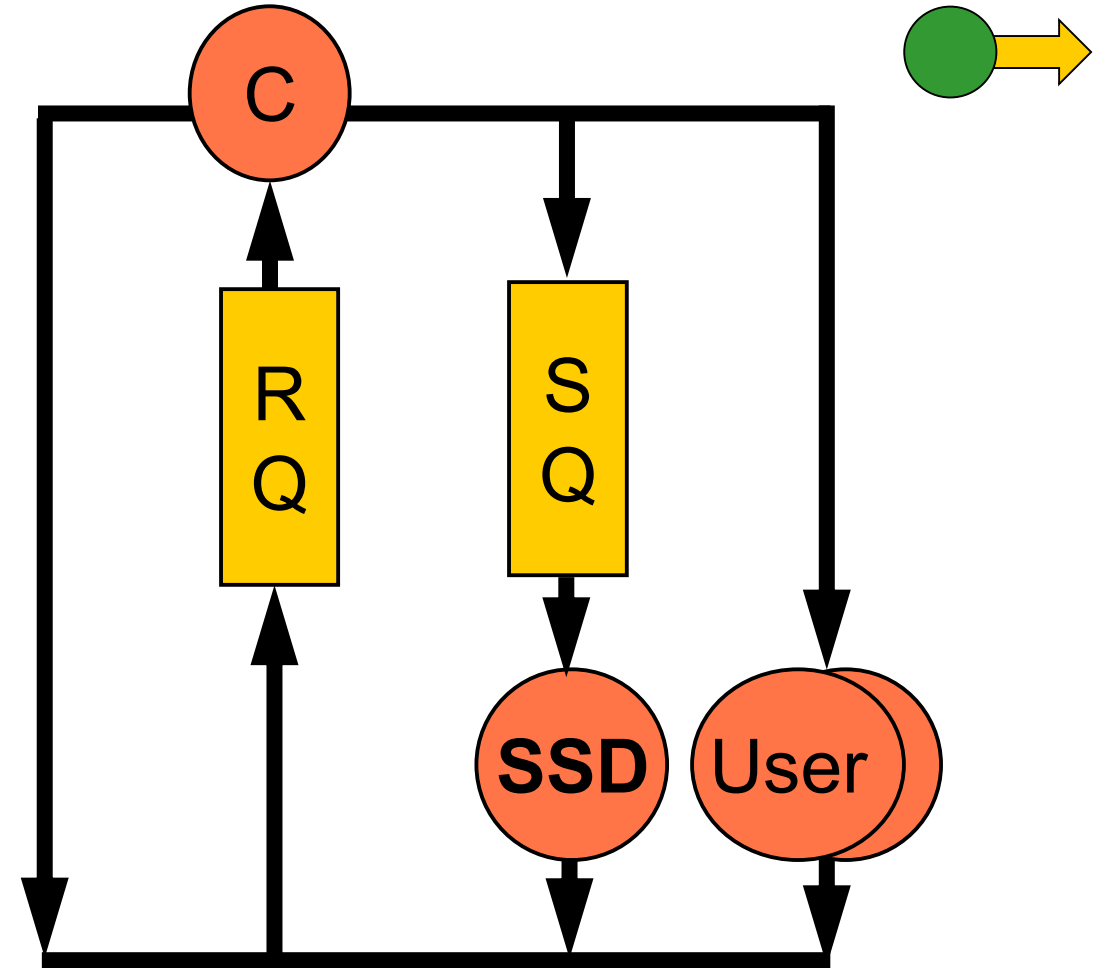


What is the next step?

- The sole scheduled event is
 - P_1 will complete its core step at $t = 350\text{ms}$
- We have no choice

P_1 terminates at $t = 350\text{ms}$

BSIZE 4096
START 0
CORE 200
WRITE 4096
CORE 30
START 100
CORE 80
WRITE 4096
CORE 40
...





Required output

Process 1 terminates at $t = 350\text{ms}$.

It performed 0 physical read(s), 0 logical read(s), and 1 physical write(s).

Process states:

1 TERMINATED



Engineering the simulation



Identifying processes

- Identify processes
 - By order in input list
 - Starting at 0



Events

- ***Absolutely nothing happens to our model between two successive "events"***
- ***Events are***
 - Arrival of a new process
 - Completion of a processing step
- We associate an ***event routine*** with each event



The simulation scheduler

1. Find the next event to process by looking at:
 - ✓ Core request completion times
 - ✓ SSD request completion times
 - ✓ INPUT and DISPLAY request completion time
 - ✓ Arrival time of the next process
2. Set current time to that time
3. Process event routine
4. Repeat until all processes are done



Simulating time

- The simulation clock does not advance by fixed increments
- It jumps from the current time to the time of the next 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**
 - **Operation (CPU, SSD, INPUT, DISPLAY)**
 - **Process sequence number**



Organizing our program (II)

- Do the same with process starts
 - **Time of process arrival**
 - **Process start**
 - **Process sequence number**

Organizing our program (II)

- Process all event notices in chronological order

t = 200 CPU 1	t = 120 SSD 5	t = 245 IO 9
---------------------	---------------------	--------------------



First notice to
be processed

*As INPUT and DISPLAY
operations will be handled
in the same manner,
we will have a common
IO event*



Organizing our program (III)

- Overall organization of main program

```
read in input file
fill event list with process arrival events
while (event list is not empty) {
    process next event in list
} // while
```

Organizing our program (IV)

- pop next event from list
clock = event.time
if (event.type is arrival) {
 arrival(event.time, event.processID);
else if (event.type is core) {
 core(event.time, event.processID);
else if (event.type is ssd) {
 ssd(event.time, event.processID)
else if (event.type is INPUT or DISPLAY) {
 io(event.time, event.processID);

In reality

- pop next event from list
clock = event.time
if (event.type is arrival) {
 arrival(event.time, event.processID);
else if (event.type is core) {
 core(event.time, event.processID);
else if (event.type is read) {
 read(event.time, event.processID)
else if (event.type is write) {
 write(event.time, event.processID)
else if (event.type is INPUT or DISPLAY) {
 io(event.time, event.processID);



Handling SSD writes

- SSD writes are blocking
 - Each write will result in an SSD access
- Easiest option
 - Replace the write request function by the SSD request function
 - ...

```
else if (event.type is write) {  
    ssd(event.time, event.processID)
```



Handling SSD reads

- Each process will keep track of the number of unread bytes in its buffer
 - Initially zero
- If the read can be satisfied using the bytes already in the buffer
 - Update the number of unread bytes
 - Start the next process request
- Otherwise
 - Fetch enough blocks to satisfy the request
 - Update the number of unread bytes
 - Start an SSD request

The pseudo-code

```
if (n_request <= n_buffer) {  
    n_buffer = n_buffer - n_request;  
    Handle next process request  
} else {  
    n_missing = n_request - n_buffer;  
    if (n_missing%BSIZE == 0 ) {  
        n_brought_in = n_missing;  
    }else {  
        n_brought_in = (n_missing/BSIZE +1)*BSIZE;  
    }  
    n_buffer = n_brought_in - n_missing;  
    ssd(event.time, event.processID);  
}
```

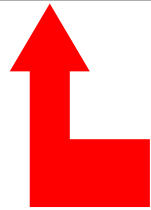
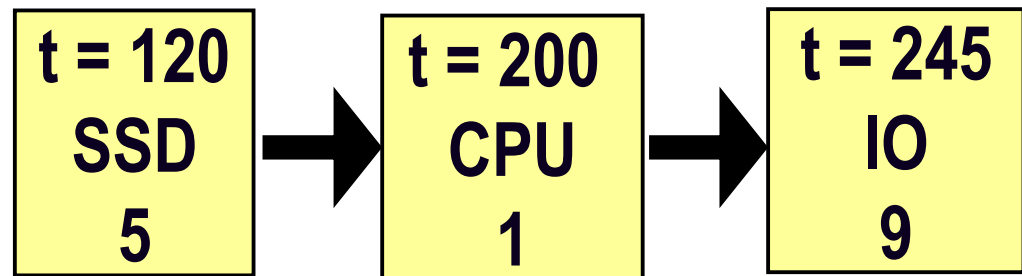


Organizing our event list (I)

- As a priority queue
- Associate an ***arrival time***
 - With each process
- Associate a ***completion time***
 - With each current core request
 - With each SSD request
 - With each INPUT or DISPLAY request

Organizing our event list

- Process all event notices in time order



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



A solution (I)

- ***Input module***

- Read in all input data
 - Store them in an input list
- Create arrival events for all processes

- ***Main loop***

- Pops the next event from the event list
 - Process that event
 - Process goes to next step



A solution (II)

- ***Starting a process***

- Handled by a ***process arrival*** routine

- ***Handling process termination***

- Just print the required information

- ***Detecting the end of the simulation***

- Event list will be ***empty***

Core request

- If the CPU is FREE
 - Set CPU status to BUSY
 - Schedule a core completion event
- Else
 - Queue process in ready queue

Ready queue entries contain

- Process sequence number
- Duration of core request



Core completion event

- If the ready queue is not empty
 - Pop the first process in the ready queue
 - Schedule a core completion event for that process
- Else
 - Set CPU status to FREE
- Proceed with process next request
 - SSD, INPUT or DISPLAY



SSD request

- If the SSD is FREE
 - Schedule an SSD completion event
 - Set SSD status to BUSY
- Else
 - Queue process in SSD queue



SSD completion event

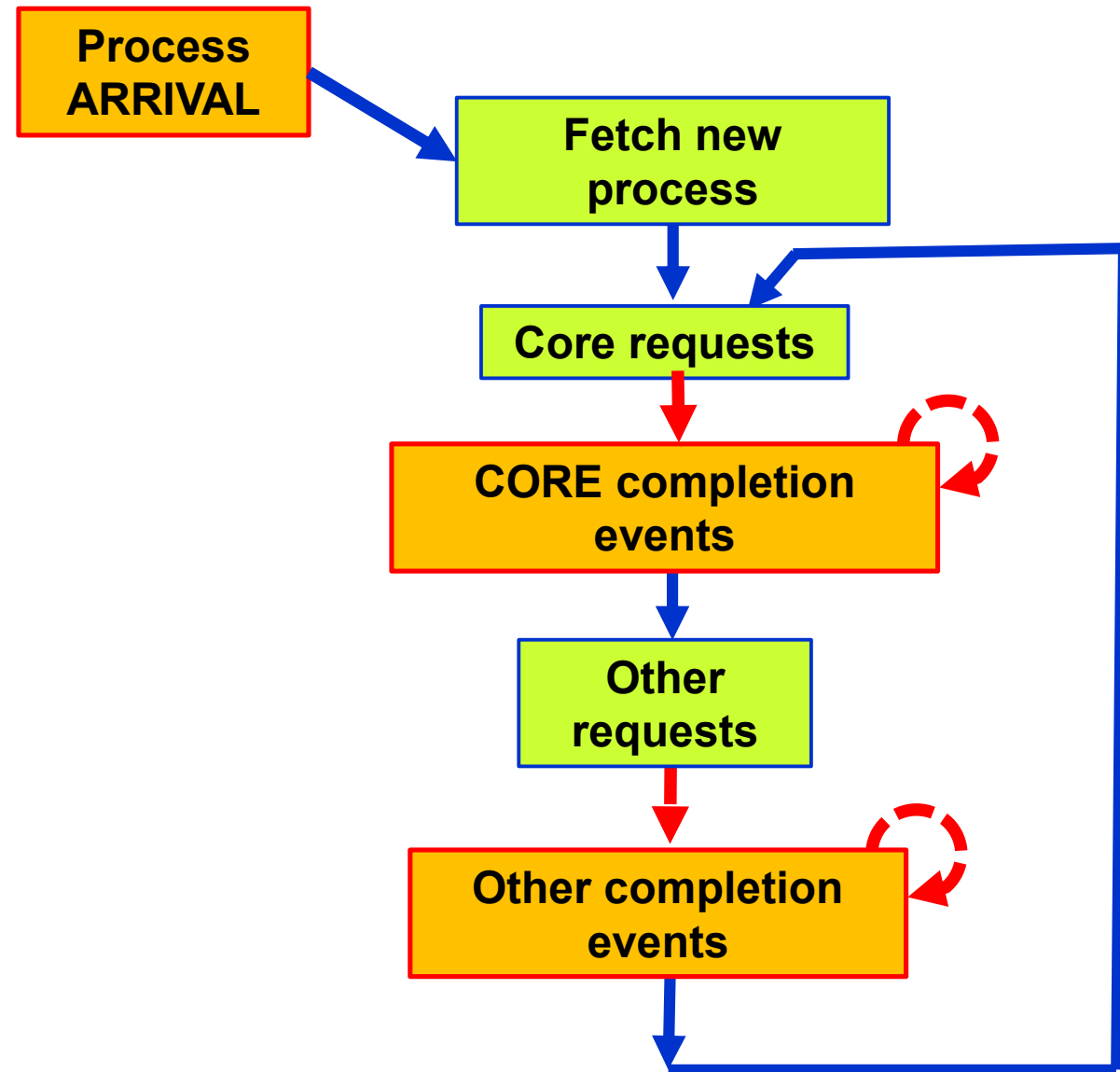
- If SSD queue is not empty
 - Pop the first process in the SSD queue
 - Schedule a SSD completion event for that process
- Else
 - Set SSD status to FREE
- Proceed with process next request
 - Necessarily a CORE request



INPUT and DISPLAY

- ***IO request***
 - Schedules an IO completion event
- ***IO completion event***
 - Proceed with process next request
 - Always a CORE request

Overall





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
 - A new process arrives
 - A core will be released by the process that used it
 - The SSD will released by the process that used it
 - A process will complete an INPUT or DISPLAY task



An implementation minimizing list usage

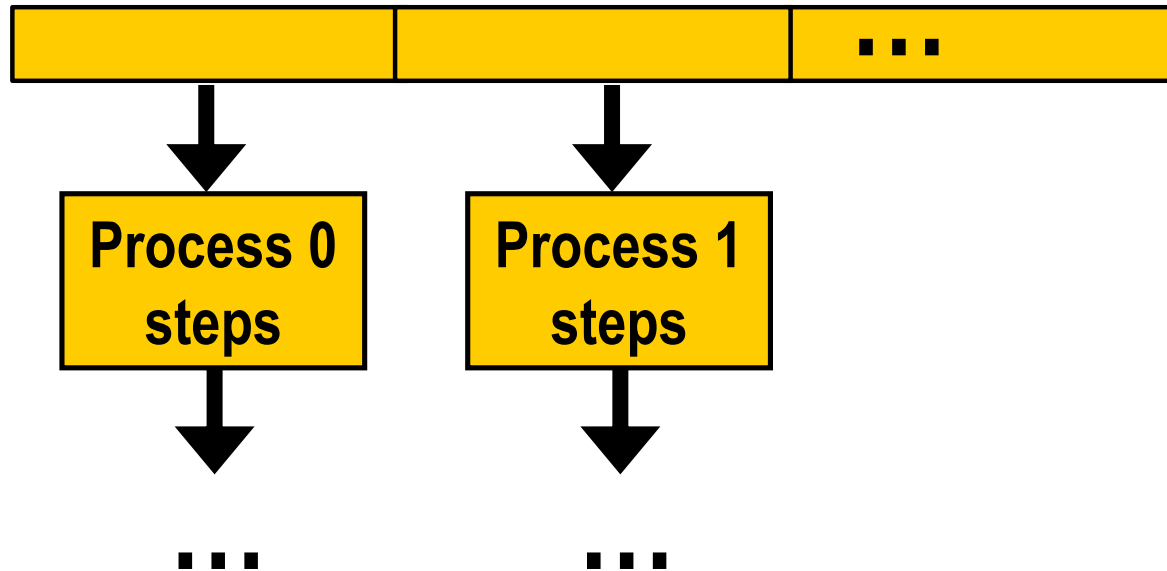
- My main data structures would include:
 - An input table
 - A process table

The input table

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

<i>Operation</i>	<i>Parameter</i>
BSIZE	4096
START	0
CORE	150
INPUT	500
CORE	80
DISPLAY	0
CORE	80
...	...

A more elegant input table



- ❑ A global array of pointers indexed by the process sequence
- ❑ The pointers point to the head of the list of steps of each process

The process table (I)

<i>Job ID</i>	<i>First Line</i>	<i>Last Line</i>	<i>Current Line</i>	<i>State</i>
3	1	4	varies	varies
4	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)

Detecting the end of data

- The easiest ways to do it
- If you use `scanf()`
 - `scanf(...)` will return **1** once it reaches the line with the **EOF**
 - `while(scanf(...) == 2) { ... }`
- If you use `cin`
 - `cin` will return **1** once it reaches the line with the **EOF**
 - `while ((cin >> keyword >> argument) == 2) { ... }`



Reading in your program input

- You **must** use I/O redirection

- `assign1 < input_file`

- ***Advantages***

- Very flexible

- Can write your code as if it was reading from standard input

- No need to mess with **`fopen()`**, **`argc`** and **`argv`**



Two bad solutions

- Your program reads its input from a hard-wired file
 - The TA will test your program using different input files
 - Will make them ***very unhappy***
- Your program prompts the user for the input file name
 - Not worth the effort

Happy TA, happy student, and happy instructor!



Still confused?

- Review pseudo code in the slides
- Draw sketches
- Post questions and read answers on Piazza
- Talk to the TAs and the instructor
 - Do not wait until the last minute
- Discuss your pseudocode with classmates
- ***Do not copy code from anyone or anywhere***
 - ***What you turn in must be your own work!***



Still discouraged?

- Turn in something
 - All submissions that compile will get some points
 - Can make a difference between and F and a better grade
- The safest ways to fail the course is
 - Not turning in anything
 - Turning in something you did not write