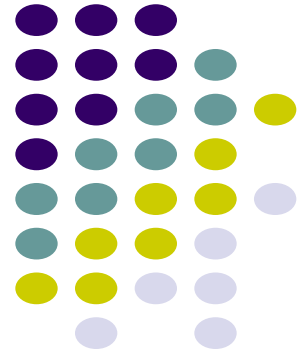


# More Scheduling and Interprocess communication

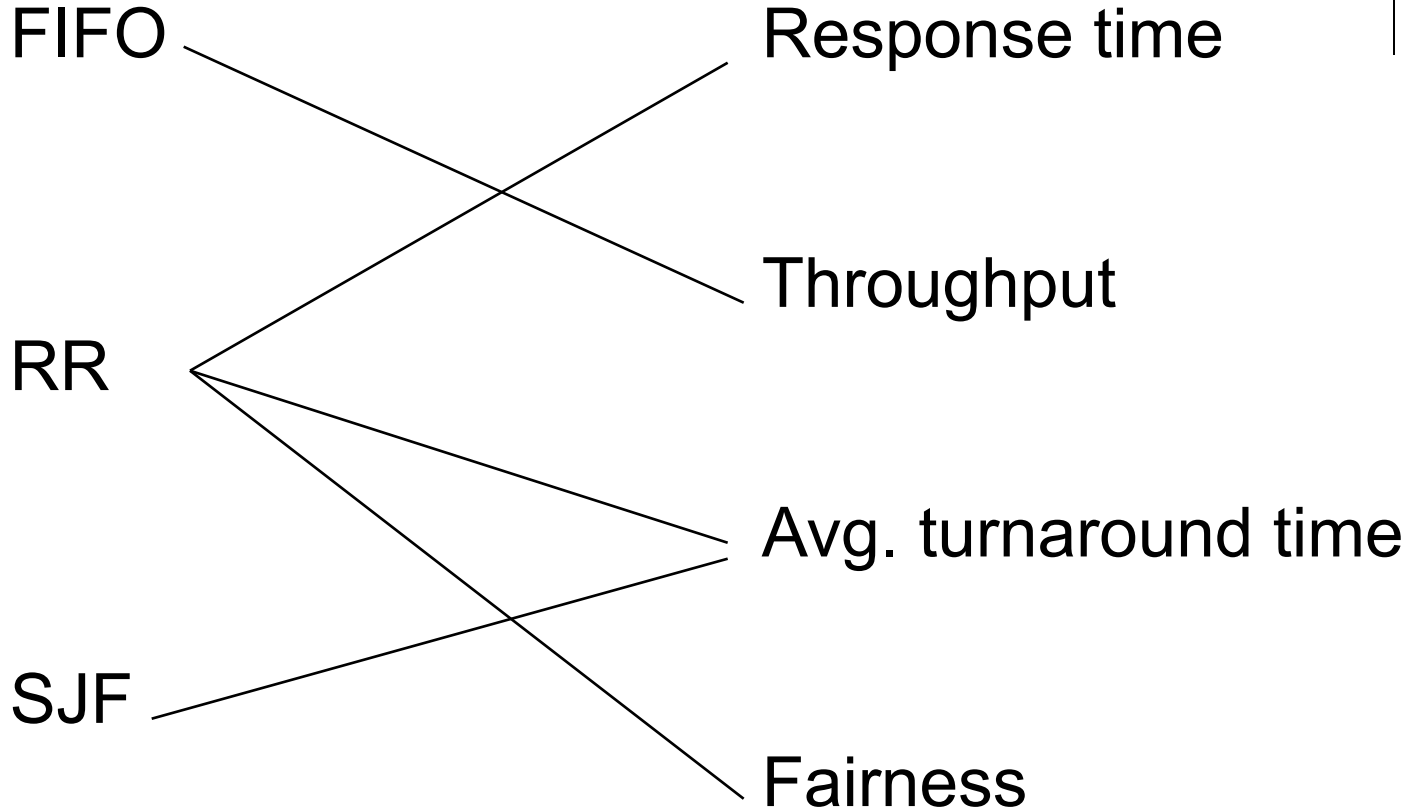
---

ECE 469, Mar 06

Aravind Machiry



# Recap: Scheduling Policies Advantages



# Priority Scheduling



- To accommodate the spirits of SJF/RR/FIFO
- The method
  - Assign each process a *priority*
  - Run the process with highest priority in ready queue first
    - Use FIFO for processes with equal priority
  - Adjust priority dynamically
    - To deal with *all* issues: e.g. aging, I/O wait raises priority

# Priority Scheduling



- Who sets the priorities
  - Internally by OS
    - I/O to computation ratio (can be dynamic)
    - Memory requirement (can be dynamic)
    - Time constraints (e.g. real-time systems)
  - Externally by users/sysadm
    - Importance
    - Funds paid for
    - Being nice
- Dynamically adjusting priority is tricky

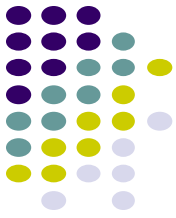
# Priority Scheduling

- Advantages
  - Flexibility.
- Disadvantages
  - Starvation: A low priority process might never get to run.

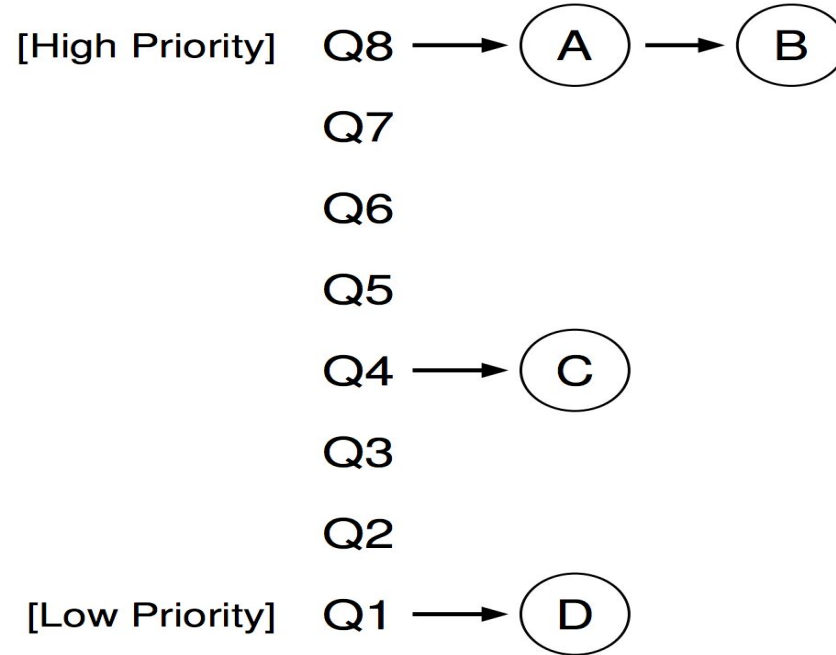


# Multiple Queue Scheduling

- Maintain multiple queues of different priorities.
  - Background queue vs Foreground queue
- Prefer jobs in higher priority queues.



# Multiple Queue Scheduling



# Multiple Queue Scheduling



- Challenges:
  - How to assign process to queues?
  - How to schedule processes within a queue?
  - How to switch between queues?



# Multiple Queue Scheduling



- How to assign processes to queues?
  - Based on User:
    - Sysadmin vs Regular user
  - Based on Type:
    - Foreground vs. background.

# Multiple Queue Scheduling



- How to schedule processes within a queue?
  - All queues have same scheduling policy.
  - Scheduling policy varies with queues:
    - Foreground queue - Need to be responsive:
      - Round Robin
    - Background queue - Shorter turnaround time:
      - SJF

# Multiple Queue Scheduling



- How to switch between queues?
  - Fixed priority preemptive scheduling (high-pri queue trumps other)
  - Time-slice between queues

# Multiple Queue Scheduling



- Pros:

# Multiple Queue Scheduling



- Pros:
  - Low scheduling overhead
    - Jobs do not move across queues

# Multiple Queue Scheduling



- Pros:
  - Low scheduling overhead
    - Jobs do not move across queues
- Cons:

# Multiple Queue Scheduling



- Pros:
  - Low scheduling overhead
    - Jobs do not move across queues
- Cons:
  - Processes permanently assigned to one queue – not flexible
    - Program behavior may change
    - E.g. can switch between I/O bound and CPU bound
    - Need some learning/adaptation at runtime
  - Starvation cannot be easily handled
    - Need some learning/adaptation at runtime

# Multilevel Feedback Queue (MLFQ)



- Problem: how to change priority?
- Jobs start at highest priority queue
- Feedback
  - Priority Decreases: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - Priority Unchanged: If a job gives up the CPU before the time slice is up, it stays at the same priority level.
  - Priority Increases: After a long time period, move all the jobs in the system to the topmost queue (aging)



# Multilevel Feedback Queue (MLFQ)



Q2



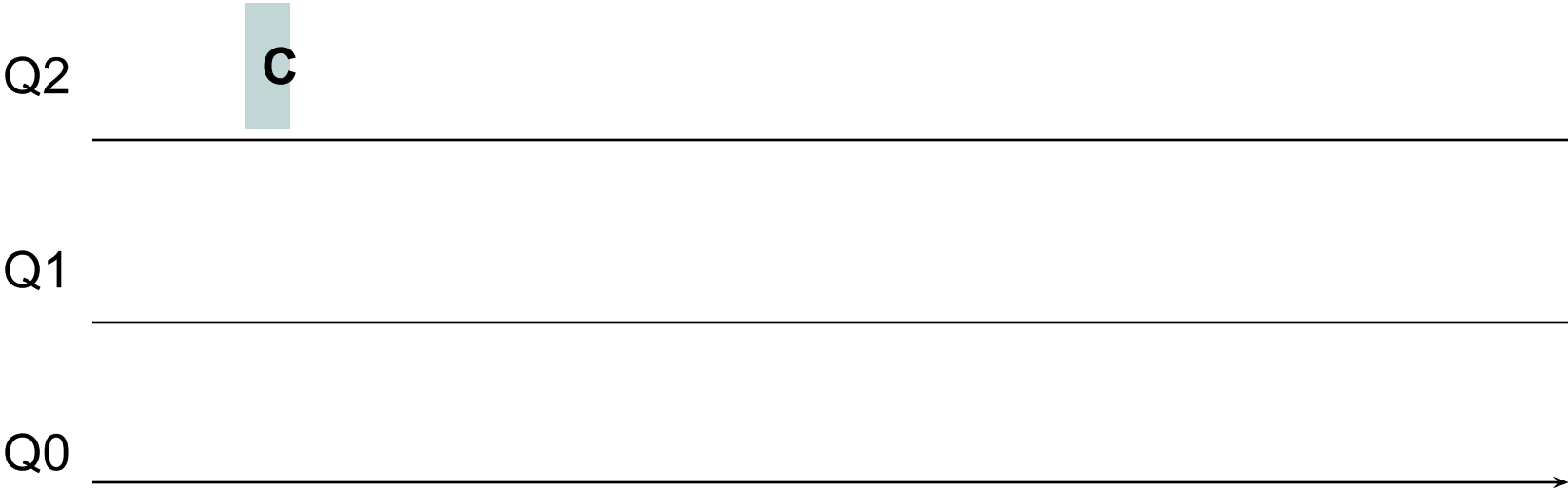
Q1



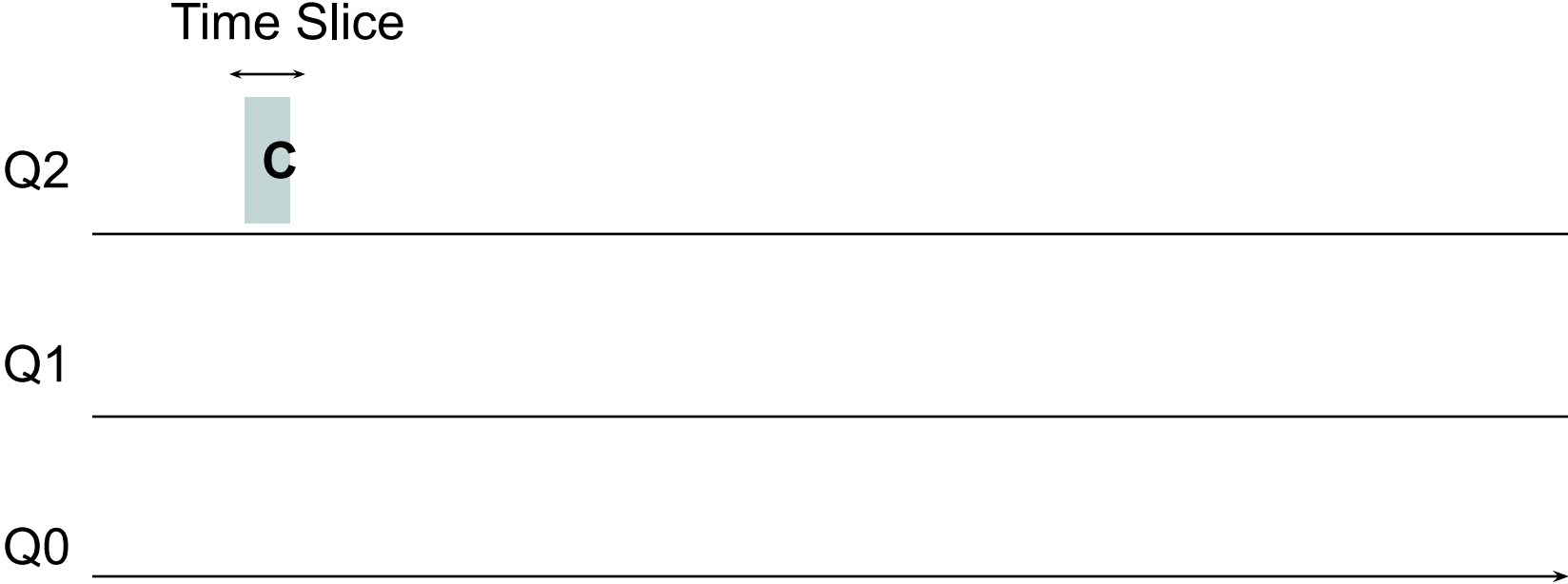
Q0



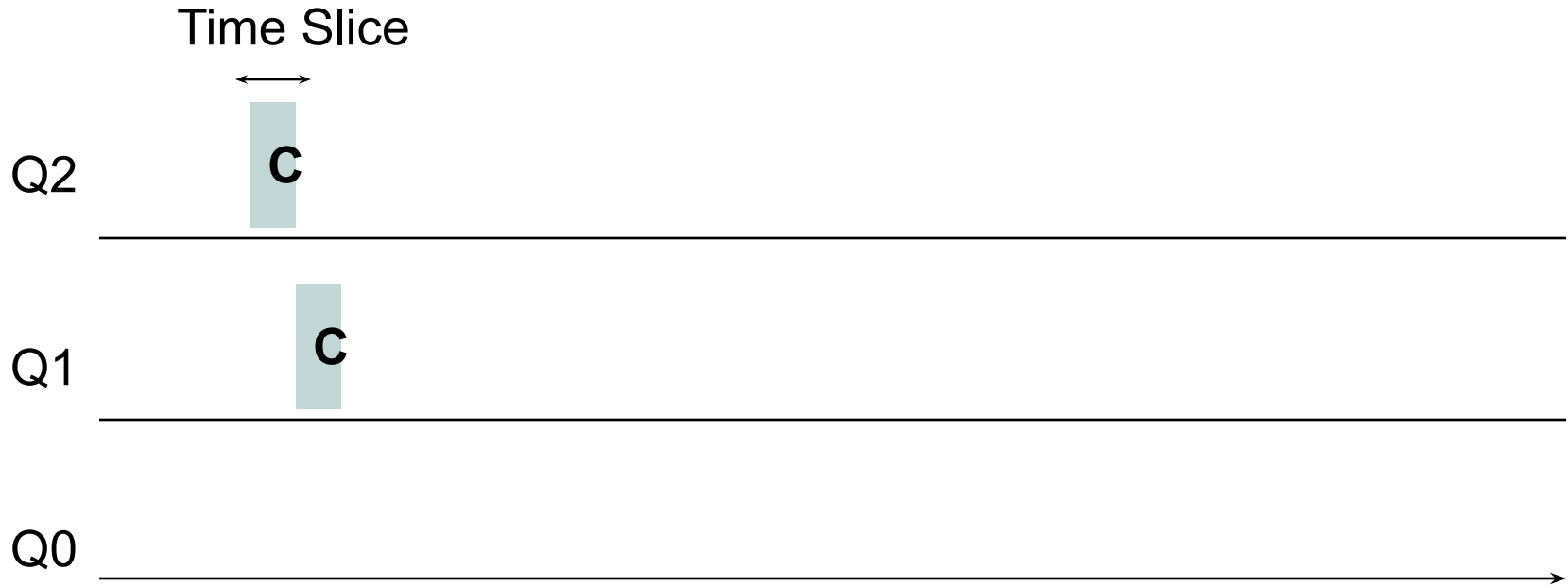
# Multilevel Feedback Queue (MLFQ)



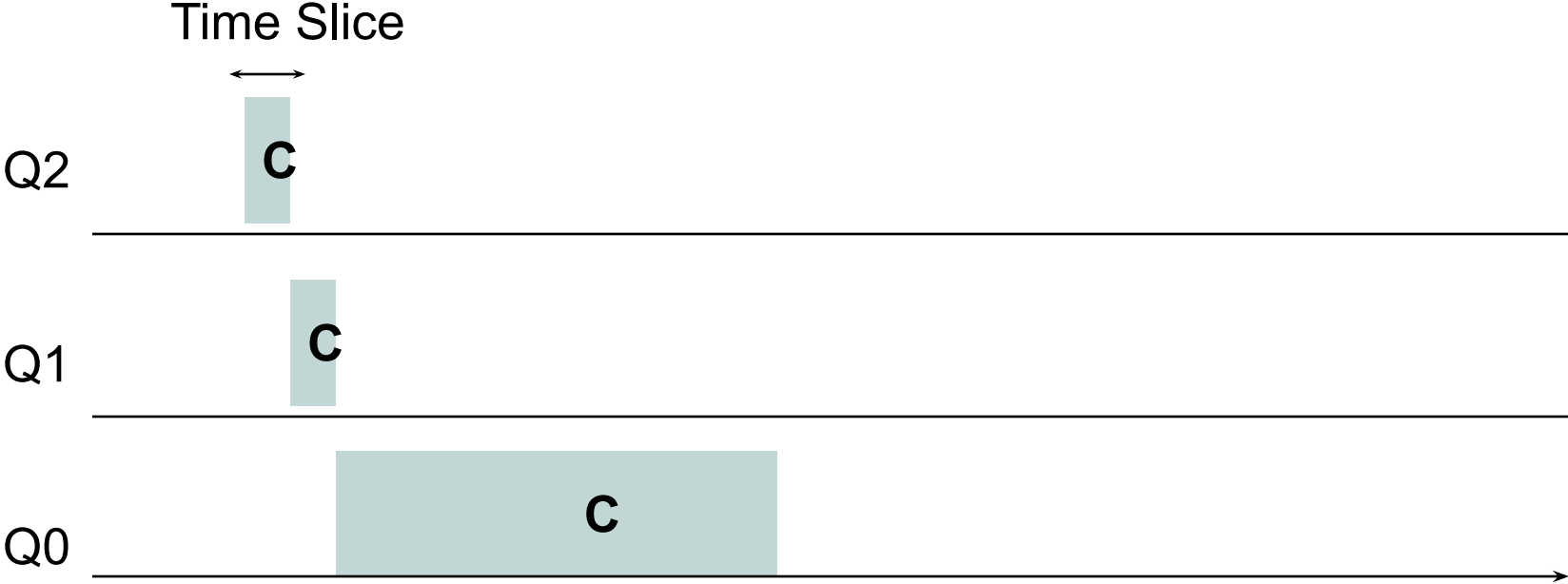
# Multilevel Feedback Queue (MLFQ)



# Multilevel Feedback Queue (MLFQ)



# Multilevel Feedback Queue (MLFQ)



# MLFQ: Long job and short jobs in between



Q2



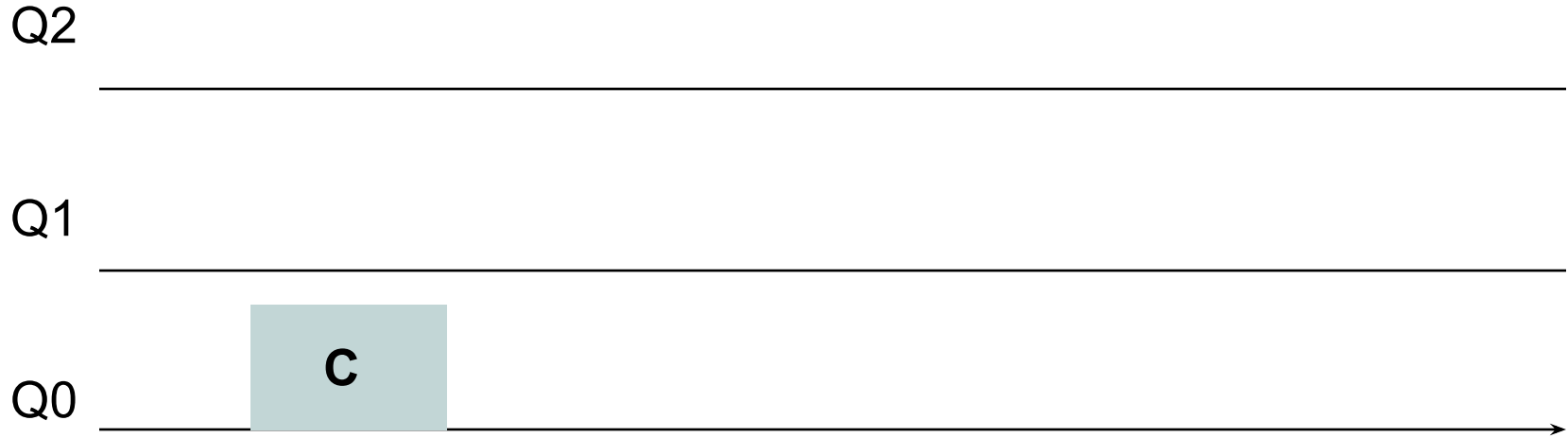
Q1



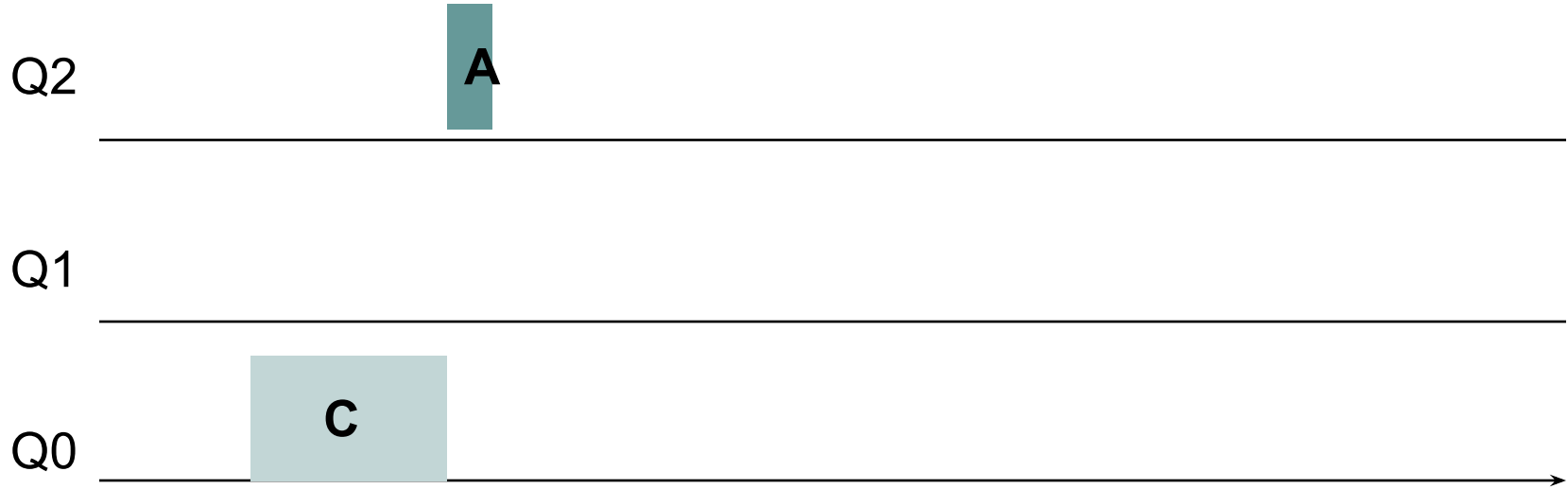
Q0



# MLFQ: Long job and short jobs in between

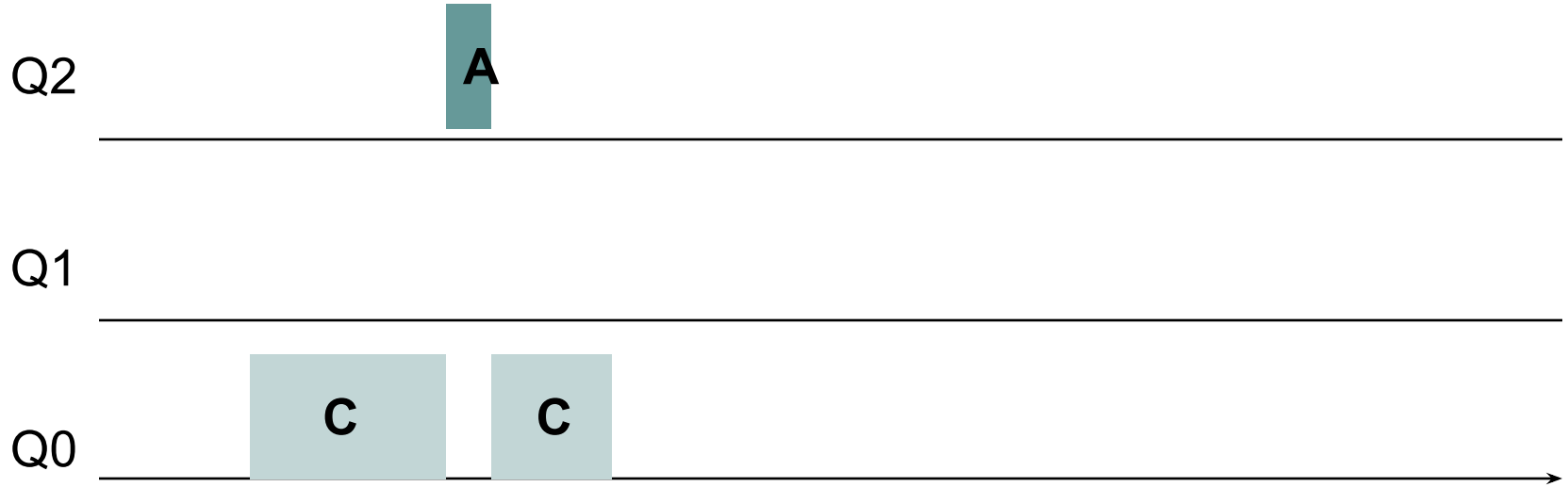


# MLFQ: Long job and short jobs in between

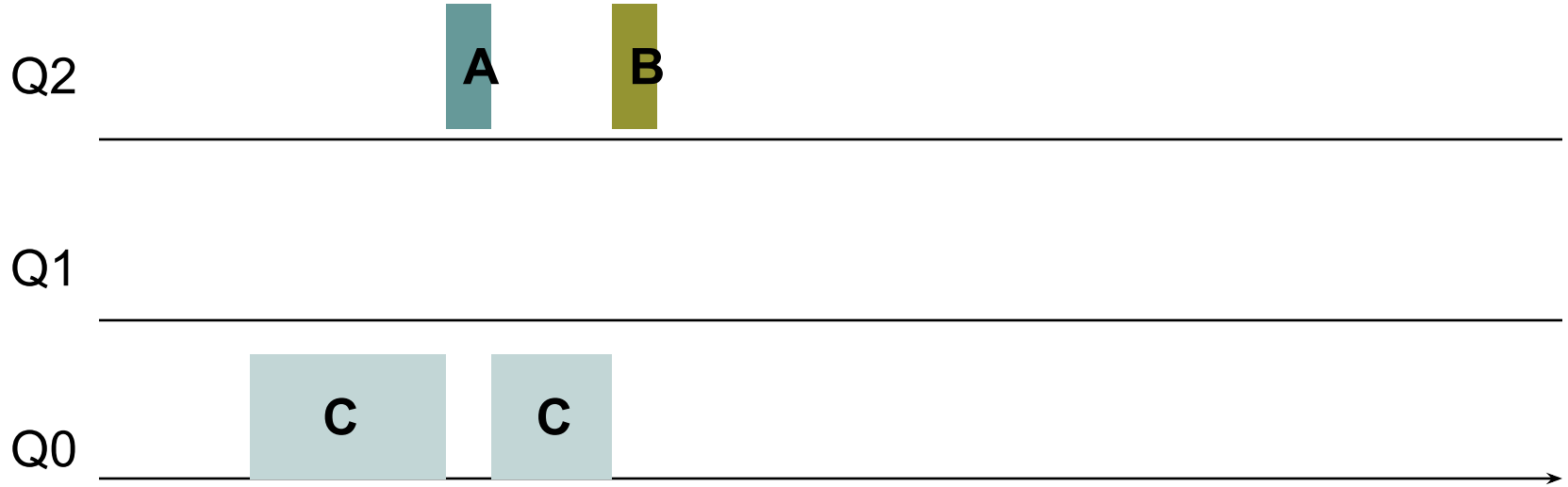




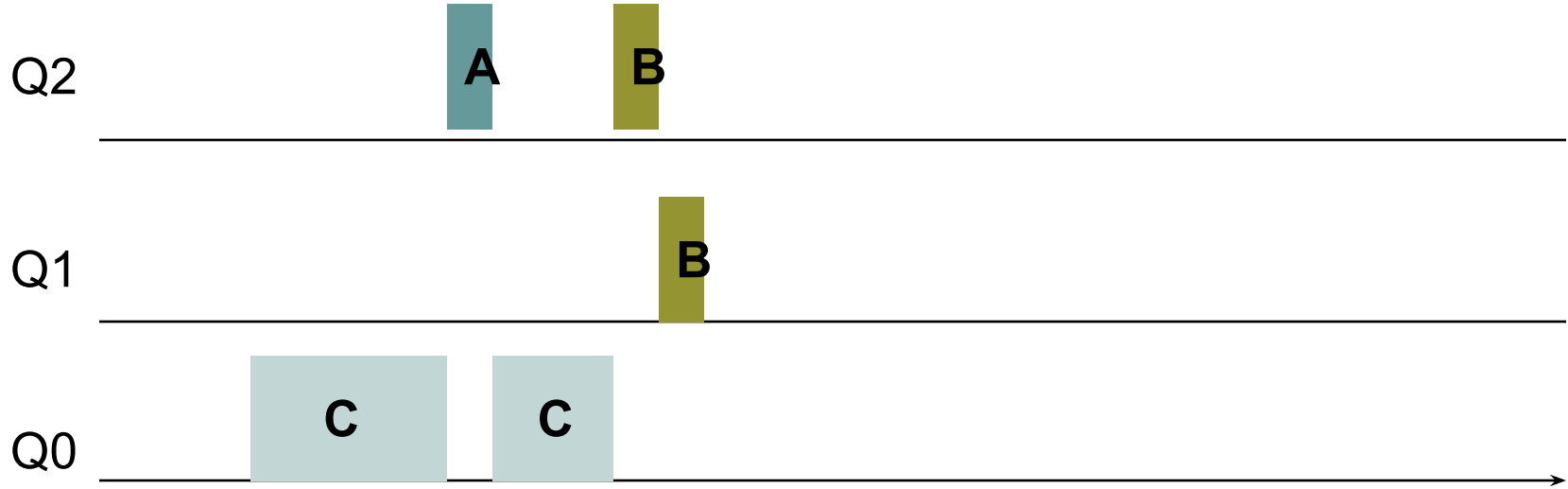
# MLFQ: Long job and short jobs in between



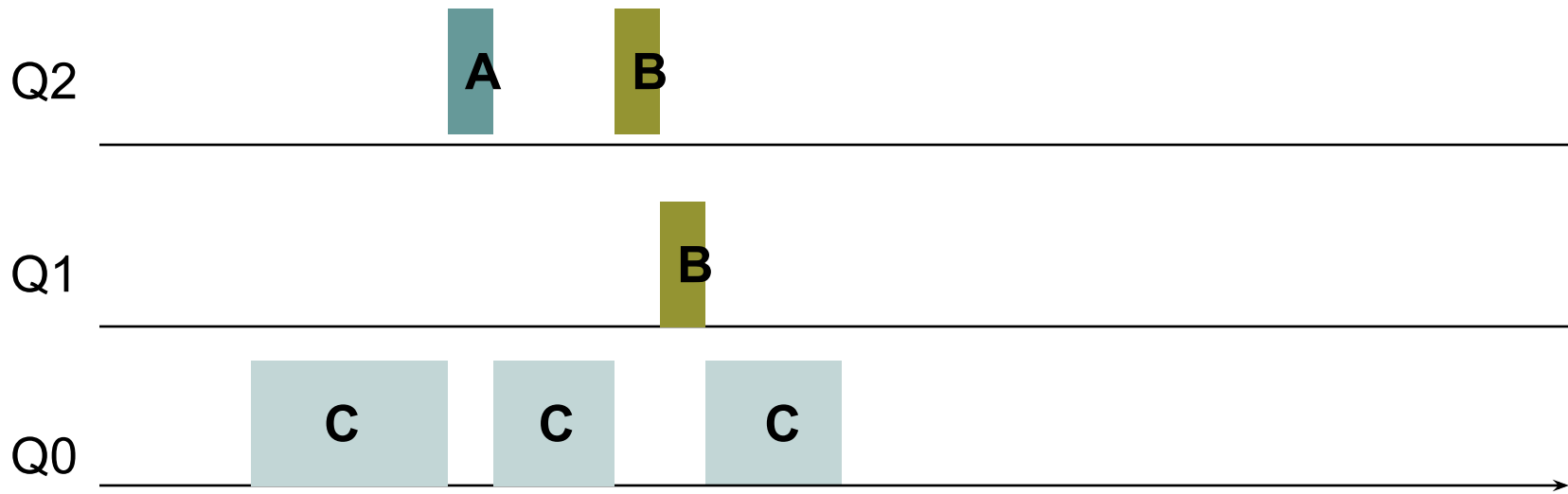
# MLFQ: Long job and short jobs in between



# MLFQ: Long job and short jobs in between



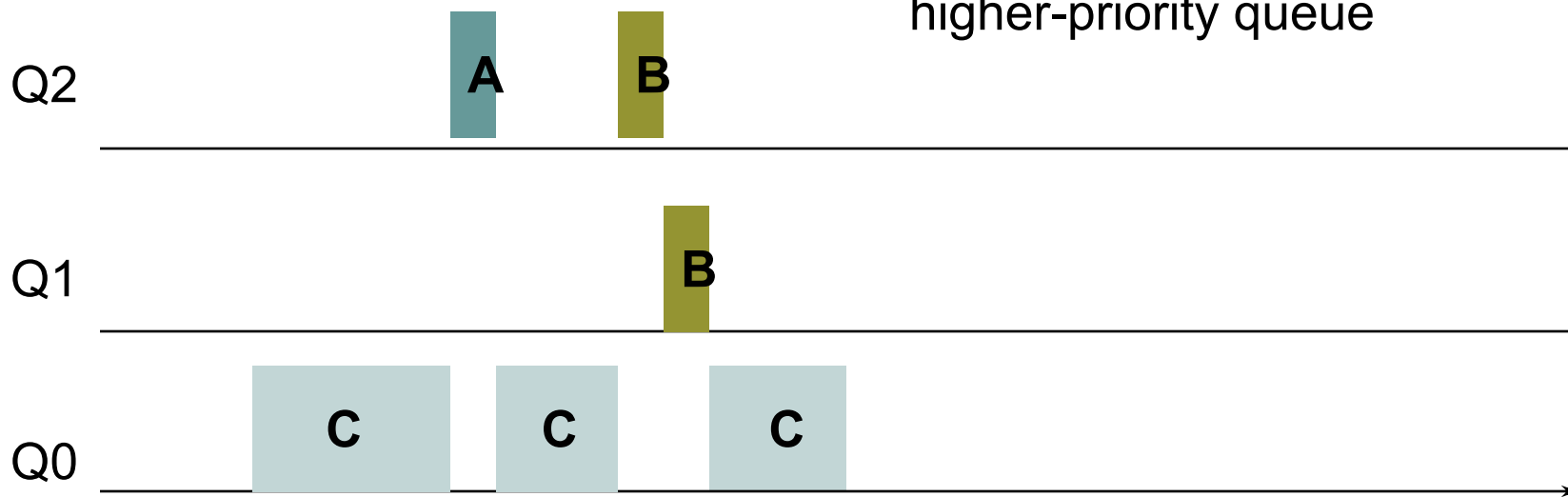
# MLFQ: Long job and short jobs in between



# MLFQ: Long job and short jobs in between



Leave I/O bound and interactive processes in higher-priority queue

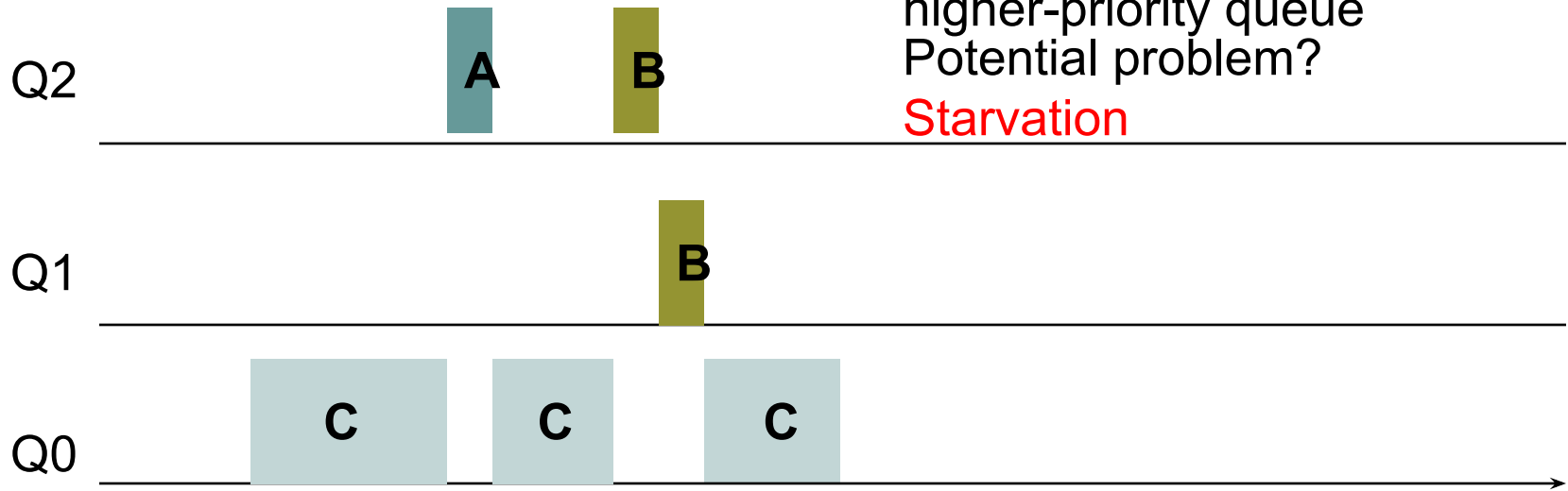


# MLFQ: Long job and short jobs in between



Leave I/O bound and interactive processes in higher-priority queue  
Potential problem?

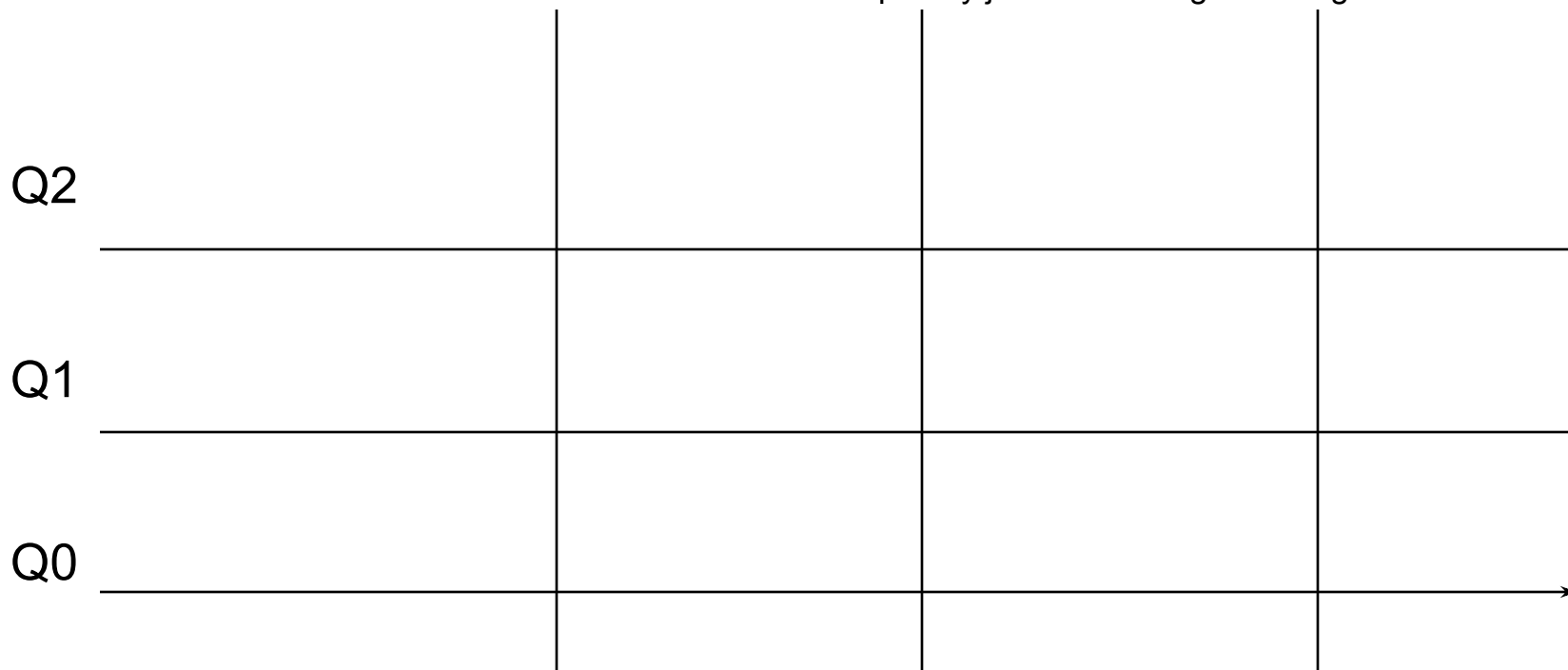
**Starvation**



# MLFQ: Long job and short jobs in between + boost



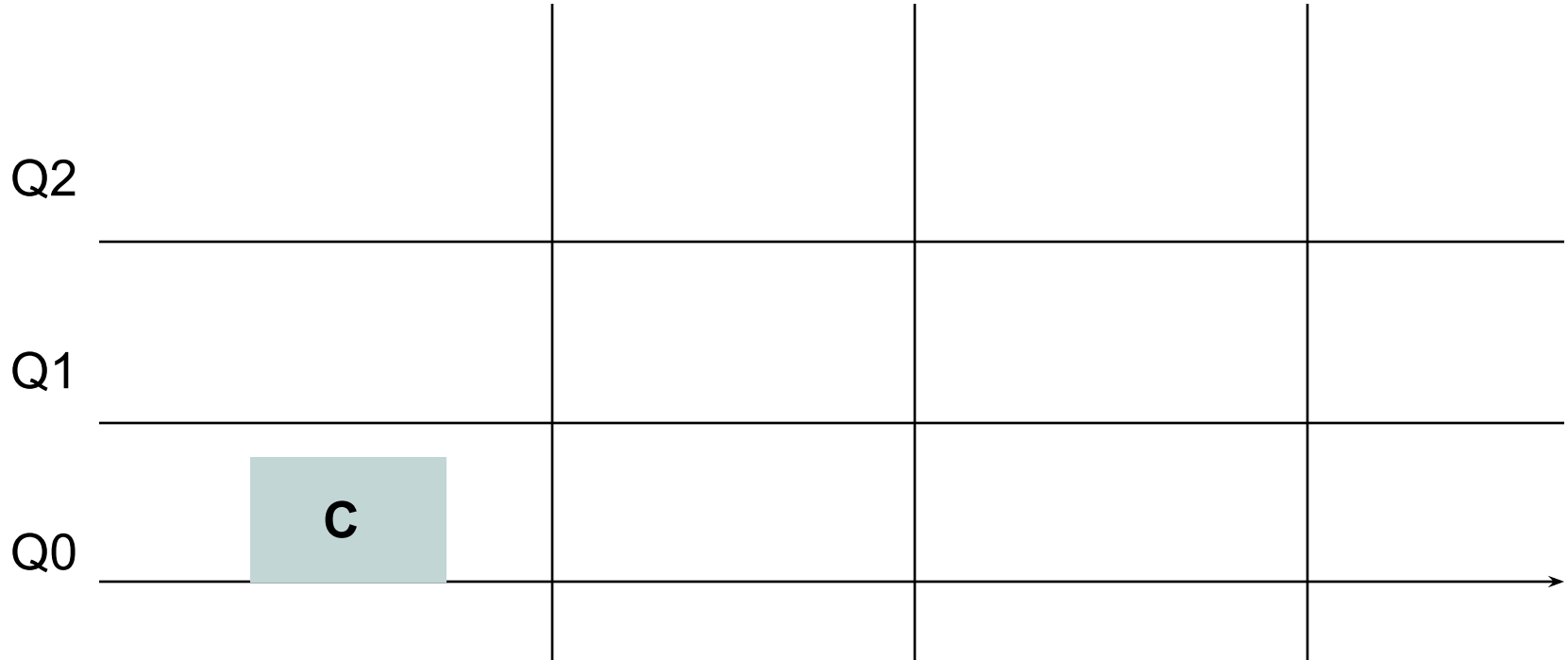
Boost Time : Low priority jobs moved higher at regular intervals



# MLFQ: Long job and short jobs in between + boost



Boost Time : Low priority jobs moved higher at regular intervals

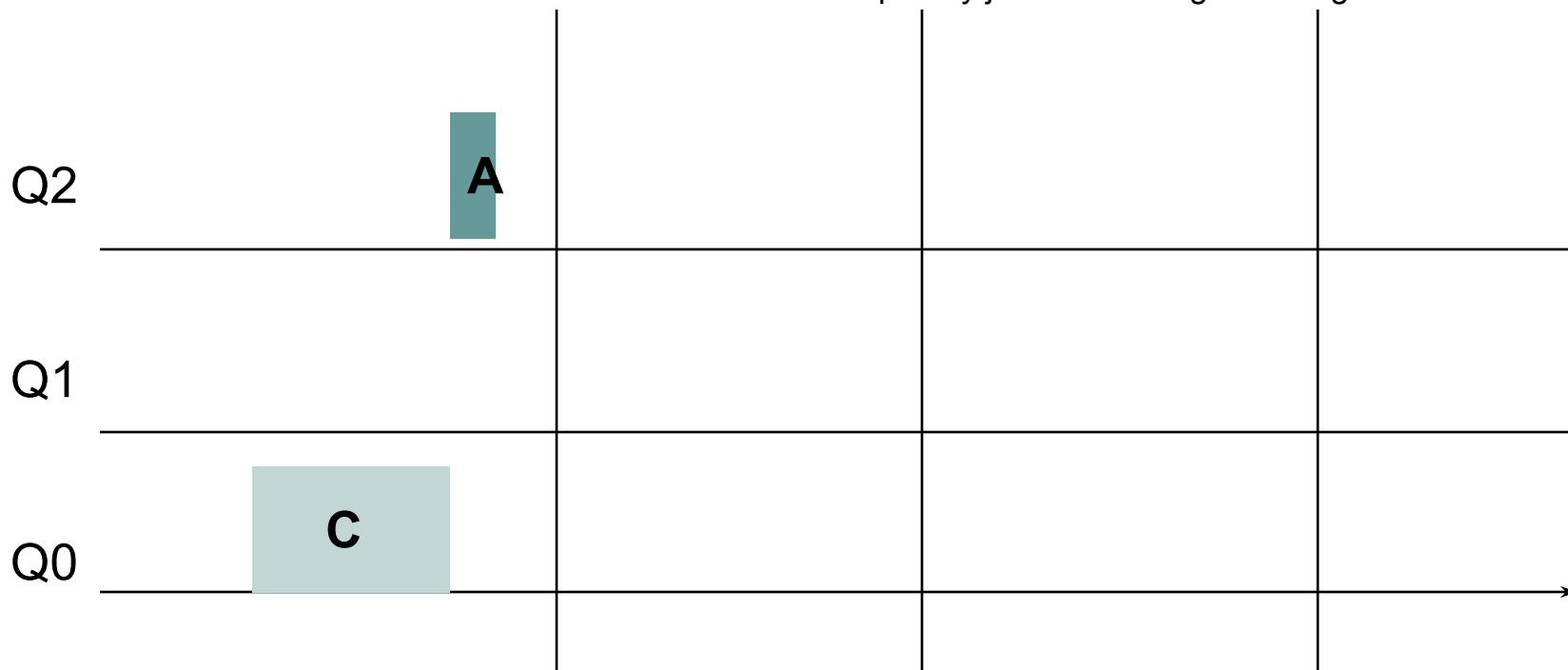




# MLFQ: Long job and short jobs in between + boost



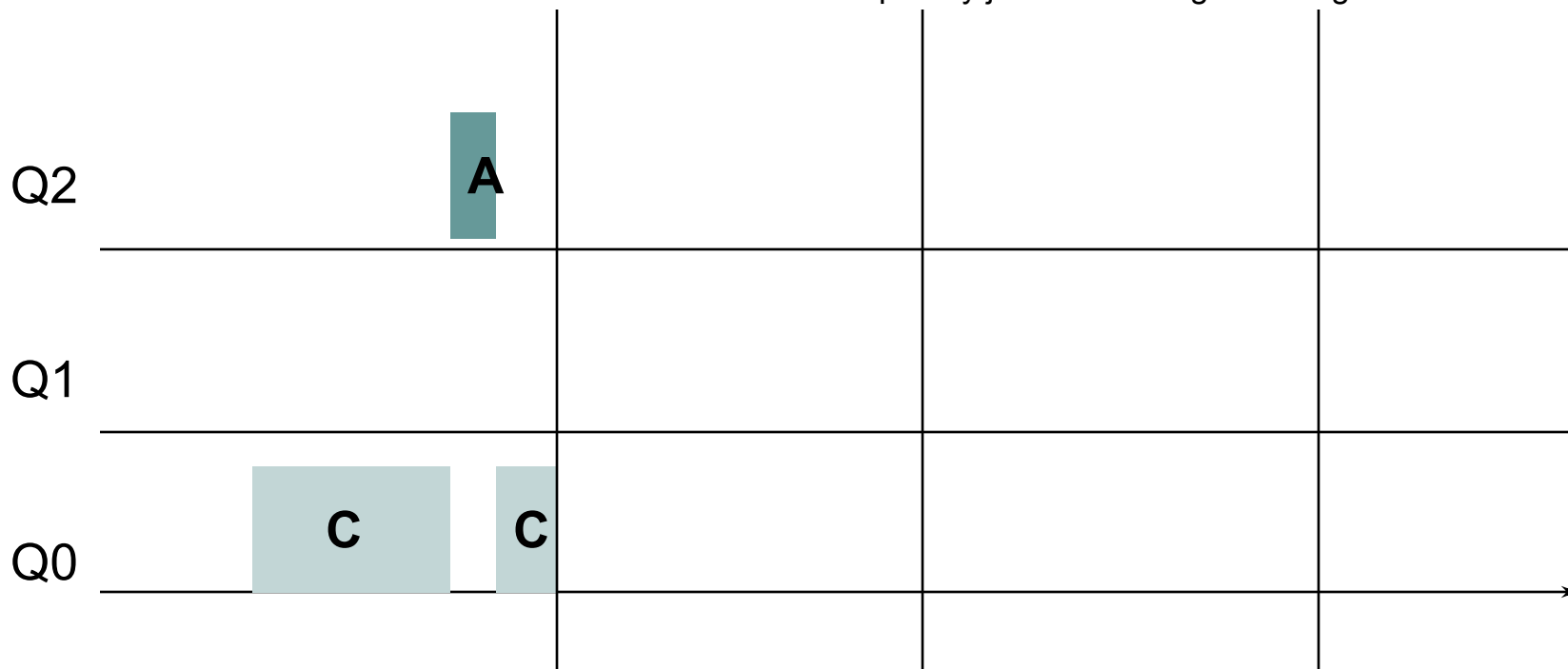
Boost Time : Low priority jobs moved higher at regular intervals



# MLFQ: Long job and short jobs in between + boost



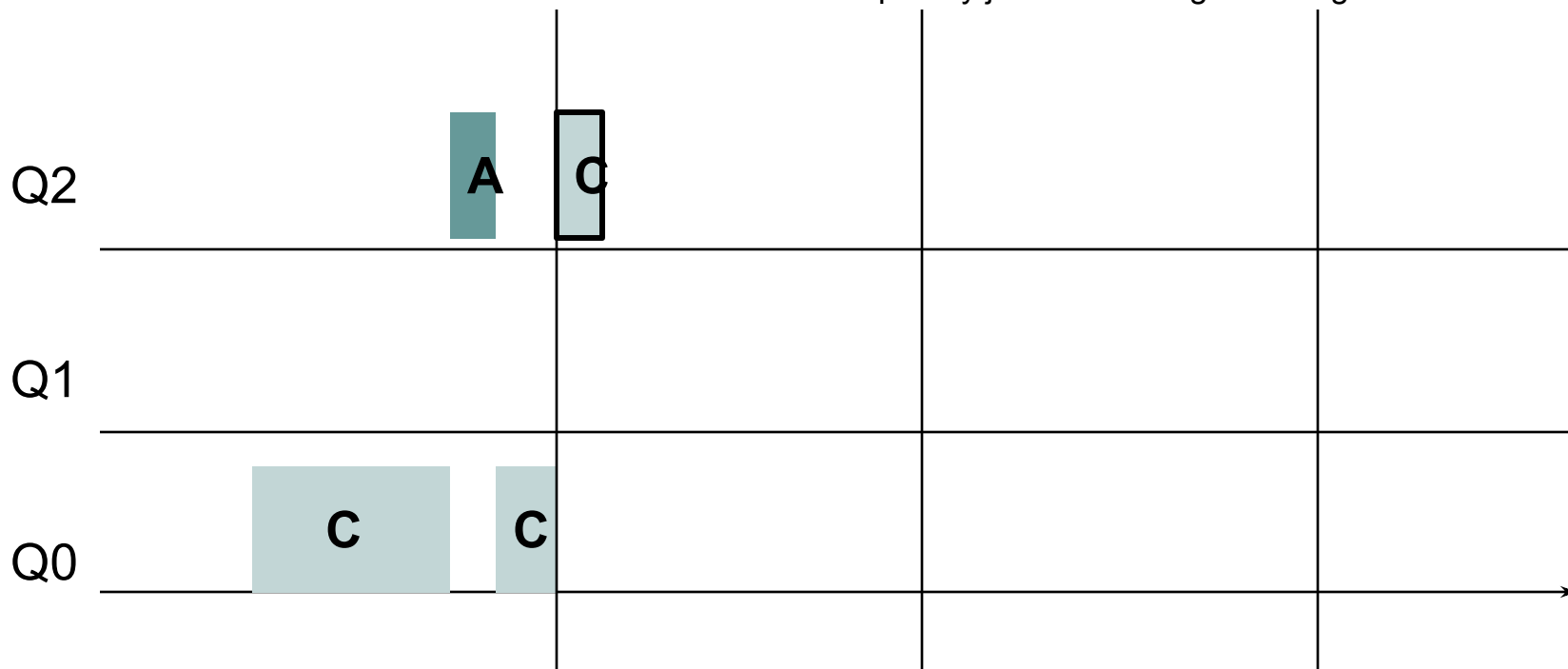
Boost Time : Low priority jobs moved higher at regular intervals



# MLFQ: Long job and short jobs in between + boost



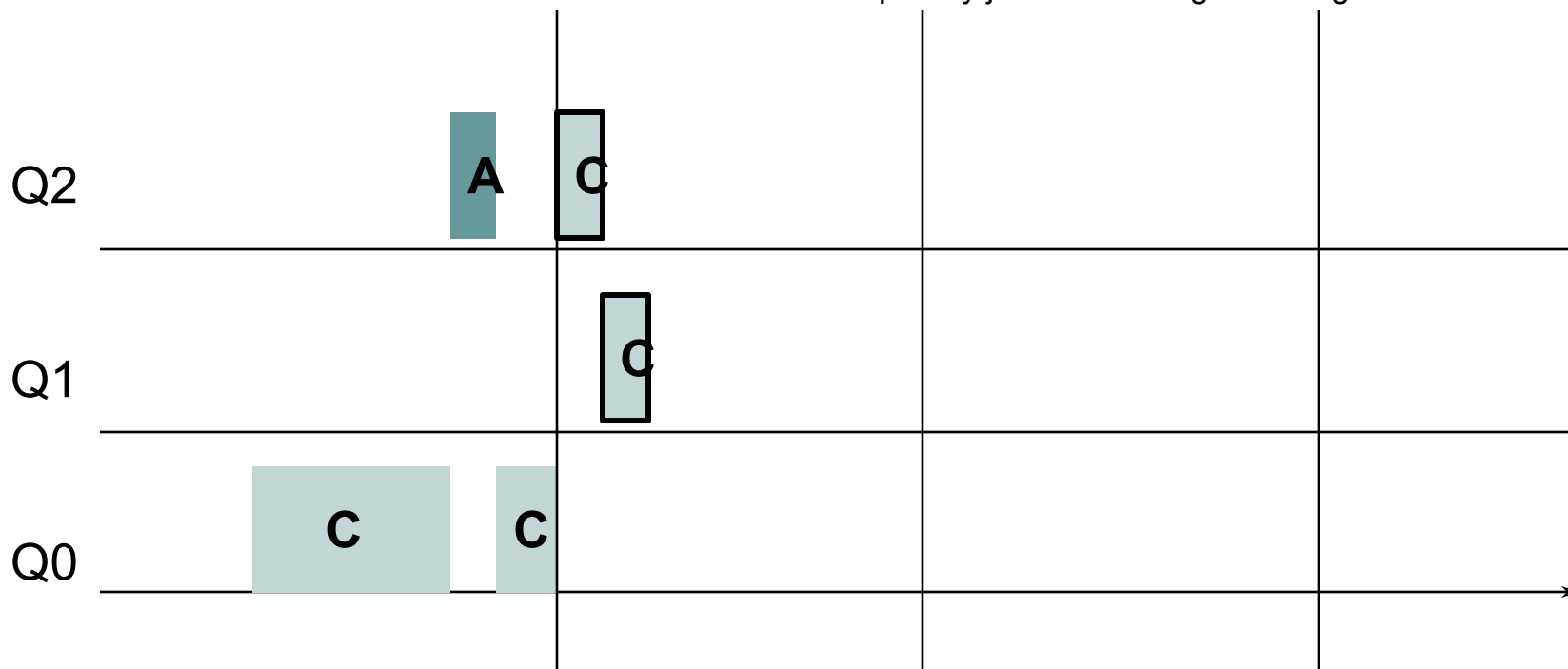
Boost Time : Low priority jobs moved higher at regular intervals



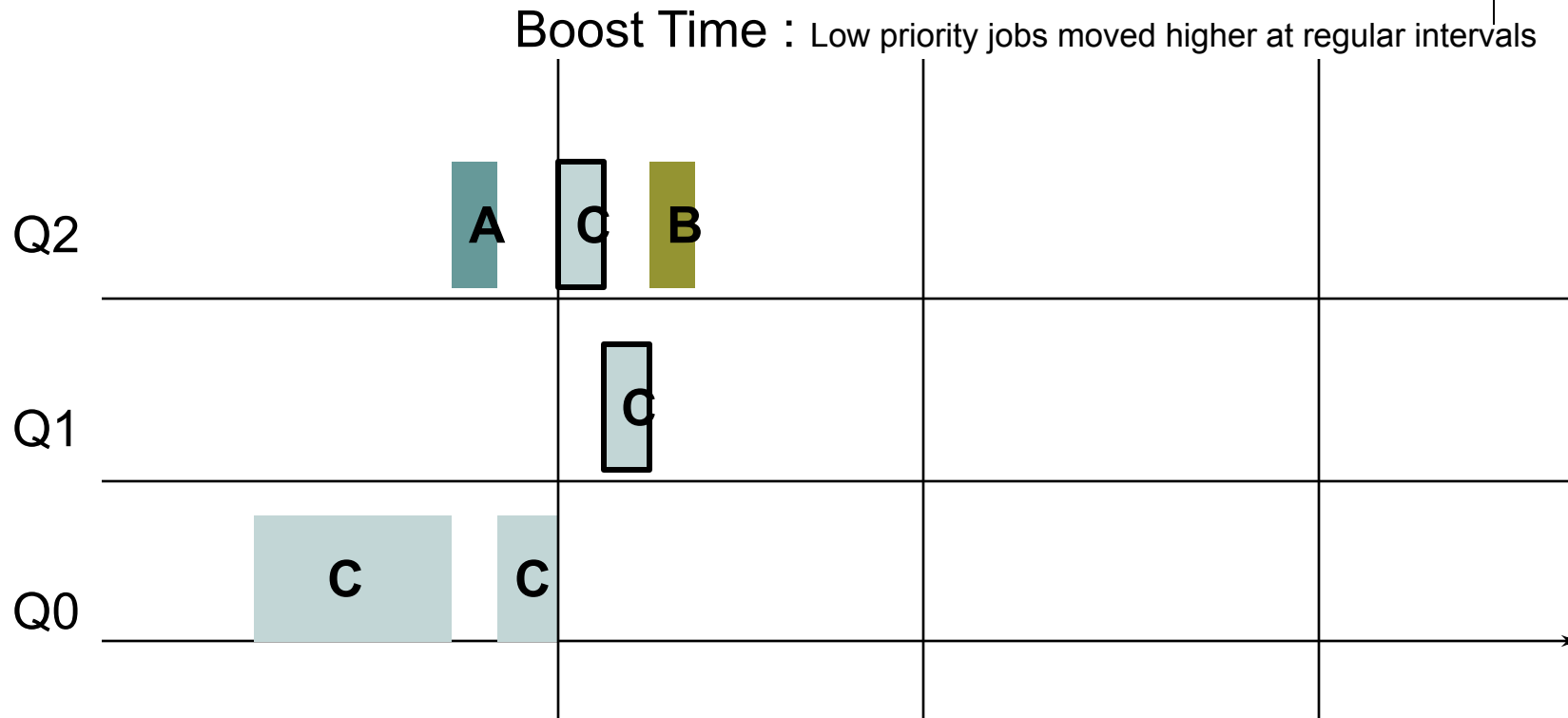
# MLFQ: Long job and short jobs in between + boost



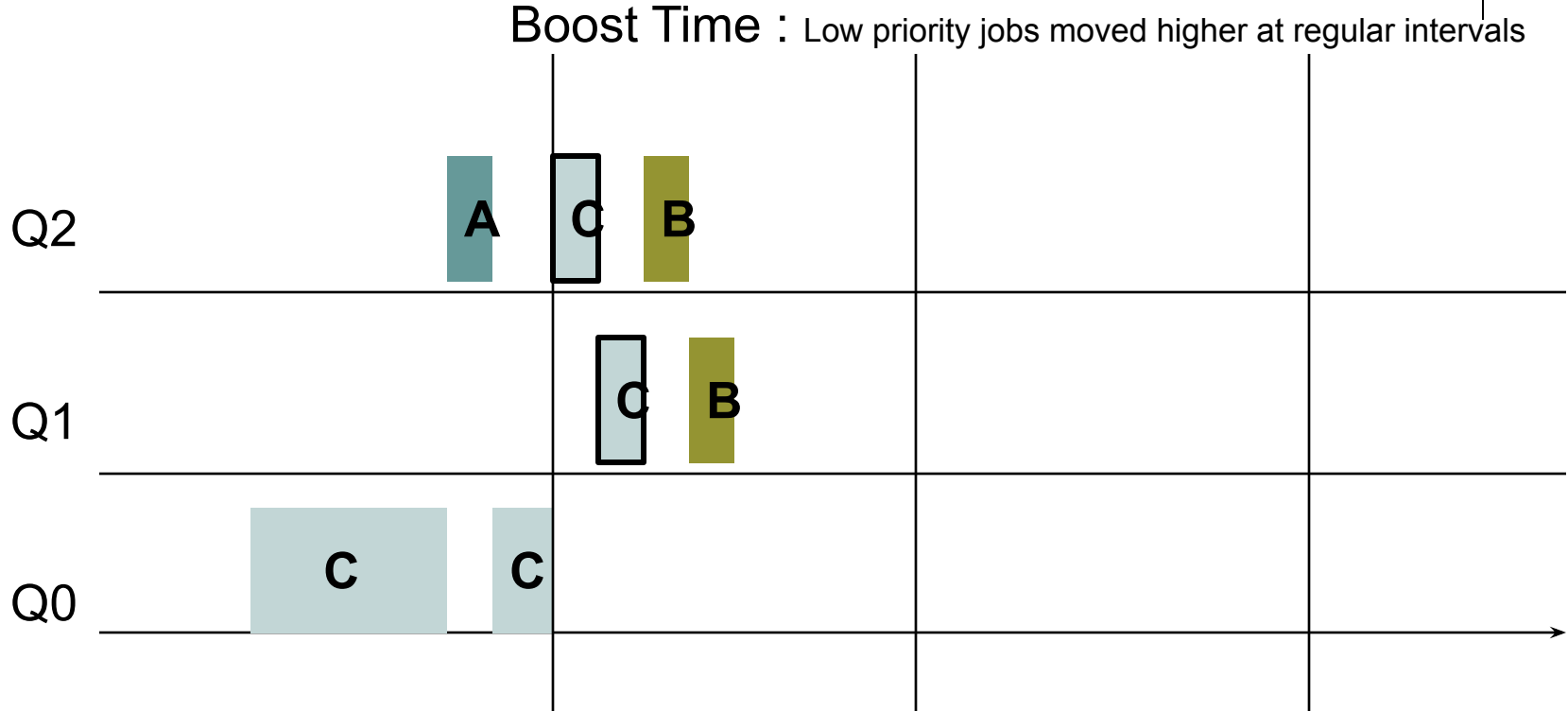
Boost Time : Low priority jobs moved higher at regular intervals



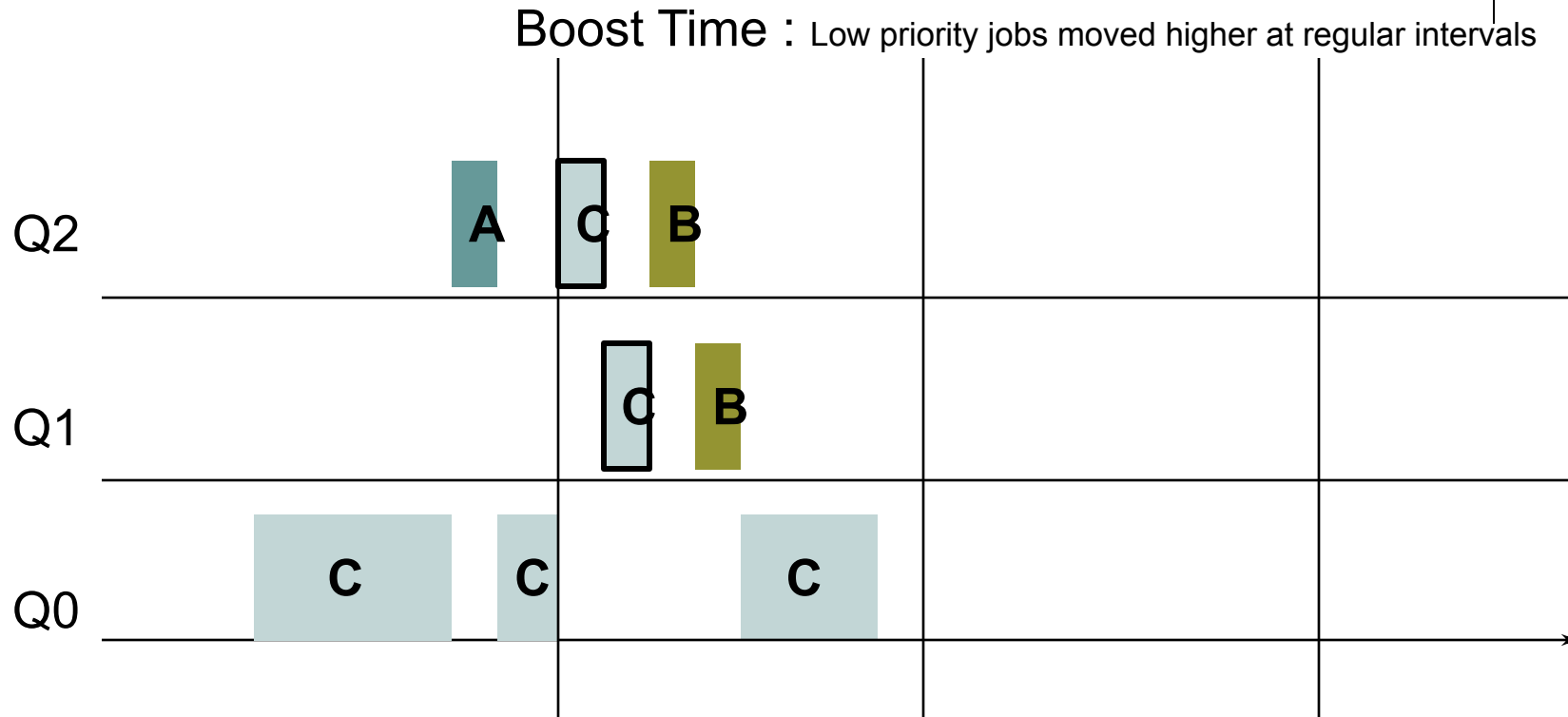
# MLFQ: Long job and short jobs in between + boost



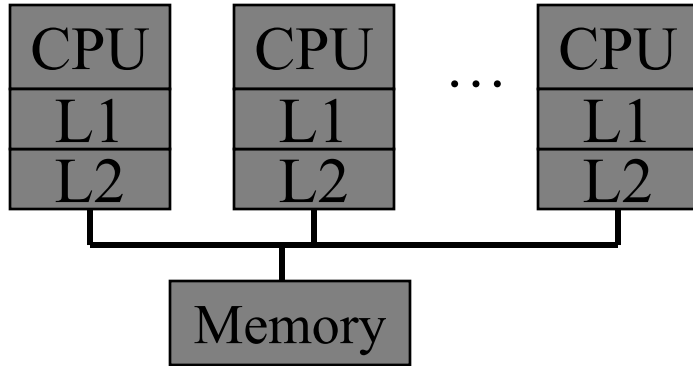
# MLFQ: Long job and short jobs in between + boost



# MLFQ: Long job and short jobs in between + boost

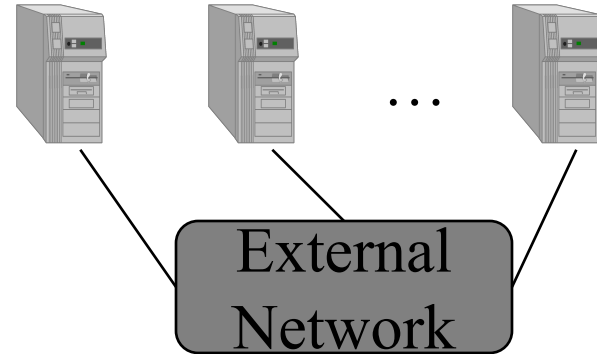


# Multiprocessor and Cluster



Multiprocessor architecture

- L2 cache coherence
- A single “image” OS



Cluster/Multicomputer

- Distributed memory
- An OS on each box



# Multiprocessor/Cluster Scheduling



- New design issue: process/thread inter-dependence
  - Threads of the same process may synchronize
  - Processes of the same job may send/recv messages

# Multiprocessor/Cluster Scheduling: Example approach

- Gang scheduling (coscheduling)



# Multiprocessor/Cluster Scheduling: Example approach



- Gang scheduling (coscheduling)
  - Related process and threads run at the same time.

# Multiprocessor/Cluster Scheduling: Example approach



- Gang scheduling (coscheduling)
  - Related process and threads run at the same time.
  - Threads of same process will run at same time on multiprocessor.

# Multiprocessor/Cluster Scheduling: Example approach



- Gang scheduling (coscheduling)
  - Related process and threads run at the same time.
  - Threads of same process will run at same time on multiprocessor.
  - Processes of same application run at the same time on cluster.

# Multiprocessor/Cluster Scheduling: Example approach



- Dedicated processor assignment
  - Threads will be running on specific processors to completion
- Pros / cons?
  - Good for reducing cache misses
  - Bad for load balance / fairness

# CPU scheduling: Final Thoughts



- Mechanism is easy, policy is hard
  - Jobs have diverse characteristics
  - 4 performance metrics
  - Don't know about future
- Hard to analyze even when narrowing down metric/job nature

# True/False



- “A CPU scheduling algorithm that minimizes avg turnaround time cannot lead to starvation.”
- “Among all CPU scheduling algorithms, Round Robin always gives the worse average turnaround time.”



# Scheduling Algorithms in OSeS



Operating System	Preemption	Algorithm
<u>Windows 3.1x</u>	None	<u>Cooperative Scheduler</u>
<u>Windows 95</u> , <u>98</u> , <u>Windows 95, 98, Me</u>	Half	Preemptive for 32-bit processes, <u>Cooperative Scheduler</u> for 16-bit processes
<u>Windows NT</u> (2000, XP, Vista, 7, and Server)	Yes	<u>Multilevel feedback queue</u>
<u>Mac OS pre-9</u>	None	<u>Cooperative Scheduler</u>
<u>Mac OS 9</u>	Some	Preemptive for MP tasks, <u>Cooperative Scheduler</u> for processes and threads
<u>Mac OS X</u>	Yes	<u>Multilevel feedback queue</u>
<u>Linux pre-2.6</u>	Yes	<u>Multilevel feedback queue</u>
<u>Linux 2.6-2.6.23</u>	Yes	<u>O(1) scheduler</u>
<u>Linux post-2.6.23</u>	Yes	<u>Completely Fair Scheduler</u>
<u>Solaris</u>	Yes	<u>Multilevel feedback queue</u>
<u>NetBSD</u>	Yes	<u>Multilevel feedback queue</u>
<u>FreeBSD</u>	Yes	<u>Multilevel feedback queue</u>

# Process



- A process contains everything needed for execution
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC (program counter), SP (stack pointer), regs, etc.)

# Web Server Example



- How does a web server handle 1 request?
- A web server needs to handle many concurrent requests
- Solution 1:
  - Have the parent process fork as many processes as needed
  - Processes communicate with each other via inter-process communication

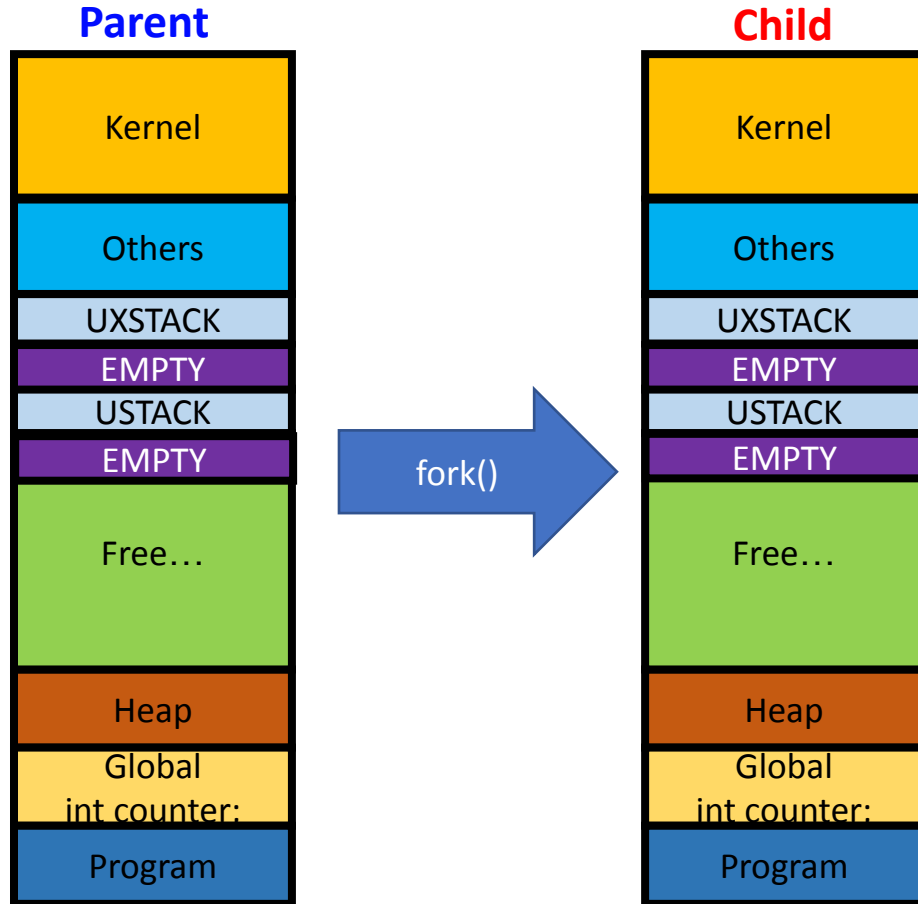
# Multiple Processes



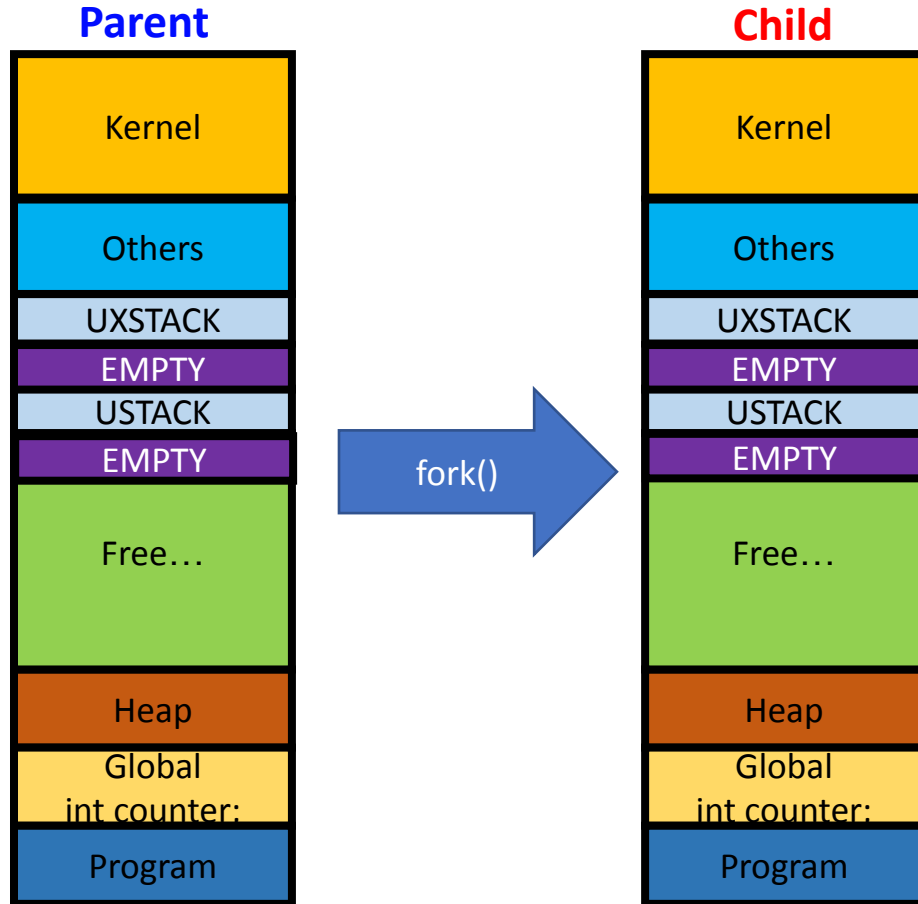
Parent



# Multiple Processes



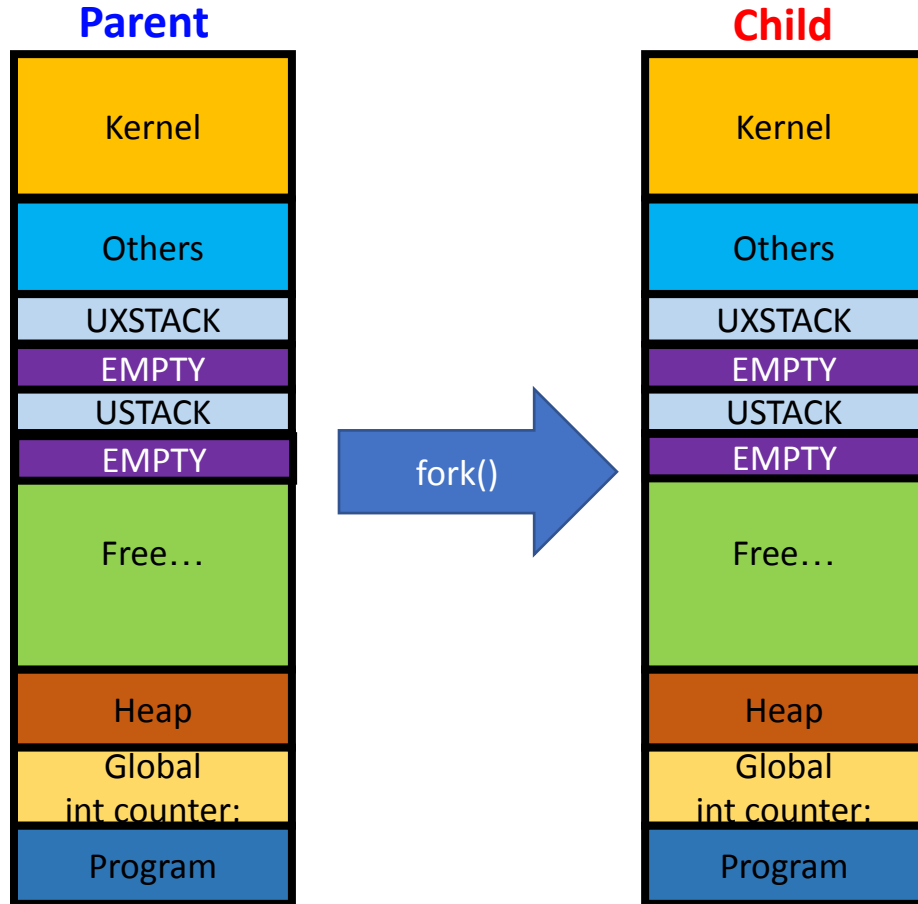
# Multiple Processes



Fork() creates new process by copying memory space

Process creates a new PRIVATE memory space

# Multiple Processes



Fork() creates new process by copying memory space

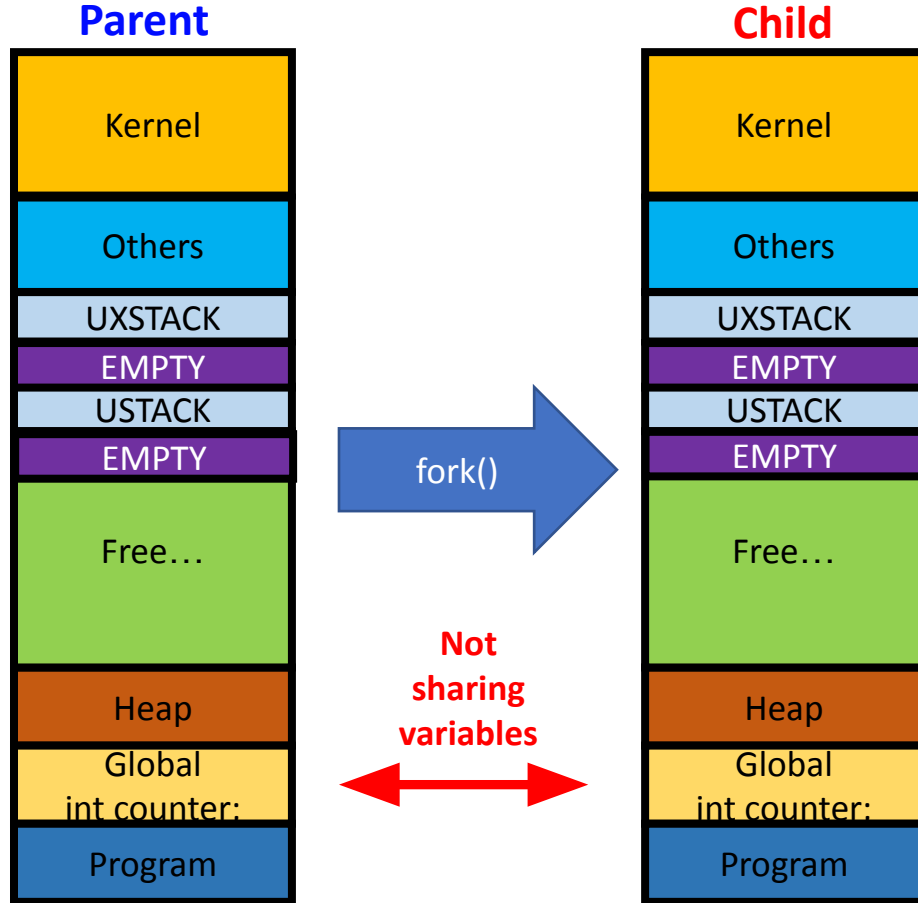
```
Pr #include <stdio.h>
#include <unistd.h>

int counter;
volatile int value = 1;

void countup() {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
}

int main() {
    pid_t pid = fork();
    countup();
    printf("%s: %d\n", pid ? "Parent" : "Child", counter);
}
```

# Multiple Processes



Fork() creates new process by copying memory space

```
Pr #include <stdio.h>
#include <unistd.h>

int counter;
volatile int value = 1;

void countup() {
    for(int i=0; i<1000000; ++i) {
        counter += value;
    }
}

int main() {
    pid_t pid = fork();
    countup();
    printf("%s: %d\n", pid ? "Parent" : "Child", counter);
}
```

**Parent: 1000000**  
**Child: 1000000**



# All these processes may need to communicate

- How can two processes communicate?



# How do Process communicate?

- At process creation time
  - Parents get one chance to pass everything at fork()



# All these processes may need to communicate

- How can two processes communicate?
  - Different machine?



# All these processes may need to communicate



- How can two processes communicate?
  - Different machine?
    - Over network

# All these processes may need to communicate



- How can two processes communicate?
  - Different machine?
    - Over network
  - Same machine?

# All these processes may need to communicate



- How can two processes communicate?
  - Different machine?
    - Over network
  - Same machine?
    - Need to have some common resource:

# All these processes may need to communicate



- How can two processes communicate?
  - Different machine?
    - Over network
  - Same machine?
    - Need to have some common resource:
      - Files (pipes)

# All these processes may need to communicate



- How can two processes communicate?
  - Different machine?
    - Over network
  - Same machine?
    - Need to have some common resource:
      - Files (pipes)
      - Shared Memory



# All these processes may need to communicate



- How can two processes communicate?
  - Different machine?
    - Over network
  - Same machine?
    - Need to have some common resource:
      - Files (pipes)
      - Shared Memory
      - Message boxes

# All these processes may need to communicate



- How can two processes communicate?
  - Different machine?
    - Over network
  - Same machine?
    - Need to have some common resource:

- Files (pipes)
- Shared Memory
- Message boxes

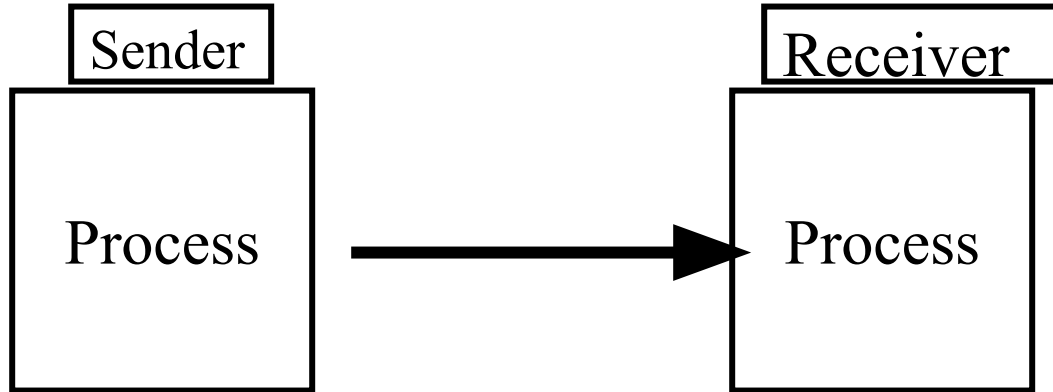
**Inter process communication  
(IPC)**

# IPC with Messages and MailBoxes



- Messages provide for communication **without shared data**
  - One process or the other owns the data, (guaranteed) never two at the same time
  - Think about USPS

# IPC with Messages and MailBoxes



# Why Messages?



- Many types of applications fit into the model of processing a sequential flow of information
- Communication across address spaces – no side effects
  - Less error-prone? (commonly believed, but **not necessarily true!**)
  - They might have been written by different programmers who aren't familiar with code
  - They might not trust each other
  - They may be running on different machines!
  - Examples?

# Message Passing API



- Generic API
  - `send( mailbox, msg )`
  - `recv( mailbox, msg )`
- What is a mailbox?
  - A *buffer* where messages are stored between the time they are sent and the time when they are received
- What should “msg” be?
  - Fixed size msgs
  - Variable sized msgs: need to specify sizes

# Implementation Options with Buffering



- When should `send()` return?
- When should `recv()` return?

# Send

- *Fully Synchronous*
  - Will not return until **data is received** by the receiving process





# Send



- *Fully Synchronous*
  - Will not return until **data is received** by the receiving process
  
- *Synchronous*
  - Will not return until data is **received by the mailbox**
  - Block on full buffer

# Send



- *Fully Synchronous*
  - Will not return until **data is received** by the receiving process
- *Synchronous*
  - Will not return until data is **received by the mailbox**
  - Block on full buffer
- *Asynchronous*
  - **Return immediately**
  - Completion
    - Require the application to check status (application polls)
    - Notify the application (OS sends interrupt)

# Receive



- *Synchronous*
  - Return data if there is a message
  - **Block on empty buffer**

# Receive



- *Synchronous*
  - Return data if there is a message
  - **Block on empty buffer**
  
- *Asynchronous*
  - Return data if there is a message
  - Return **status if there is no message** (probe)

# Implementing Mailboxes



- What is the conceptual problem for OS implementation here?
  - Assume sender and receiver are on the same machine

# Implementing Mailboxes



- What is the conceptual problem for OS implementation here?
  - Assume sender and receiver are on the same machine:
    - Buffering
    - Managing mailboxes:
      - between a pair or processes?
      - One for each process?
    - Process Termination

# Buffering



- No buffering
  - Sender must wait until the receiver receives the message
  - Rendezvous on each message
- Bounded buffer
  - Finite size
  - Sender blocks when the buffer is full
  - Receiver blocks when the buffer is empty
  - Using lock/condition variable (or semaphore)

# Direct Communication



- Each process must name the sending or receiving process
- A communication link
  - is set up between the pair of processes
  - is associated with exactly two processes
  - exactly one link between each pair of processes

P: send( process Q, msg )

Q: recv( process P, msg )



# Producer-Consumer with Message passing



```
Producer(){  
  while (1) {  
    ...  
    produce item  
    ...  
    send( consumer, item);  
  }  
}
```

```
Consumer(){  
  while (1) {  
    recv( producer, item );  
    ...  
    consume item  
    ...  
  }  
}
```

# Indirect Communication

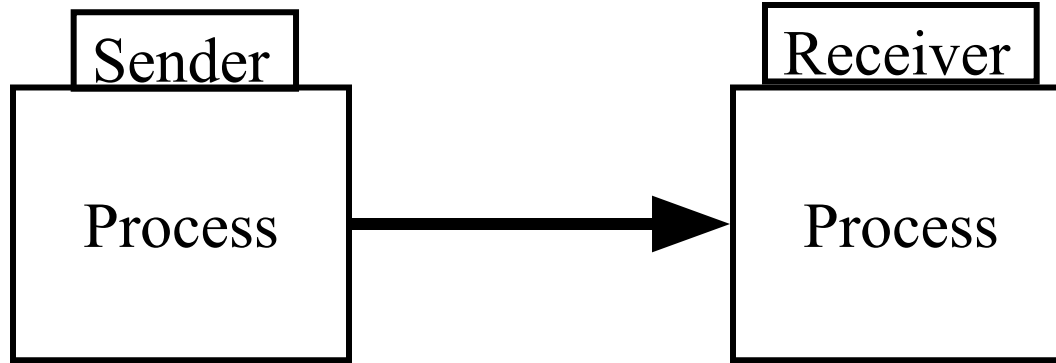


- Use a “mailbox” or “ports” to allow **many-to-many communication**
  - Mailbox typically owned by the OS
    - Requires open/close a mailbox before allowed to use it
- A “link”
  - is set up among processes only if they have a shared mailbox
  - Can be associated with more than two processes

P: open (mailbox); send( mailbox, msg);  
close(mailbox)

Q: open (mailbox); recv( mailbox, msg );  
close(mailbox)

# Process Termination



- S has terminated
  - Problem: R may be blocked forever
  - Solution: R pings S once a while
- R has terminated
  - Problem: S runs out buffer and will be blocked forever
  - Solution: S checks on R occasionally

# IPC Big Debate: Message passing v/s Shared memory



- Two programming models are equally powerful
- But result in very different-looking programming styles
- Do you think shared-data or message-passing is easier to work with?
  - Programming? Debugging?
- What about concurrent programming across machines?
  - Message passing is more natural
  - Shared memory can still be simulated in software
    - Distributed Shared Memory (DSM) – hot topic in 80-90's, coming back these years