Week 2

Introduction to Process Management

Max Kopinsky

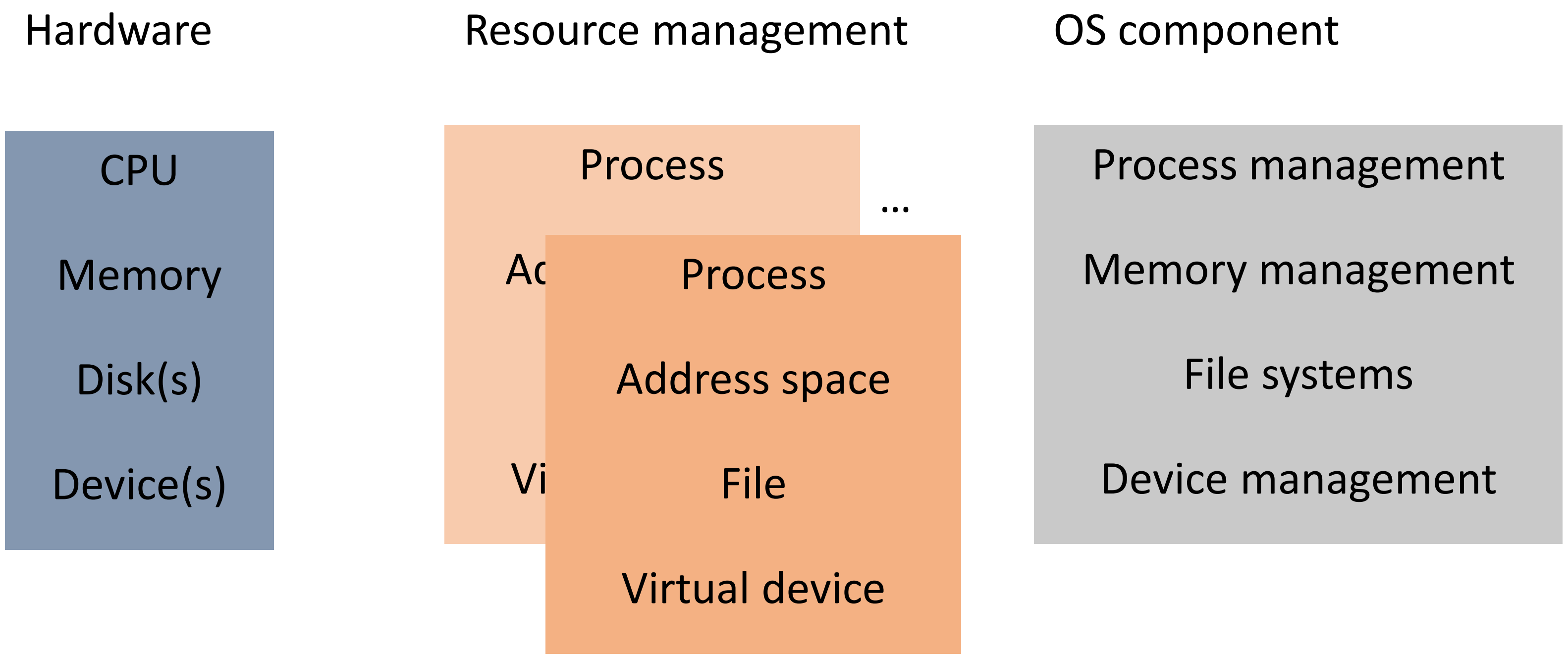
January 14, 2025

Recap from Week 1

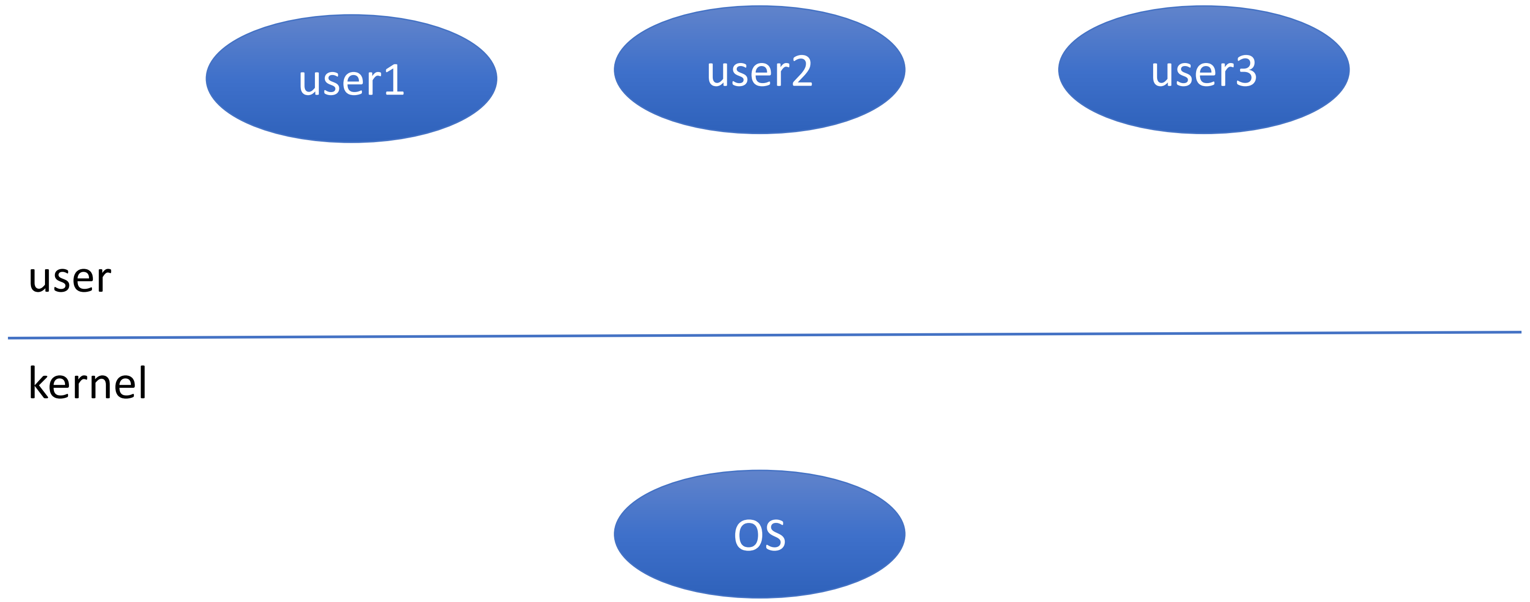
- What does the OS do?
- Where does the OS live?
- OS interfaces
- OS control flow
- OS structure

Recap from Week 1:
What does the OS do?

- **Abstraction and Resource management**



Recap from Week 1: User/OS Separation



Recap from Week 1:
Kernel mode vs User mode

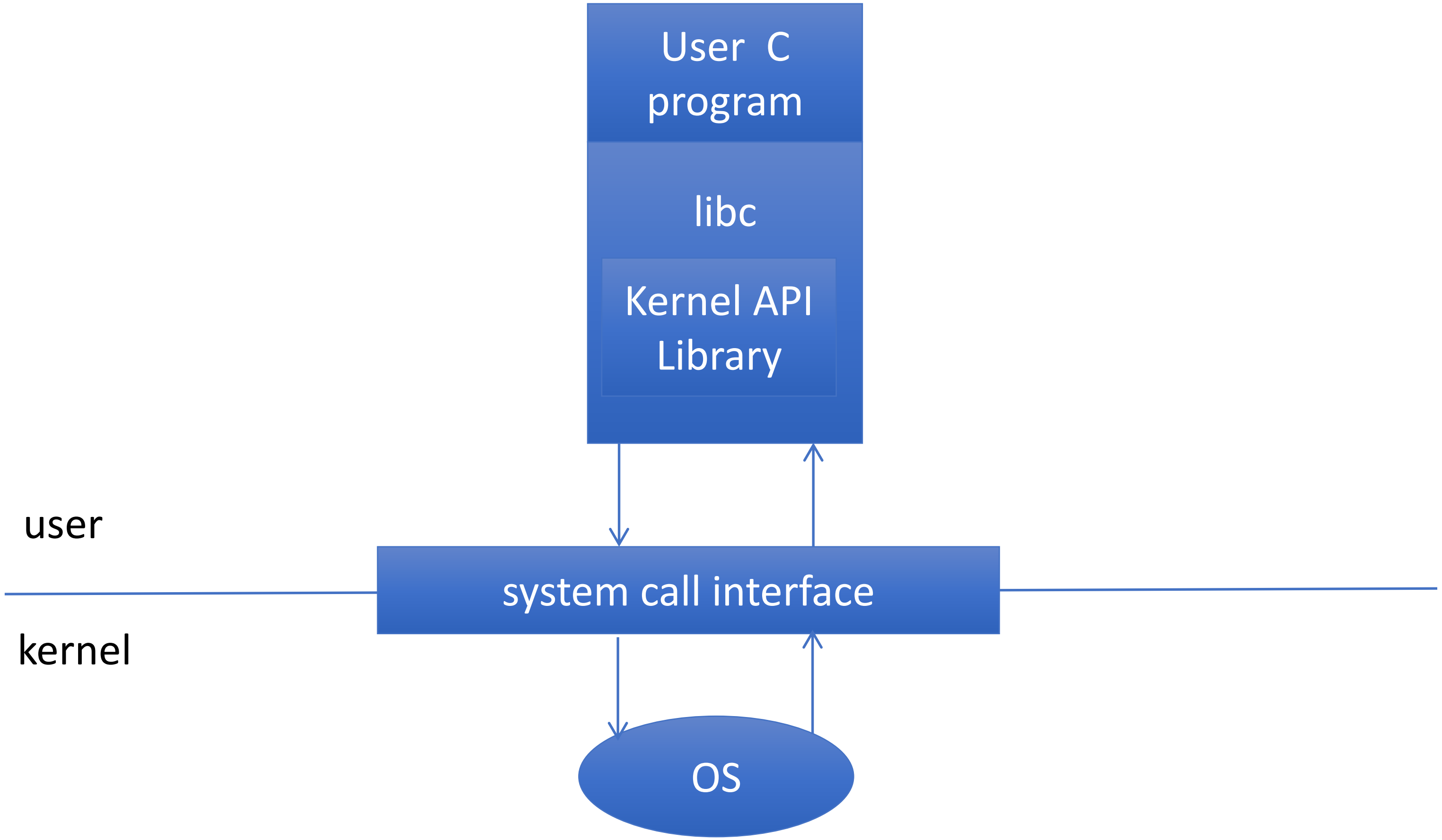
Kernel Mode

- Privileged instructions:
  - Set mode bit
  - …
- Direct access to all of memory
- Direct access to devices

User Mode

- No privileged instructions:
  - Set mode bit
  - …
- No direct access to all of memory
- No direct access to devices

Recap from Week 1: System calls, kernel API, libc



Recap from Week 1
System Calls, Traps Interrupts

- System calls
- Traps
- Interrupts

Recap from Week 1
System Calls, Traps Interrupts

- System calls
  - Are the *only* interface from program to OS
  - Narrow interface essential for integrity of OS
- Traps
  - Trap is generated by CPU as a result of error
  - Works like an “involuntary” system call
- Interrupts
  - Generated by a device that needs attention

Recap from Week 1
OS Control Flow: Event-Driven Program

- Nothing to do

Do nothing

- Interrupt (from device)
- Trap (from process)
- System call (from process}

Start running

Key Concepts for this week

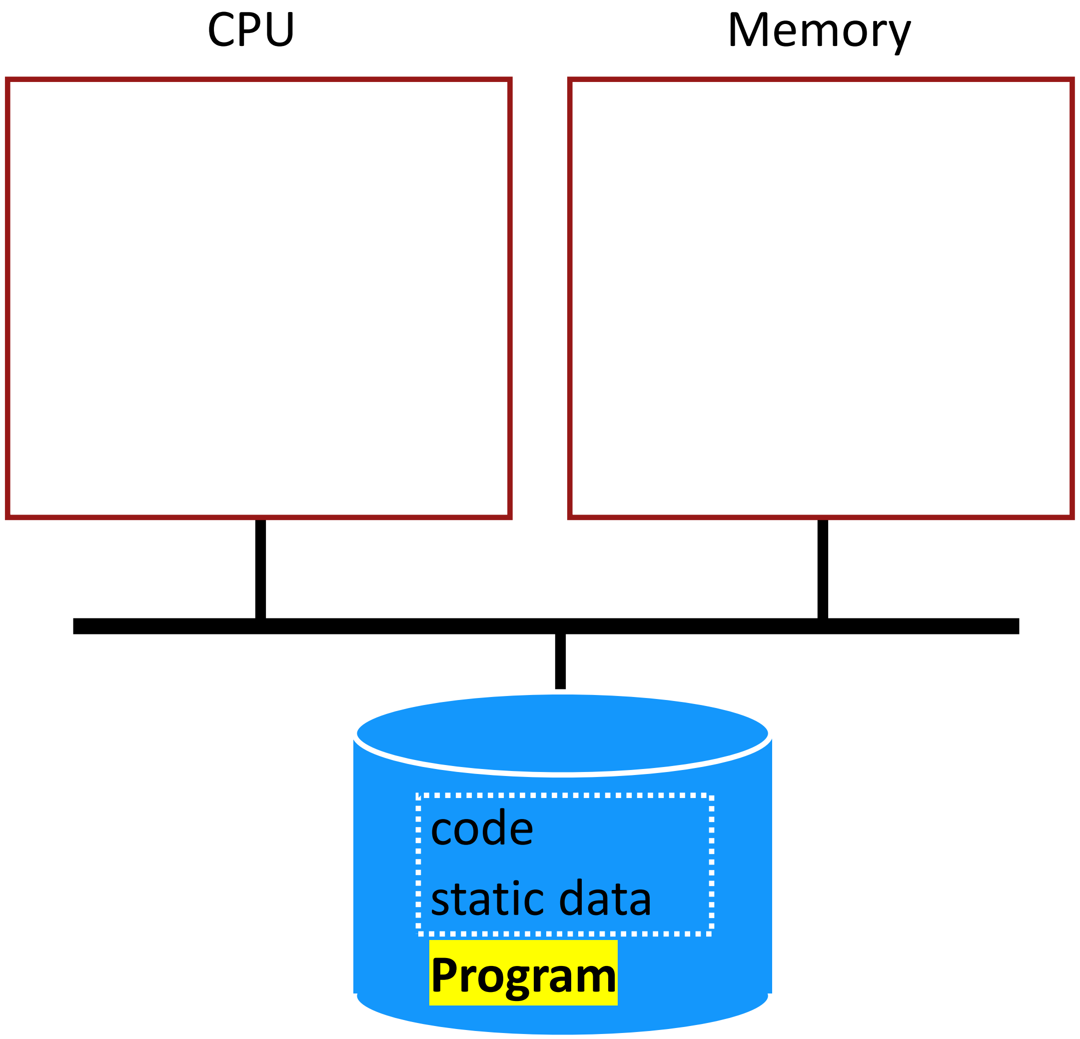
- Process
- Linux process tree
- Process switch
- Process scheduler

Process vs Program

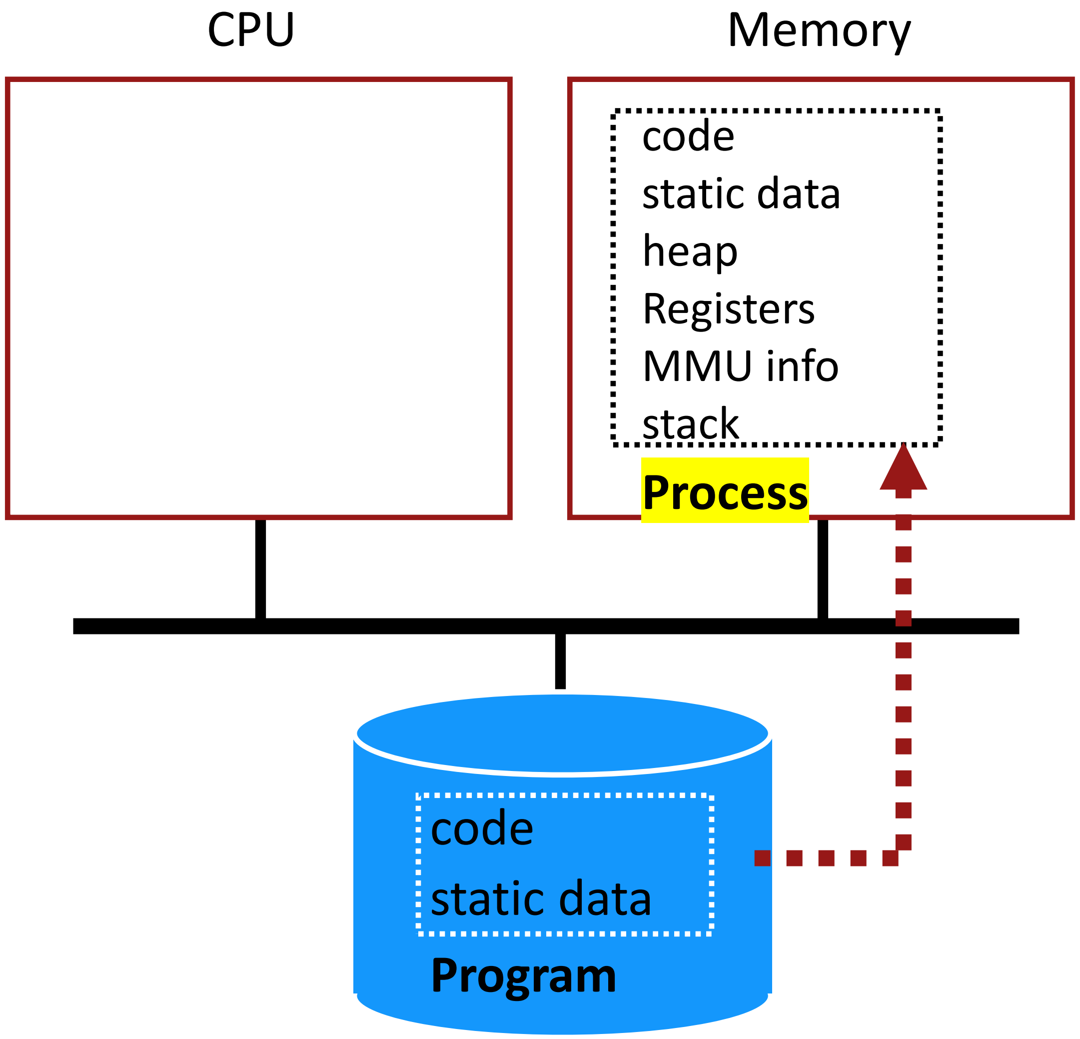
What is a Process?

- **Process = program in execution**
- Program
  - Executable code
  - Usually represented by a file on disk
- Process
  - Executing code
  - Usually represented in memory

Process Creation



Process Creation



What is a Process?

- Process: An execution stream in the context of a process state

What is a Process?

- Process: An execution stream in the context of a process state
- What is an execution stream?
  - Stream of executing instructions
  - Running piece of code
  - “thread of control”

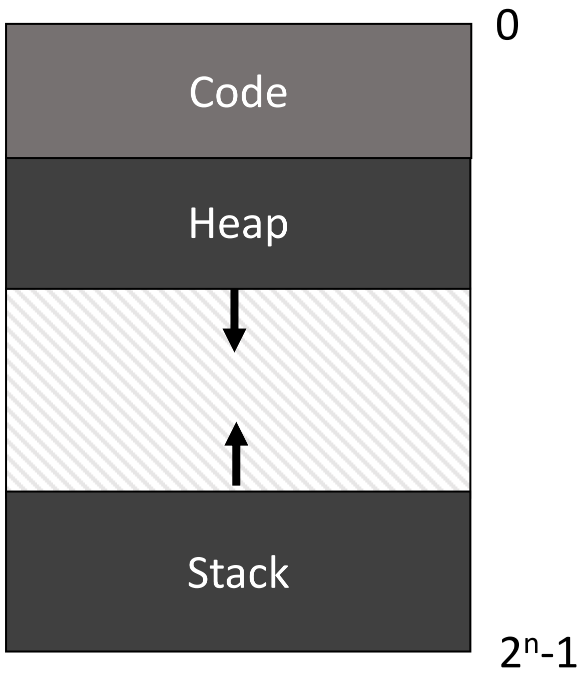
What is a Process?

- Process: An execution stream in the context of a process state
- What is process state?
  - Everything that the running code can affect or be affected by
  - Registers
    - General purpose, floating point, status, program counter, stack pointer
  - Address space
  - Heap, stack, and code
  - Open files

Address Space Review

Static and dynamic components

- Static: Code and some global variables
- Dynamic: Stack and Heap



Motivation for Dynamic Memory Allocation

Why do processes need dynamic allocation of memory?

- Do not know amount of memory needed at compile time.
- Must be pessimistic for static memory allocation.
- If statically allocate enough for worst possible case, storage is used inefficiently.

Motivation for Dynamic Memory Allocation

Why do processes need dynamic allocation of memory?

- Recursive procedures
  - Do not know how many times procedure will be nested
- Complex data structures: lists and trees
  - `struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));`

Dynamic Memory Allocation

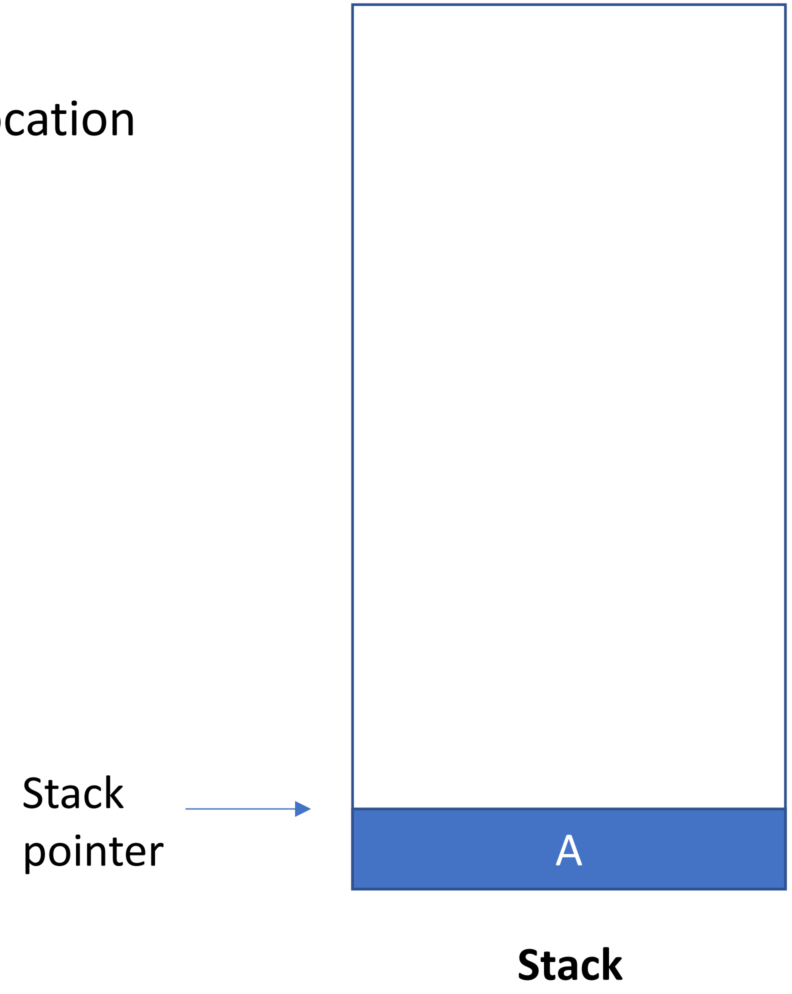
Two types of dynamic allocation

- Stack
- Heap

Stack

Memory is freed in opposite order from allocation

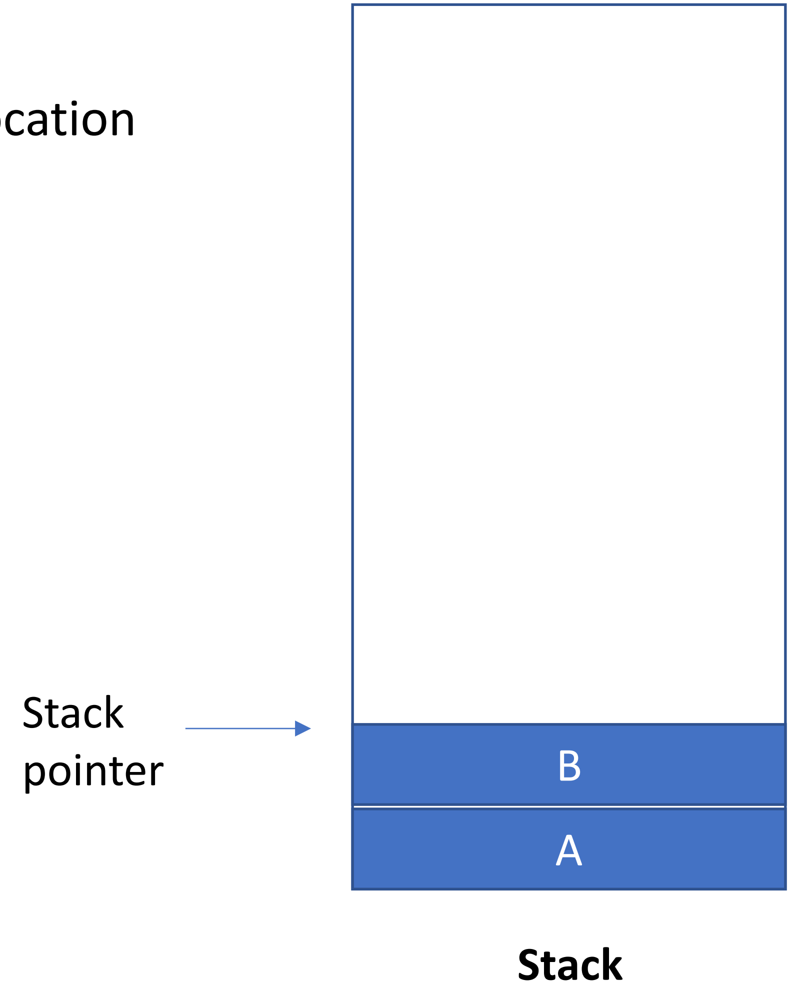
```
alloc(A);
```



Stack

Memory is freed in opposite order from allocation

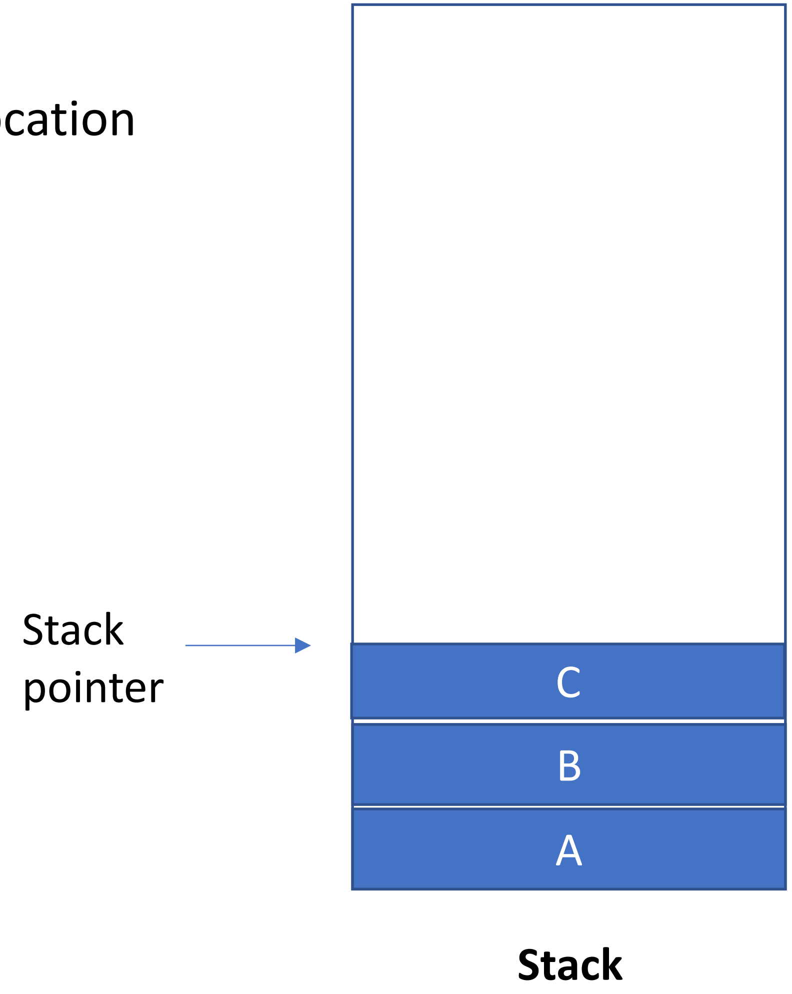
```
alloc(A);
alloc(B);
```



Stack

Memory is freed in opposite order from allocation

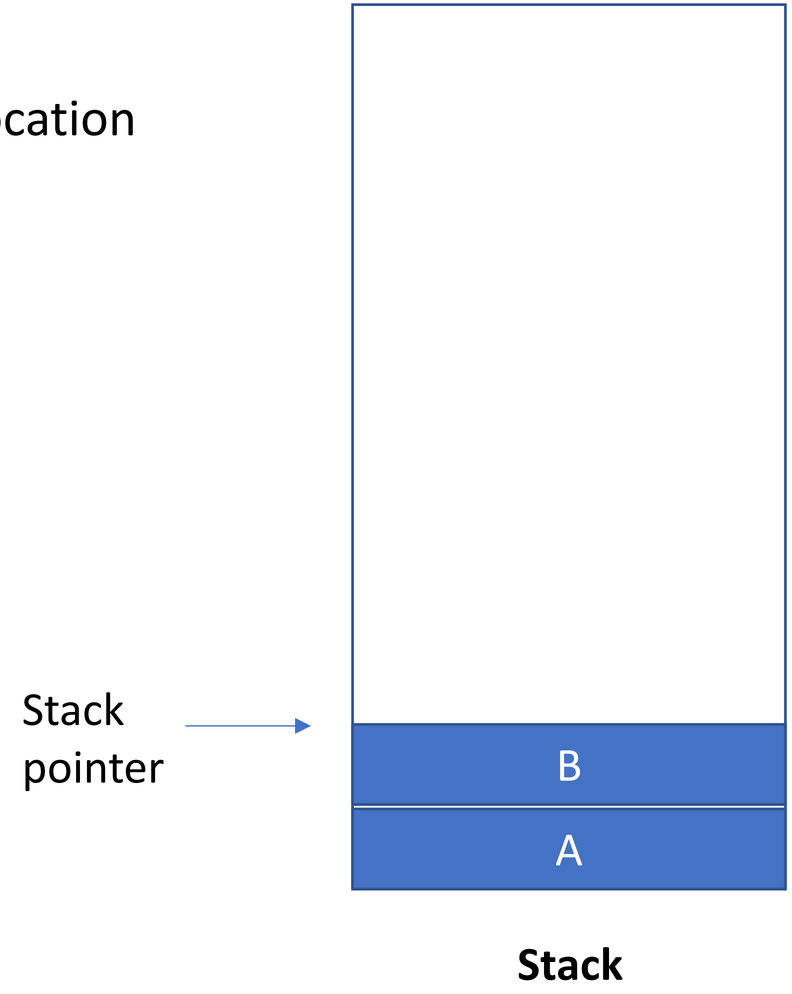
```
alloc(A);
alloc(B);
alloc(C);
```



Stack

Memory is freed in opposite order from allocation

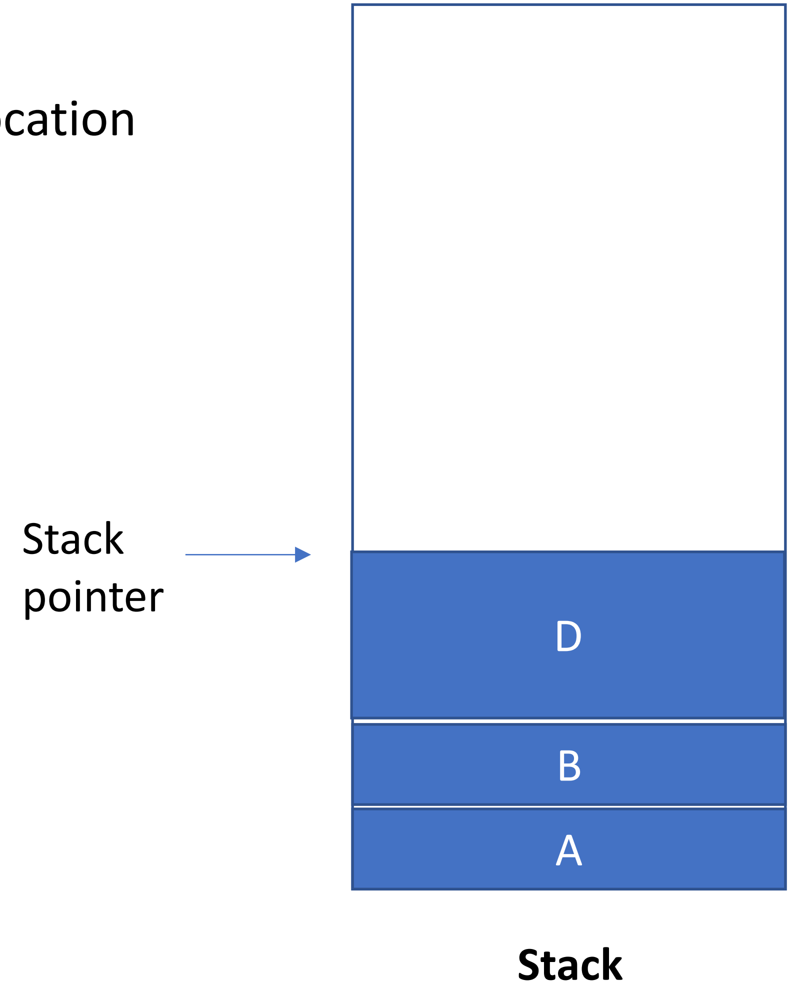
```
alloc(A);
alloc(B);
alloc(C);
free(C);
```



Stack

Memory is freed in opposite order from allocation

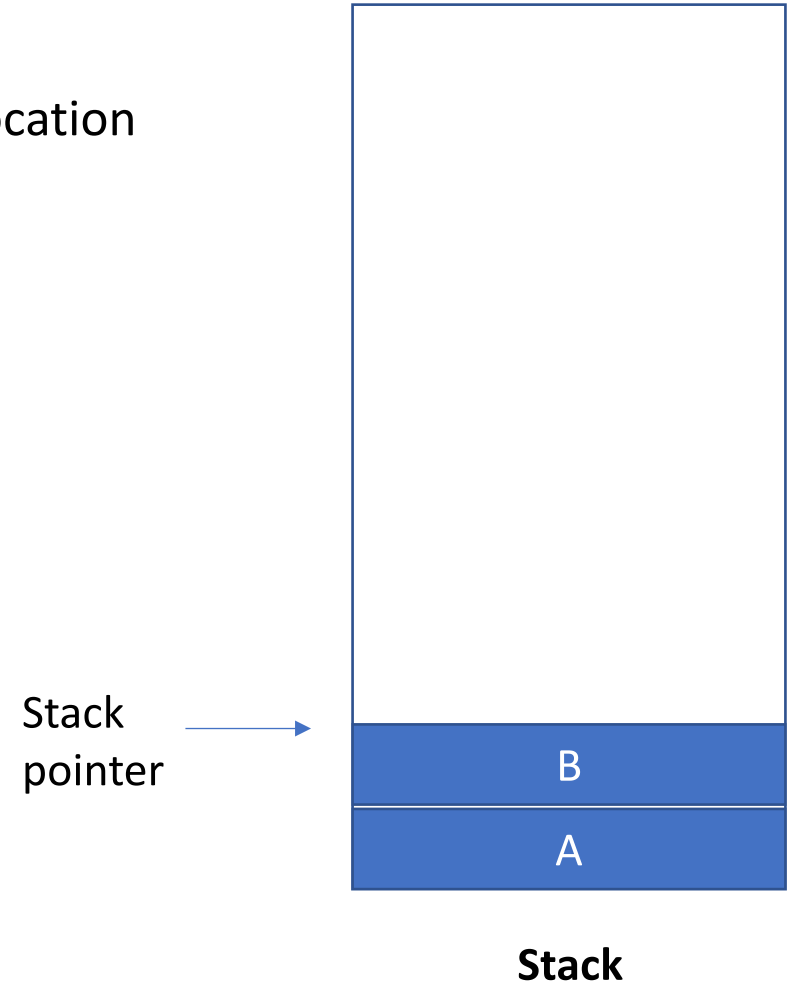
```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
```



Stack

Memory is freed in opposite order from allocation

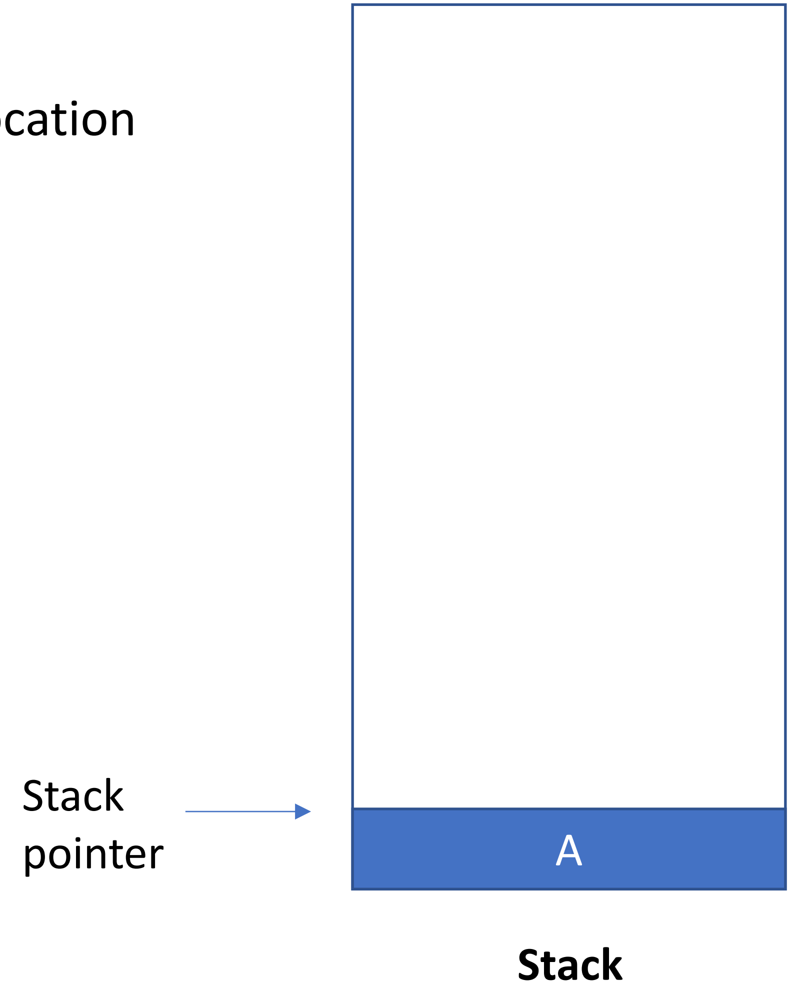
```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
```



Stack

Memory is freed in opposite order from allocation

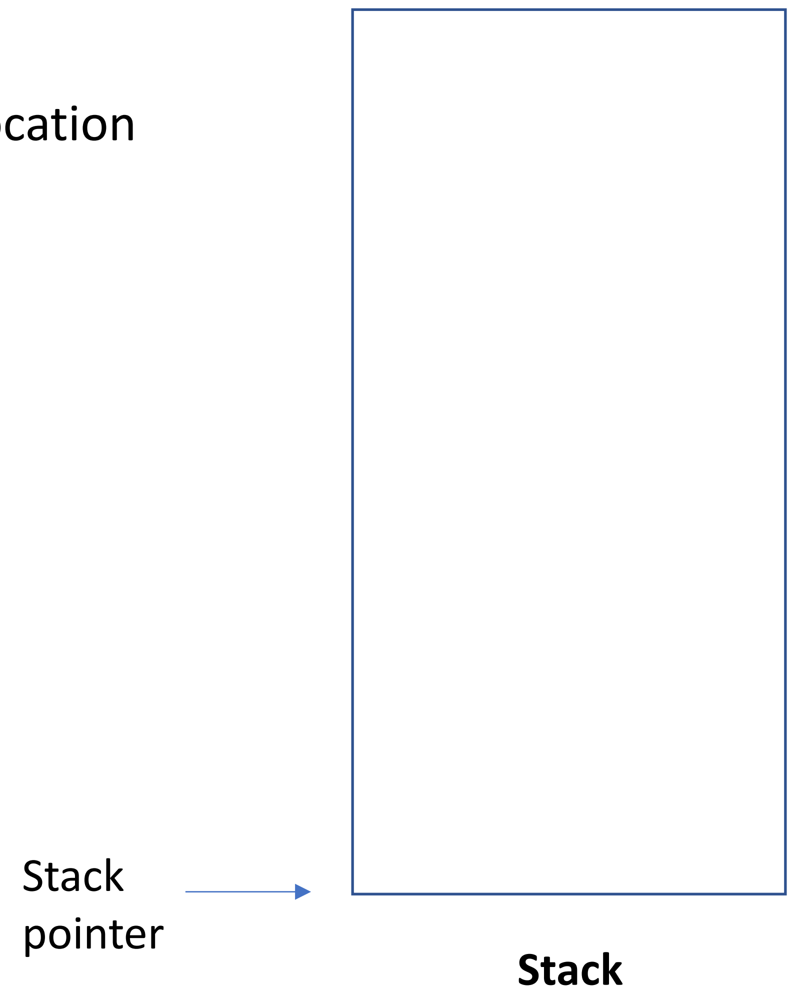
```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```



Stack

Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```



Stack

Simple and efficient implementation:

- Pointer separates allocated and freed space
- Allocate: Increment pointer
- Free: Decrement pointer

No fragmentation

Stack management done automatically

Where are stacks used?

OS uses stack for procedure call frames (local variables and parameters)

```
main () {
      int A = 0;
      foo (A);
      printf("A: %d\n", A);
}

void foo (int Z) {
      int A = 2;
      Z = 5;
      printf("A: %d Z: %d\n", A, Z);
}
```

Where are stacks used?

OS/compilers uses stack for procedure call frames (local variables and parameters)

```
main () {
      int A = 0;
      foo (A);
      printf(“A: %d\n”, A);
}

void foo (int Z) {
      int A = 2;
      Z = 5;
      printf(“A: %d Z: %d\n”, A, Z);
}
```

```
Prints:

A: 2 Z: 5
A: 0
```

Heap

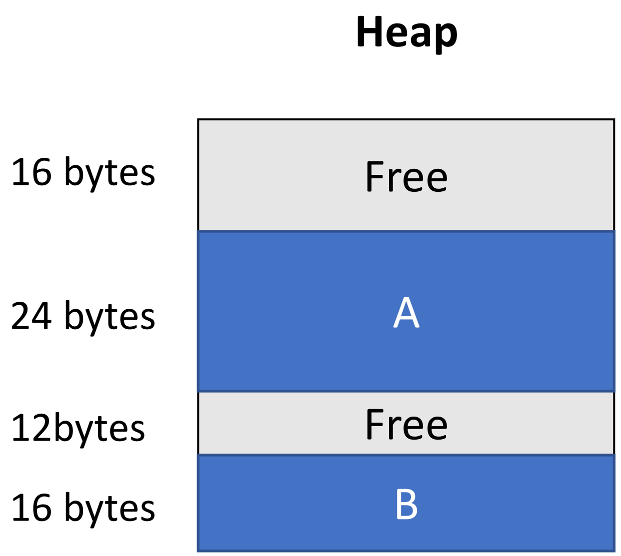
Allocate from any random location

- Heap consists of allocated areas and free areas (holes)
- Order of allocation and free is unpredictable

\+ Works for all data structures

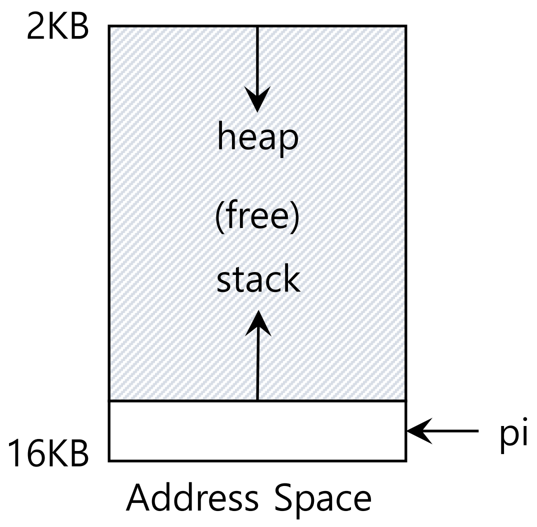
☹ Allocation can be slow

☹ Fragmentation



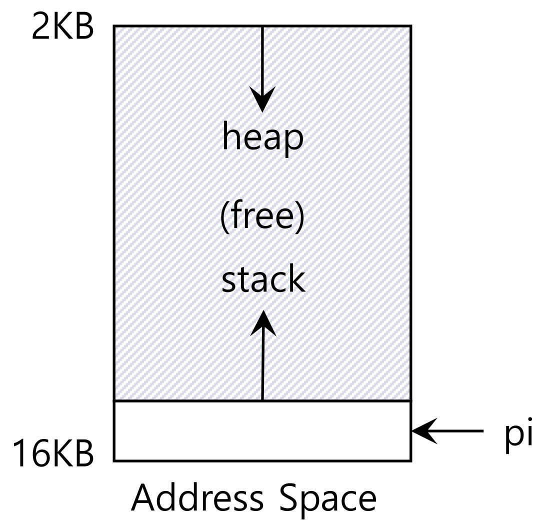
Programmers manage allocations/deallocations with library calls (**malloc/free**)

Memory allocation example

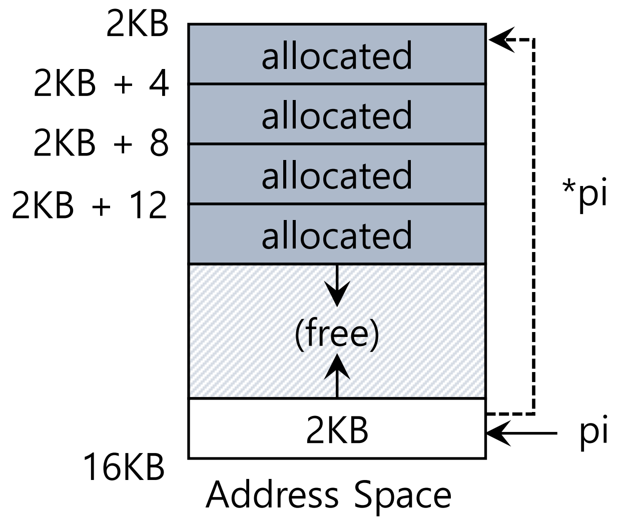



```
int *pi; // local variable
```

Memory allocation example




```
int *pi; // local variable
```




```
pi = (int *)malloc(sizeof(int)* 4);
```

Quiz: Match that Address Location

```
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof int));
}
```

What if no static data segment?

Possible segments: **static data, code, stack, heap**

Address	**Location**
x	Static data → Code
main	Code
y	Stack
z	Stack
*z	Heap

What does a Process do?
(as far as a user is concerned)

What does a Process do? (as far as a user is concerned)

- It can do anything

- Shell
- Compiler
- Editor
- Browser
- …
- These are all processes

Process Identification

- Each process has a unique process identifier
- Always referred to as “pid”

Basic Operations on Processes

- Create a process
- Terminate a process
  - Normal exit
  - Error
  - Terminated by another process

Linux Process Primitives

- pid = fork()
- exec(filename)
- exit()
- wait()
  - We won't talk about this one very much, but you should still know it!

pid = fork()

- Creates an *identical* copy of parent
- In parent, returns pid of child
- In child, returns 0

exec(filename)

- Loads executable from file with filename

wait()

- Wait for one of its children to terminate

exit()

- Terminate the process

Typical fork()-ing Code Segment

```
if (pid = fork()) {
        wait()
}
else {
        exec(filename)
}
```

Before fork()

```
if (pid = fork()) {
        wait()
}
else {
        exec(filename)
}
```

After fork()

parent

```
if (pid = fork()) {
		wait()
}
else {
		exec(filename)
}
```

child

```
if (pid = fork()) {
		wait()
}
else {
		exec(filename)
}
```

After fork()

parent

```
if (pid_child) {
        wait()
}
else {
        exec(filename)
}
```

child

```
if (0) {
        wait()
}
else {
        exec(filename)
}
```

After fork()

parent

```
if (pid_child) {
        wait()
}
else {
        exec(filename)
}
```

child

```
if (0) {
        wait()
}
else {
        exec(filename)
}
```

After exec()

parent

```
if (pid_child) {
        wait()
}
else {
        exec(filename)
}
```

child

```
main () {

  …

  exit()
}
```

After exit()

parent

```
if (pid_child) {
        wait() //wait returns
}
else {
        exec(filename)
}
```

child

```
main () {

  ...

  exit()
}
```

Outline of Linux Shell

```
forever {
  read from input

  if (logout) exit()

  if (pid = fork()) {
    wait()
  }

  else {
    exec( filename )
  }
}
```

Shell Operation

- New command line (!= logout)
  - Shell forks a new process and waits
  - Child executes program on command line

The Linux Process Tree

Boot

- First process after boot is the init process

Boot

- First process after boot is the init process
- Happens by black magic
  - This example is quite simplified because init is such arcane mystery that it could consume the whole hour we have to talk.

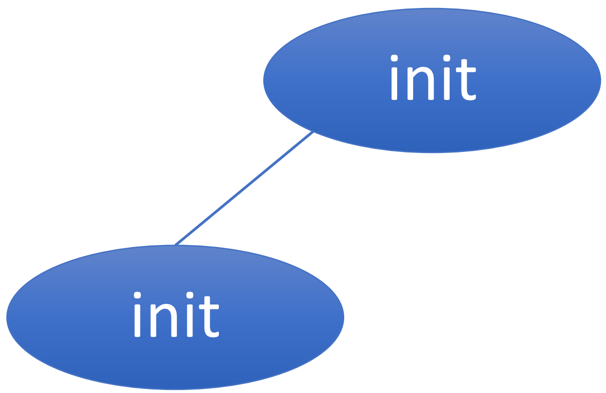
User logs in

User logs in

- Init forks (and *doesn’t* wait)
- Child execs shell

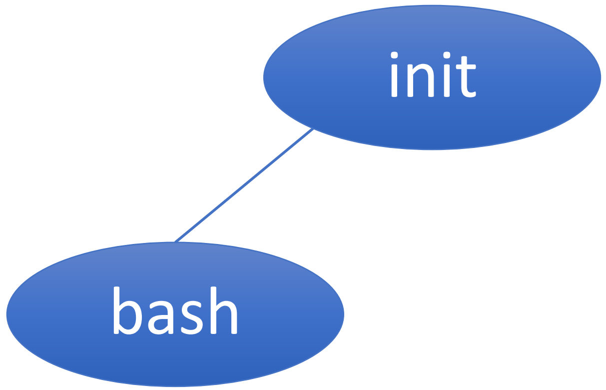
User logs in

- Init forks
- Child execs shell

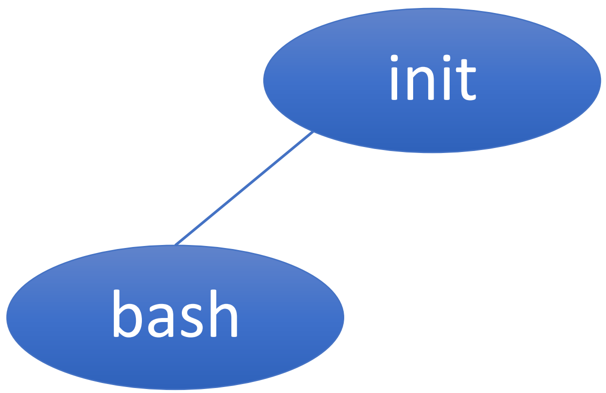


User logs in

- Init forks
- Child execs shell

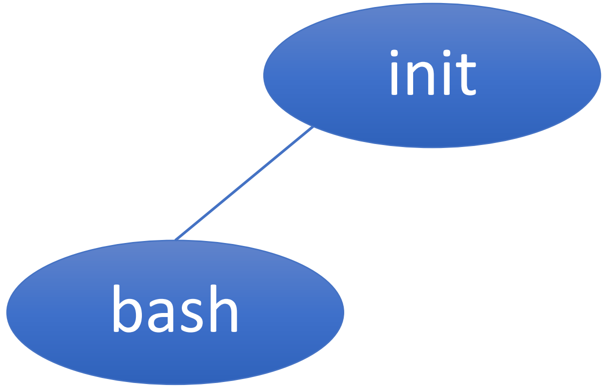


User runs make



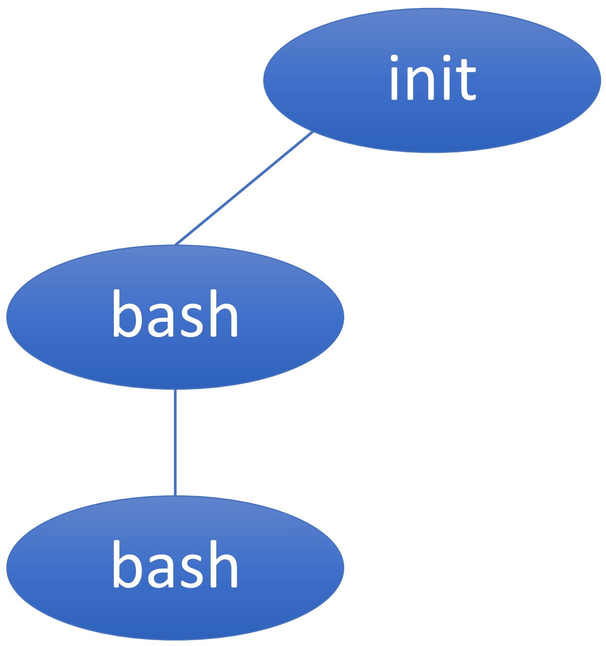
User runs make

- Shell forks and waits
- Child execs make



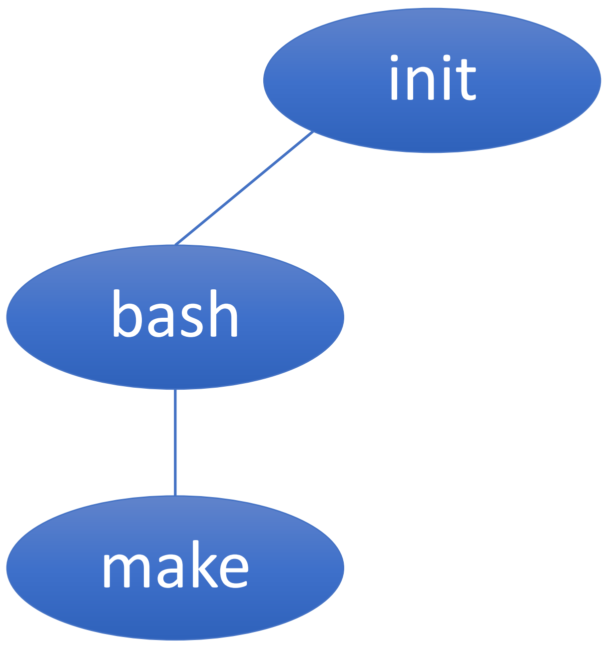
User runs make

- Shell forks and waits
- Child execs make

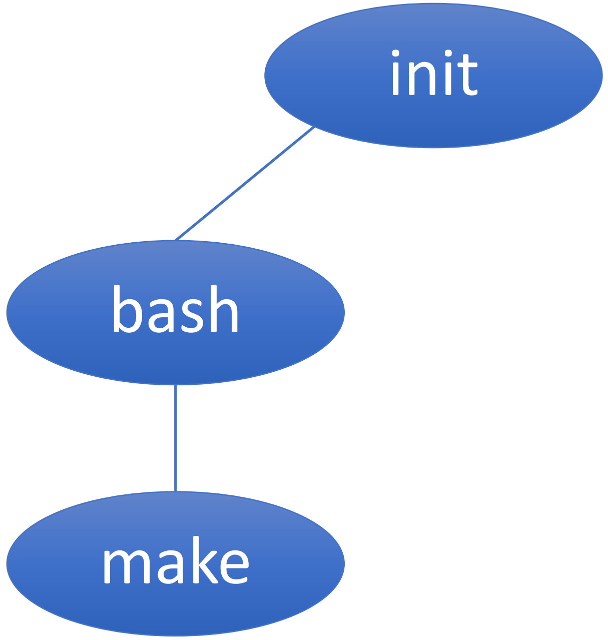


User runs make

- Shell forks and waits
- Child execs make

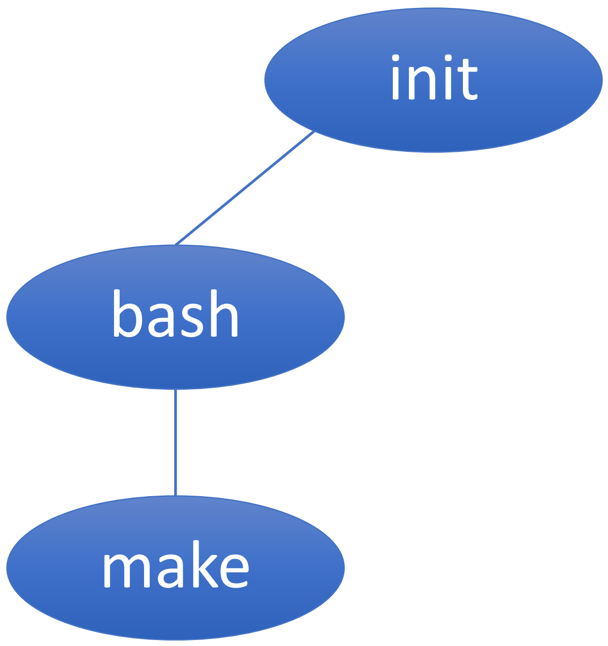


Another user logs in



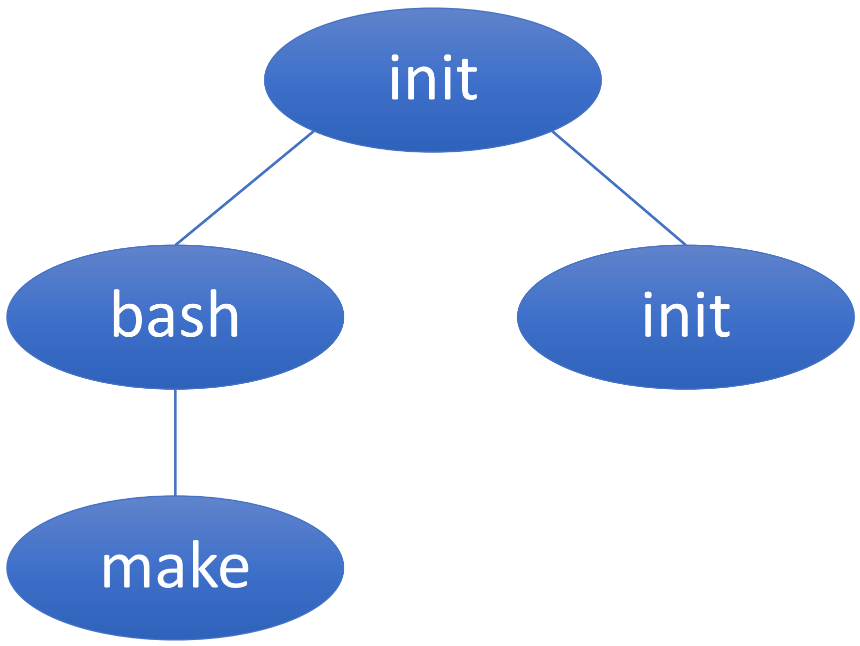
Another user logs in

- Init forks (but *doesn't* wait)
- Child execs shell



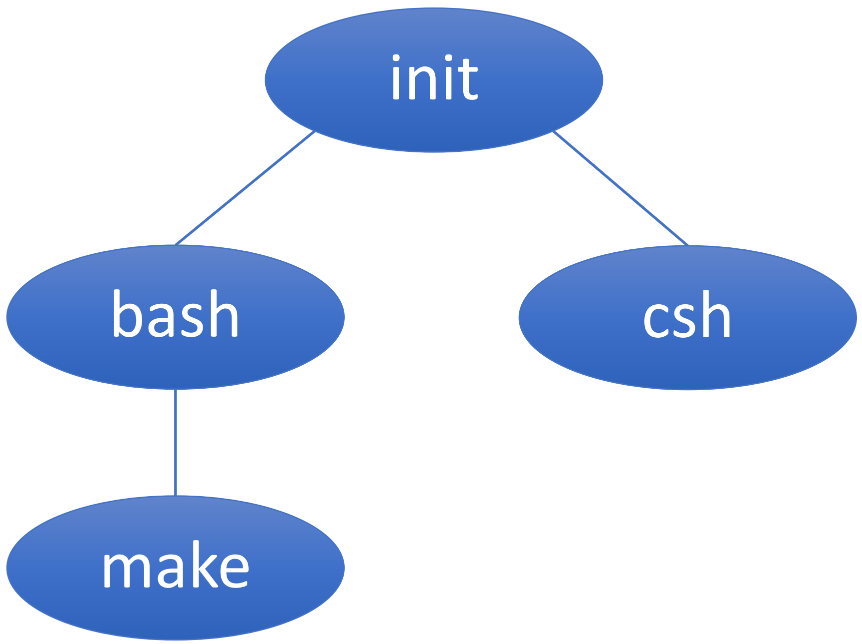
Another user logs in

- Init forks (possible: not waiting!)
- Child execs shell

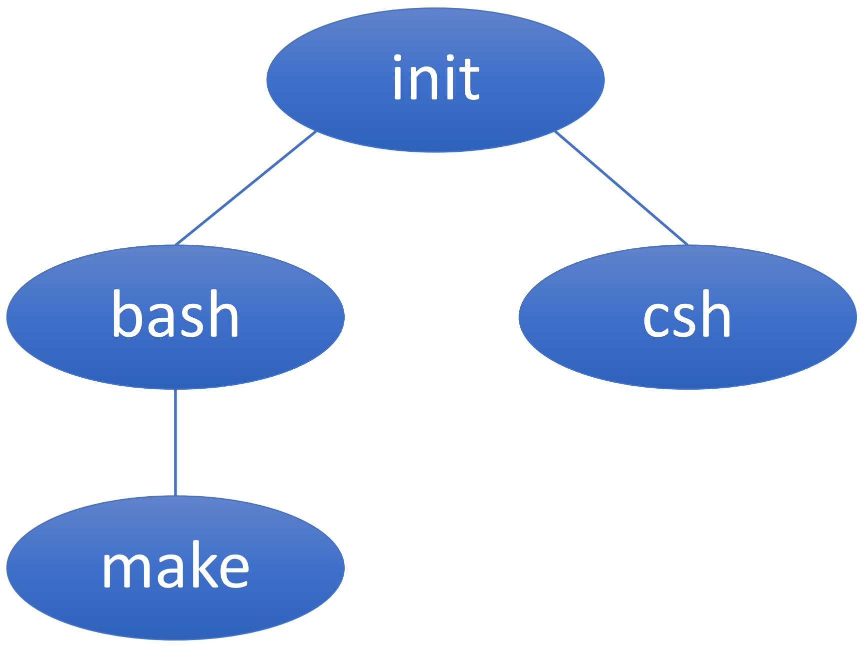


Another user logs in

- Init forks
- Child execs shell

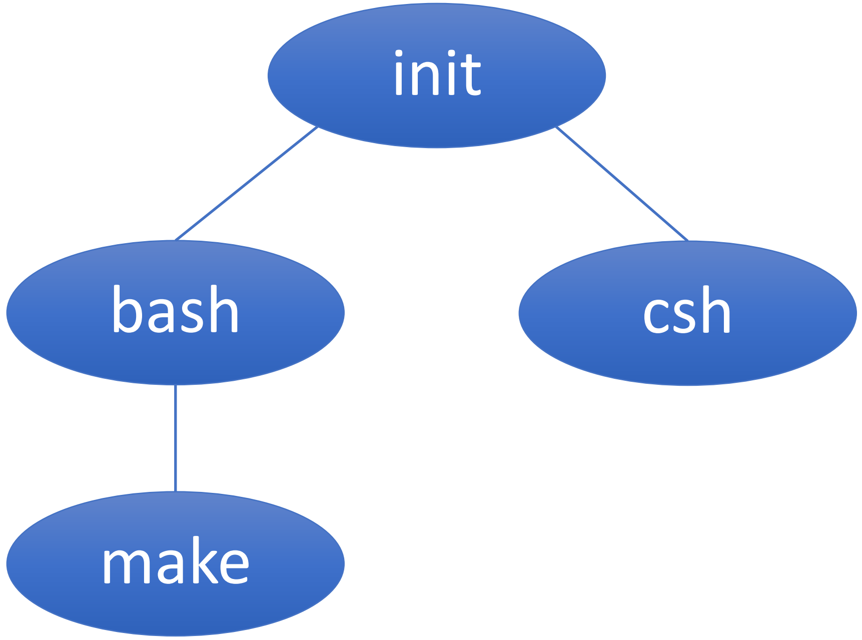


Make runs gcc



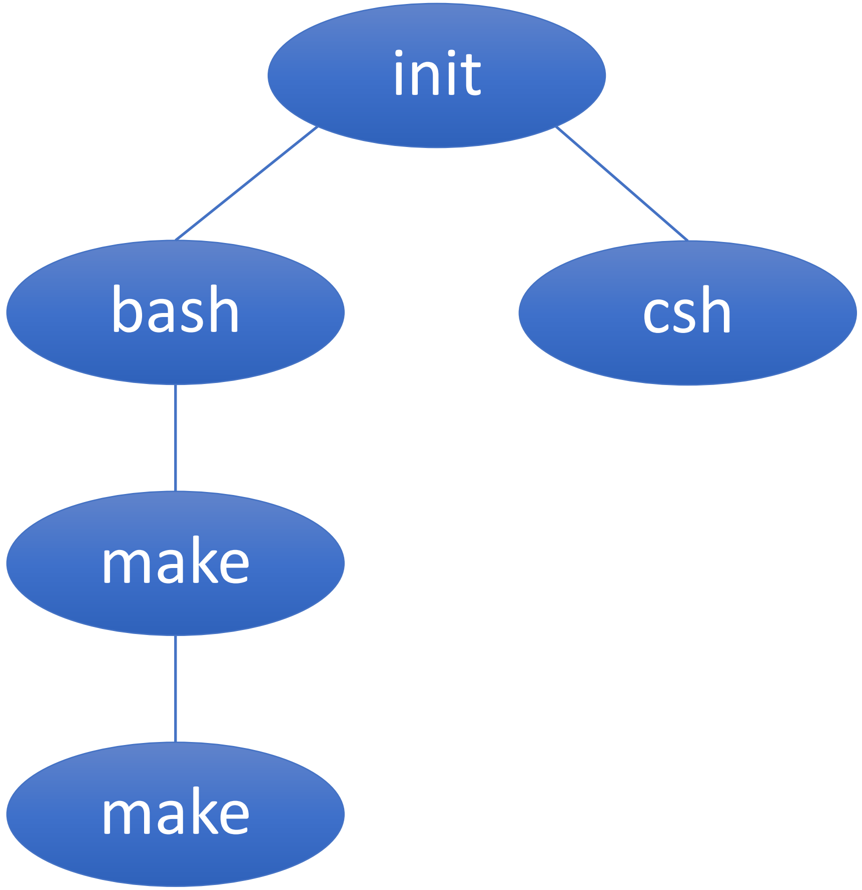
Make runs gcc

- Make forks and waits
- Child execs gcc



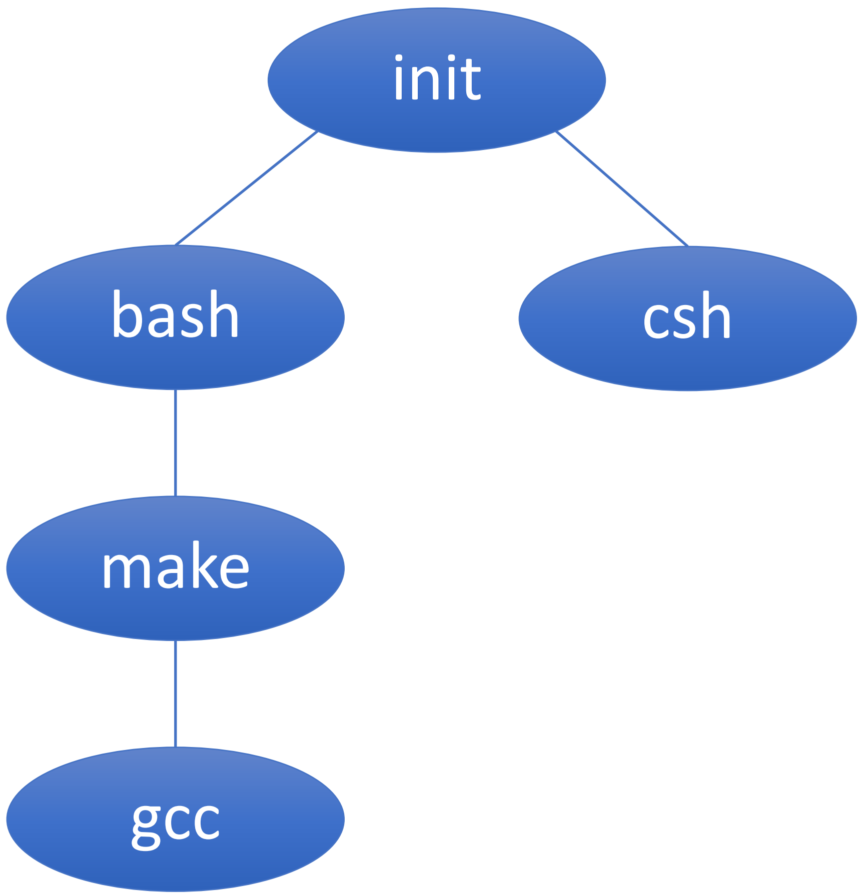
Make runs gcc

- Make forks and waits
- Child execs gcc



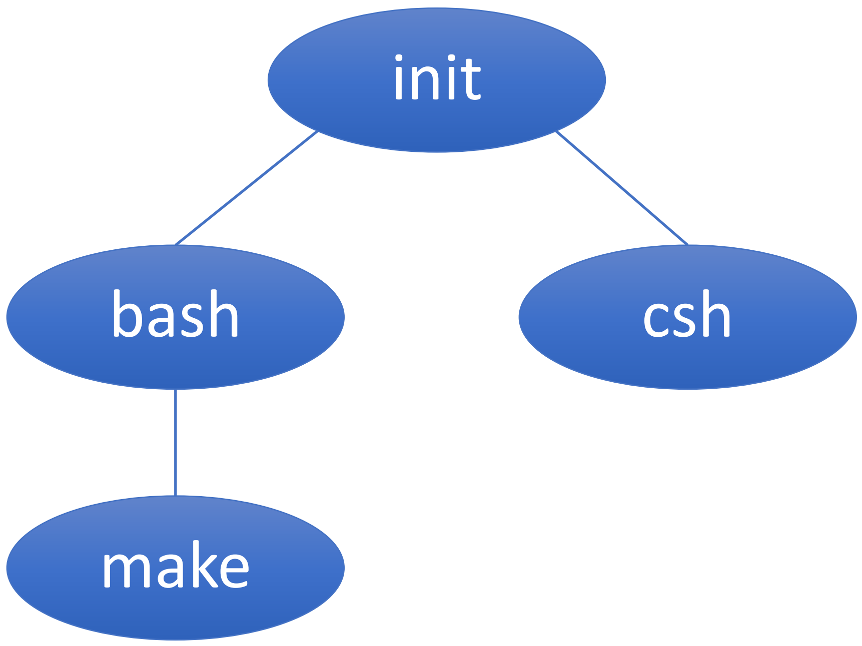
Make runs gcc

- Make forks and waits
- Child execs gcc



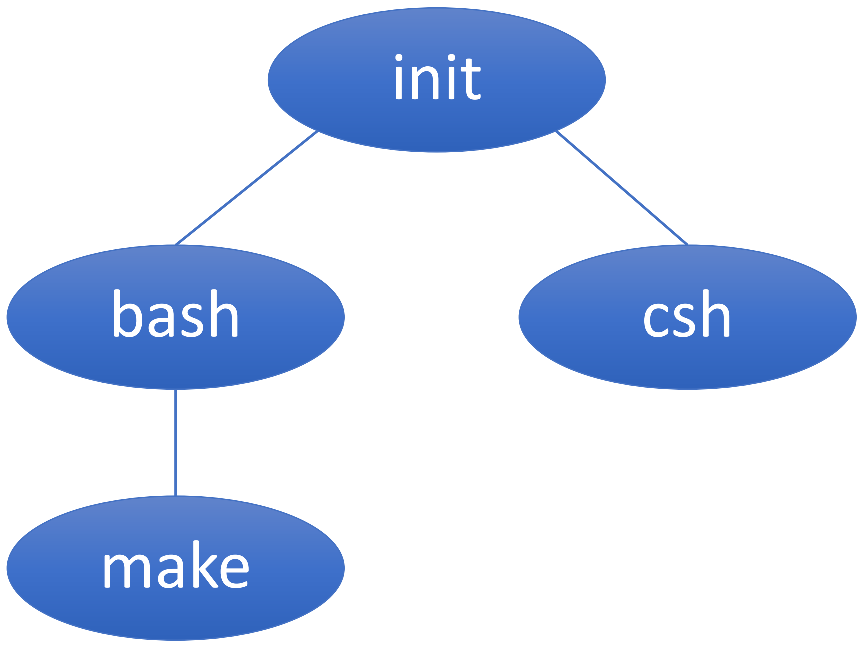
Gcc finishes

- Gcc exits
- Make returns from wait



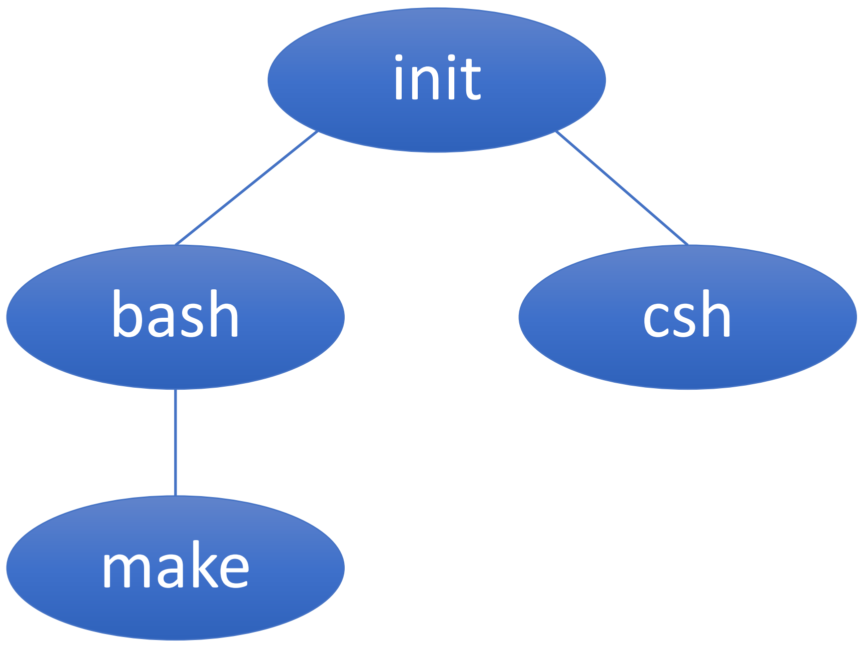
Gcc finishes

- Gcc exits
- Make returns from wait

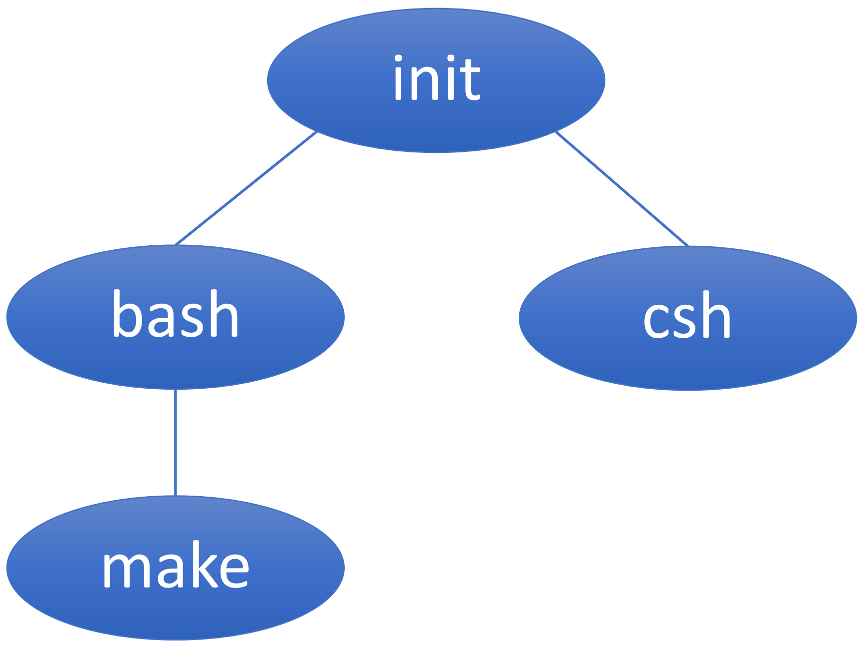


Gcc finishes

- Gcc exits
- Make returns from wait

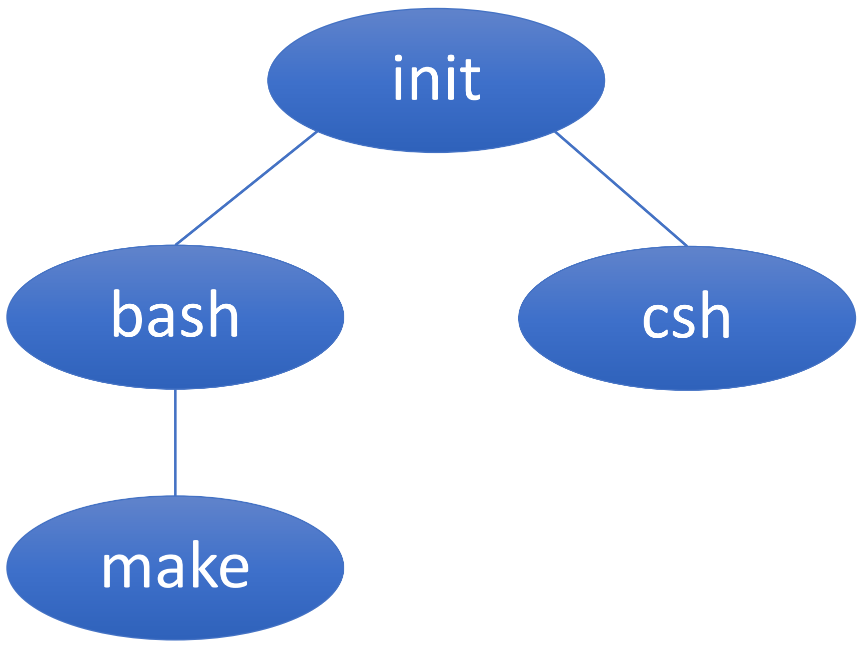


Second user logs out



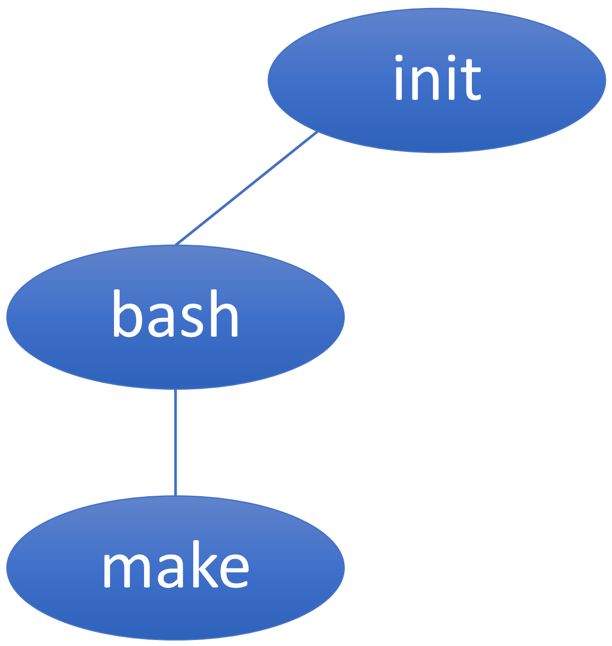
Second user logs out

- Csh exits
- Init *not* waiting – second session simply stops existing



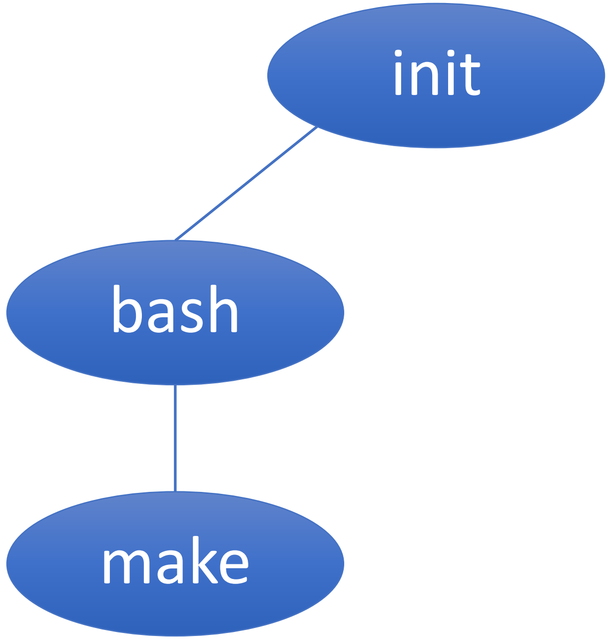
Second user logs out

- Csh exits



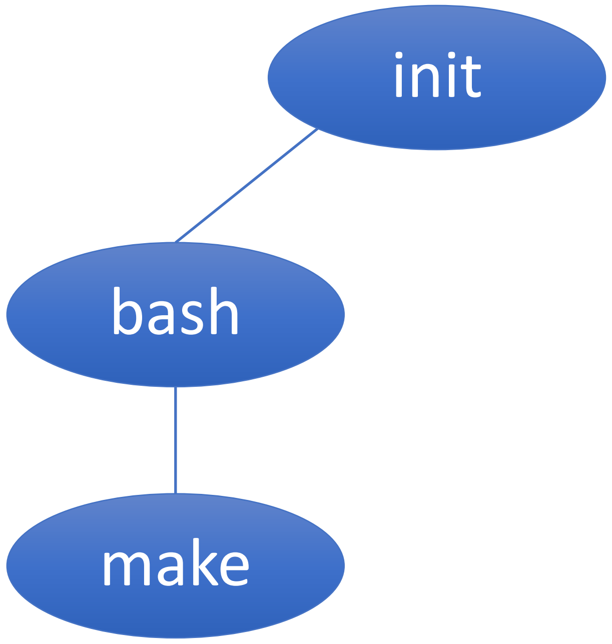
Second user logs out

- Csh exits
- Second session stops existing



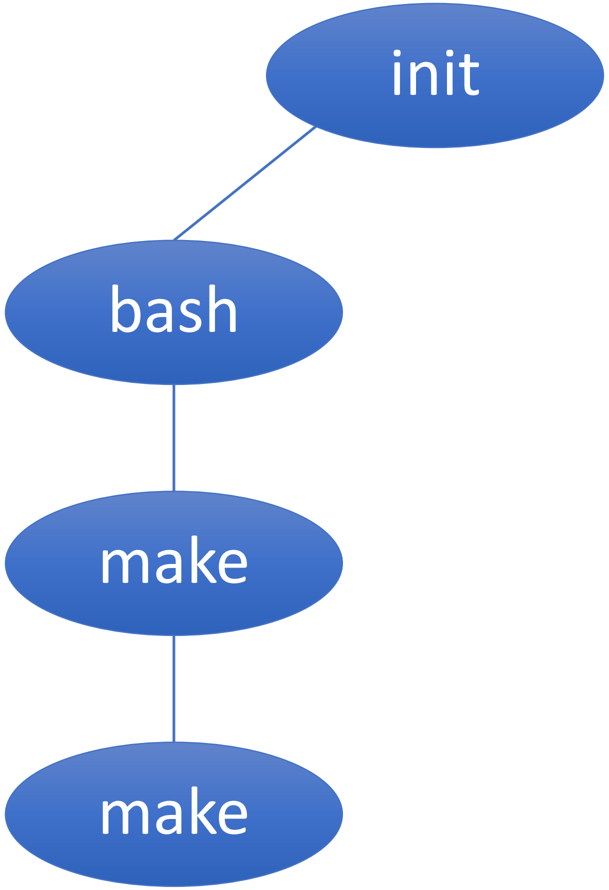
Make runs cp

- Make forks and waits
- Child execs cp



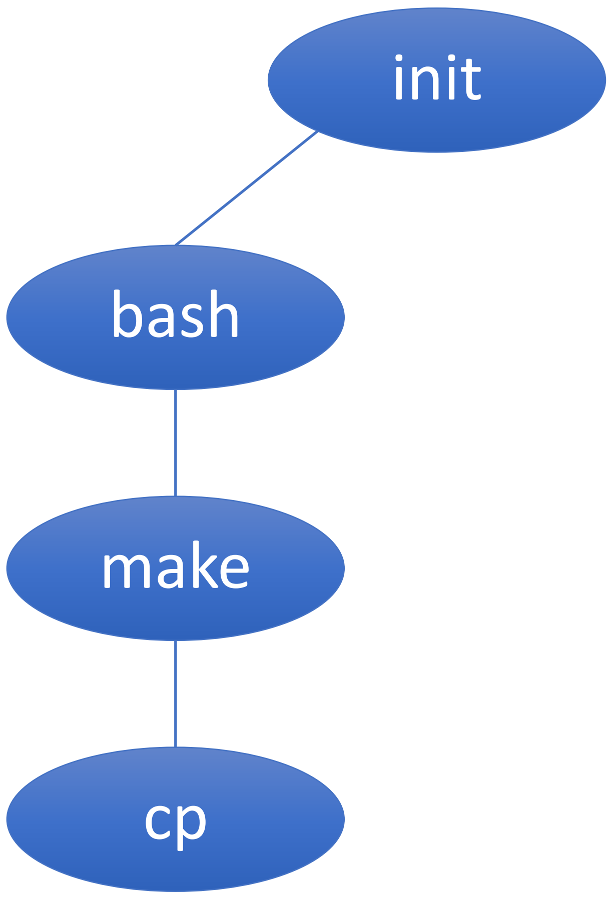
Make runs cp

- Make
  - Make forks and waits
  - Child execs cp



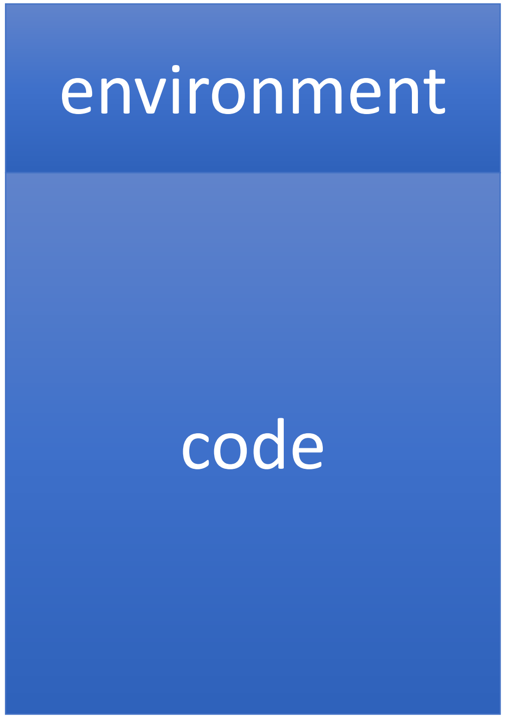
Make runs cp

- Make
  - Make forks and waits
  - Child execs cp



Why fork+exec vs. create?

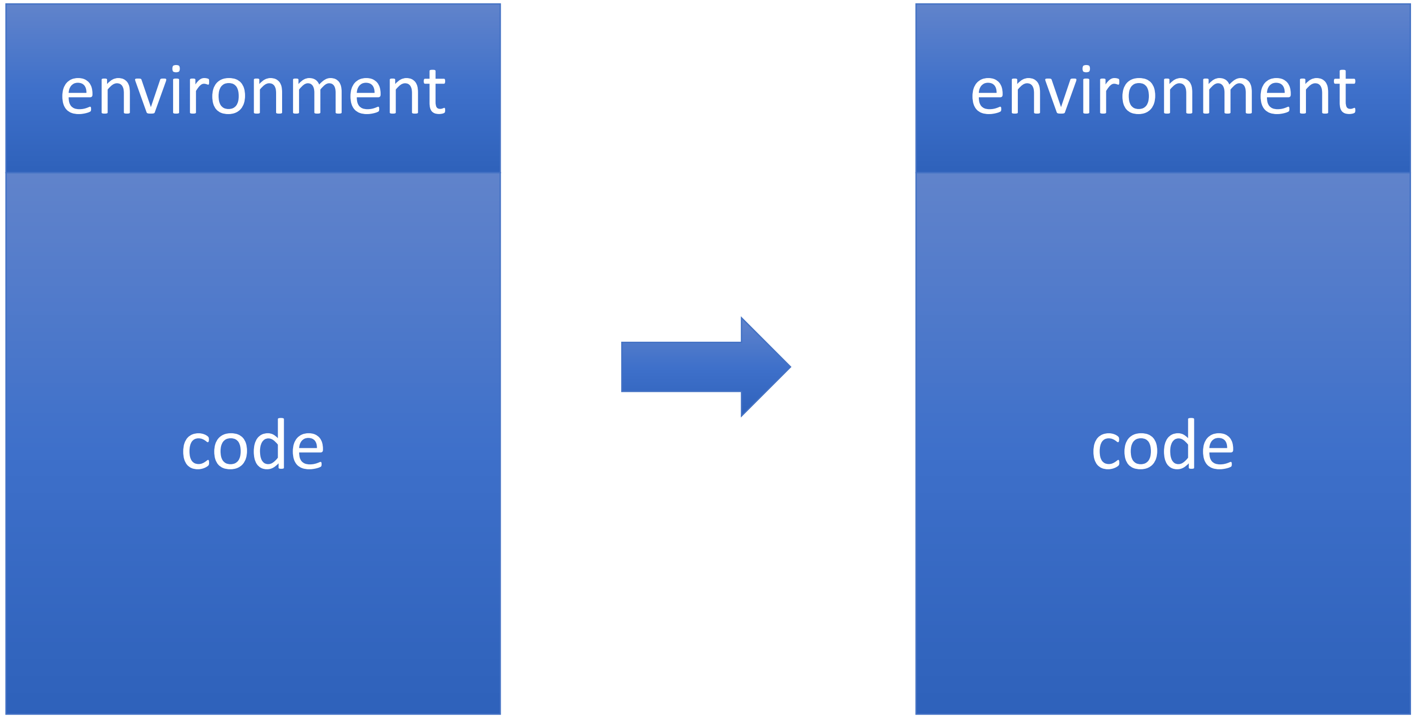
Process = Environment + Code



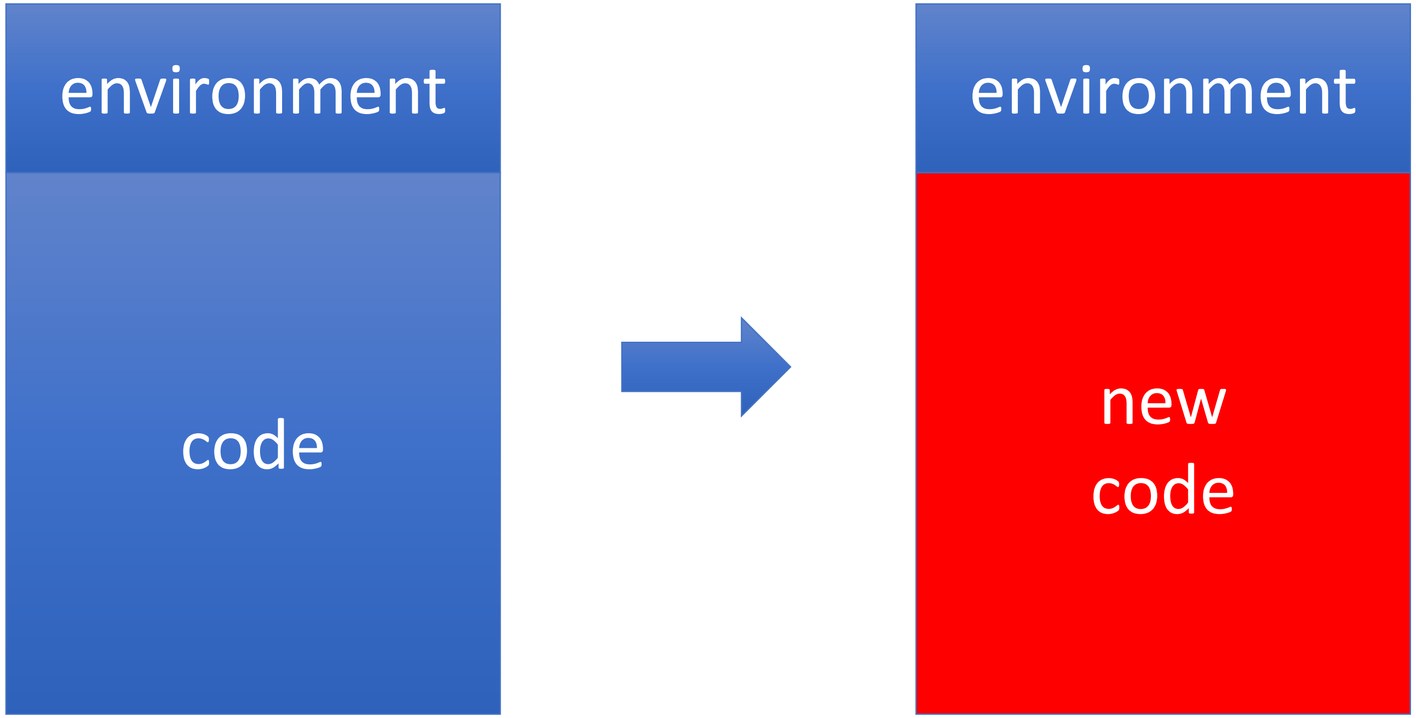
Process = Environment + Code

- Environment includes:
  - Ownership
  - Open files
  - Values of environment variables

After a fork()



After an exec() in the Child



Advantage

- Child automatically inherits environment

Given New Definition of exec

```
forever {
  read from input

  if (logout) exit()

  if (pid = fork()) {
    wait()
  }

  else {

    exec( filename )
  }
}
```

does it make sense to write code here?

Answer

- Yes
- Shell can manipulate environment of child
- For instance, can manipulate stdin and stdout

Let’s practice!

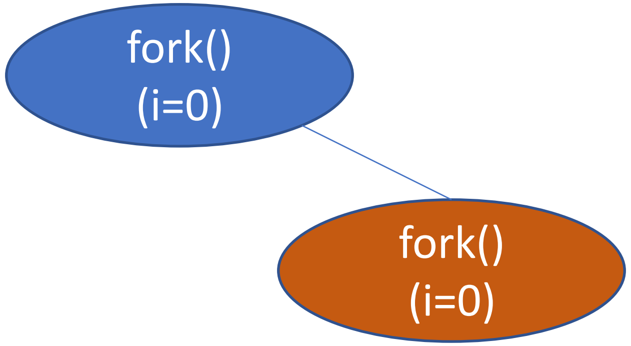
How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

```c
int main(void) {
  for (int i = 0; i < 3; i++) {
     pid_t fork_ret = fork();
  return 0;
}
```

Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

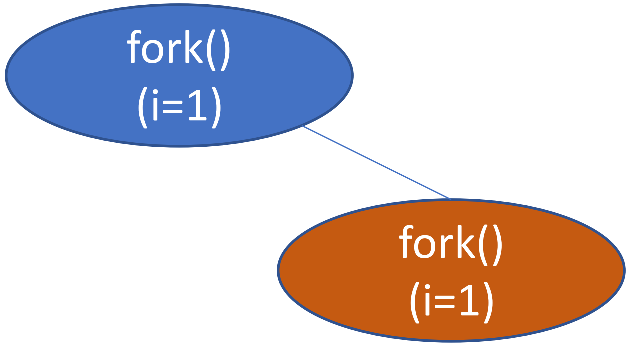
```
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

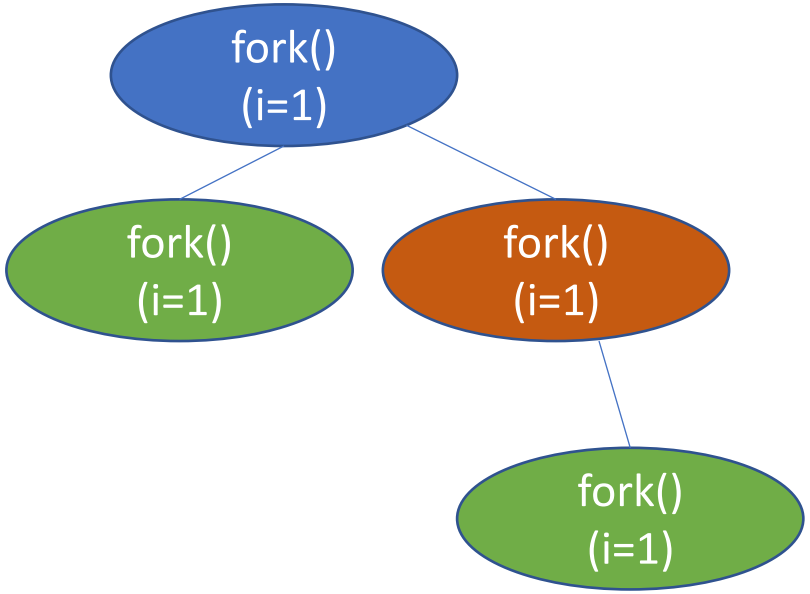
```c
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

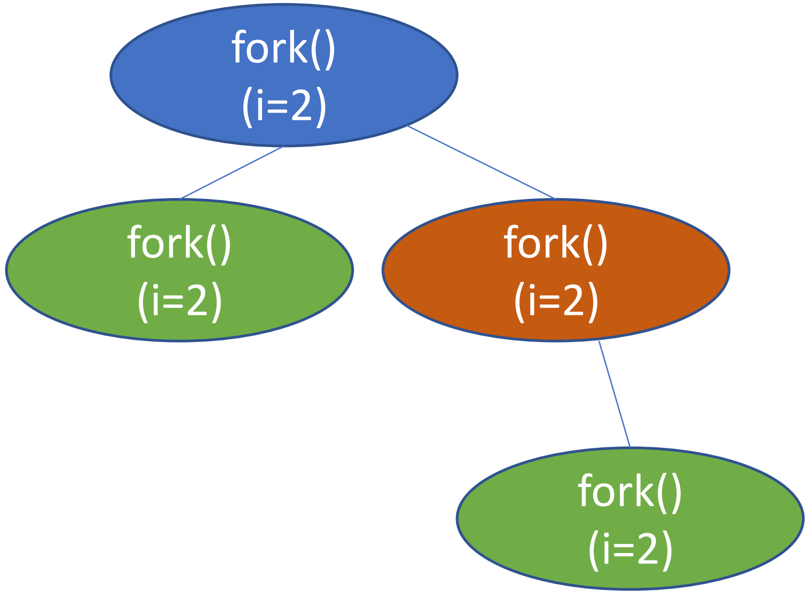
```
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

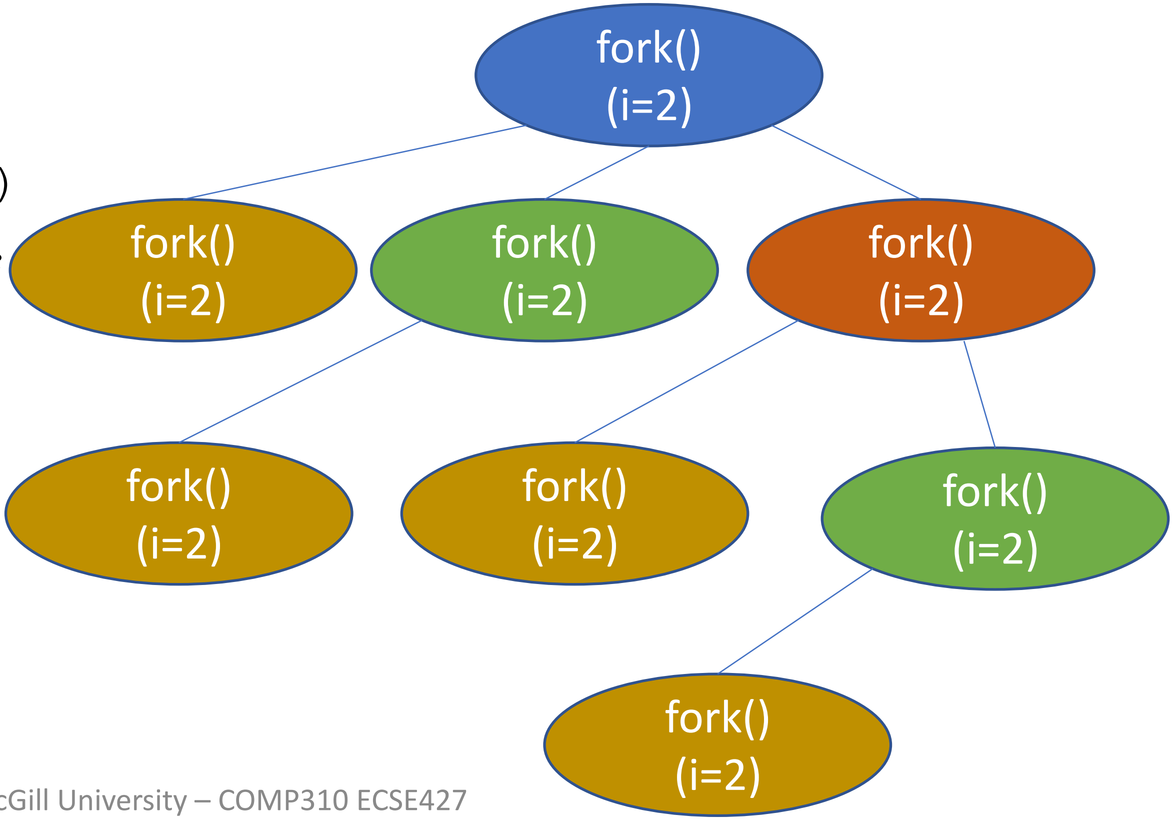
```c
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



Let’s practice!

How many new processes (not including the original process) are created when the following program is run? Assume all fork calls succeed.

```
int main(void) {
  for (int i = 0; i < 3; i++)
     pid_t fork_ret = fork();
  return 0;
}
```



7 new processes created

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

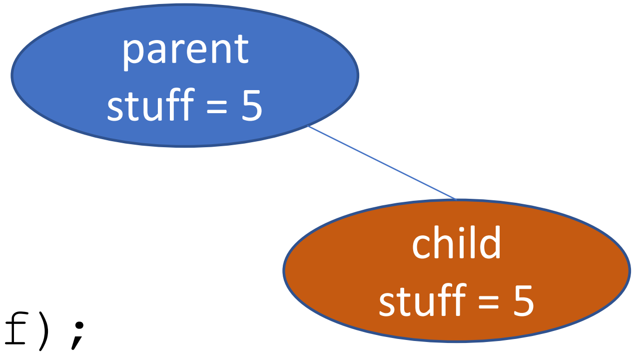
```c
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```



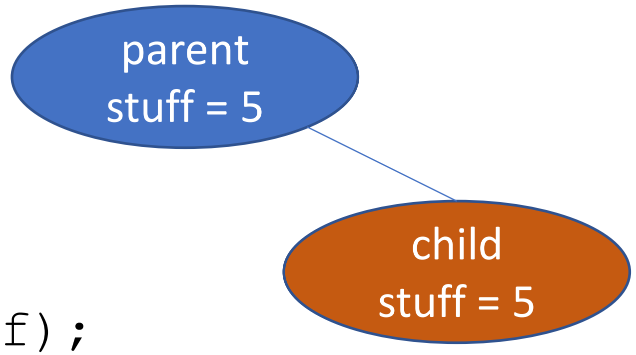
In this case, parent does not wait
→ Child and parent can be executed in any order

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```



In this case, parent does not wait
→ Child and parent can be executed in any order

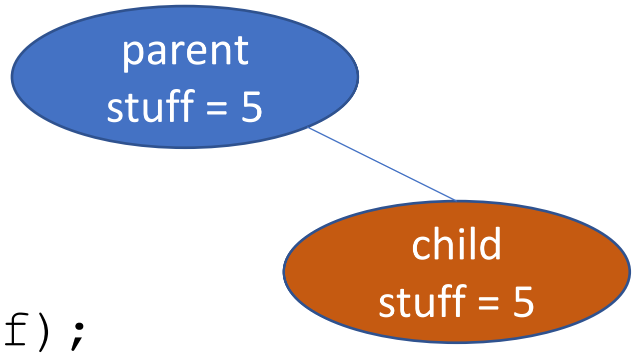
Does the order matter?

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
int main(void) {
  int stuff = 5;
  pid_t fork_ret = fork();
  printf("The last digit of pi is %d\n", stuff);
  if (fork_ret == 0)
    stuff = 6;
  return 0;
}
```



Assume parent goes first:
Program prints:
The last digit of pi is 5.
The last digit of pi is 5.

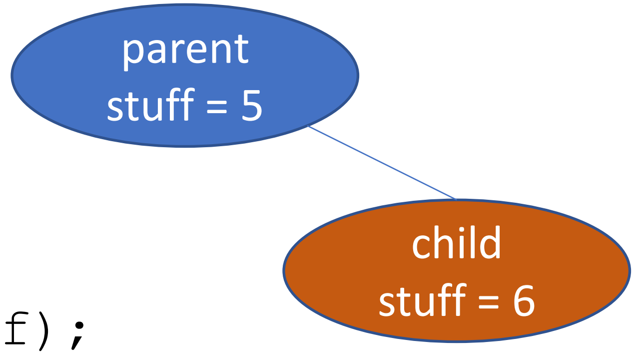
Why?

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
1 int main(void) {
2   int stuff = 5;
3   pid_t fork_ret = fork();
4   printf("The last digit of pi is %d\n", stuff);
5   if (fork_ret == 0)
6     stuff = 6;
7   return 0;
8 }
```



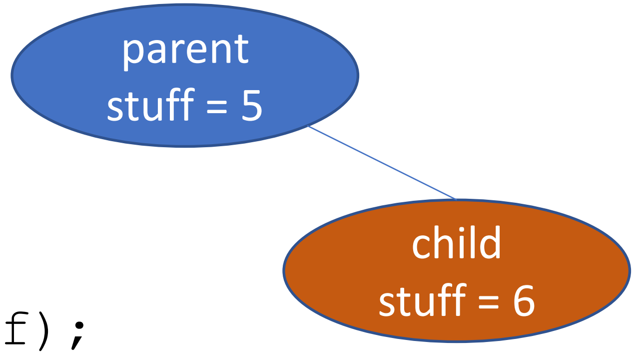
Assume child goes first:
Line 6 executes.
Does this affect the parent's `stuff` value?

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.

```
1 int main(void) {
2   int stuff = 5;
3   pid_t fork_ret = fork();
4   printf("The last digit of pi is %d\n", stuff);
5   if (fork_ret == 0)
6     stuff = 6;
7   return 0;
8 }
```



Assume child goes first:

Line 6 executes.

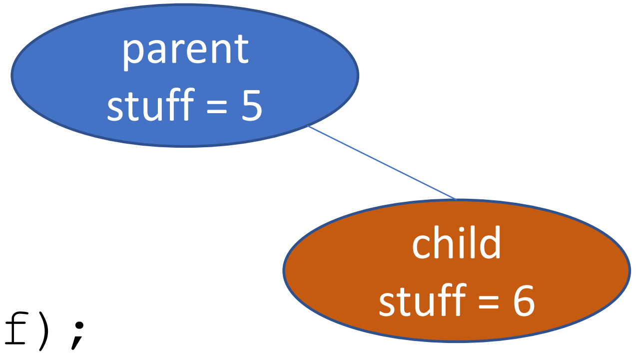
Does this affect the parent's `stuff` value?

No. Child's environment is cloned.

More practice

What are the possible outputs when the following program is run?

Assume all fork calls succeed.




```
1 int main(void) {
2   int stuff = 5;
3   pid_t fork_ret = fork();
4   printf("The last digit of pi is %d\n", stuff);
5   if (fork_ret == 0)
6     stuff = 6;
7   return 0;
8 }
```

Program prints:
The last digit of pi is 5.
The last digit of pi is 5.

Assume child goes first:
Line 6 executes.
Does this affect the parent's `stuff` value?
No. Child's environment is cloned.

What does a process do?
(as far as a user is concerned)

- It can do anything

- Shell
- Compiler
- Editor
- Browser
- …

- These are all processes

What does a process do?
(as far as the OS is concerned)

Week 2

Introduction to Process Management

Max Kopinsky
January 16, 2025

What does a process do?
(as far as the OS is concerned)

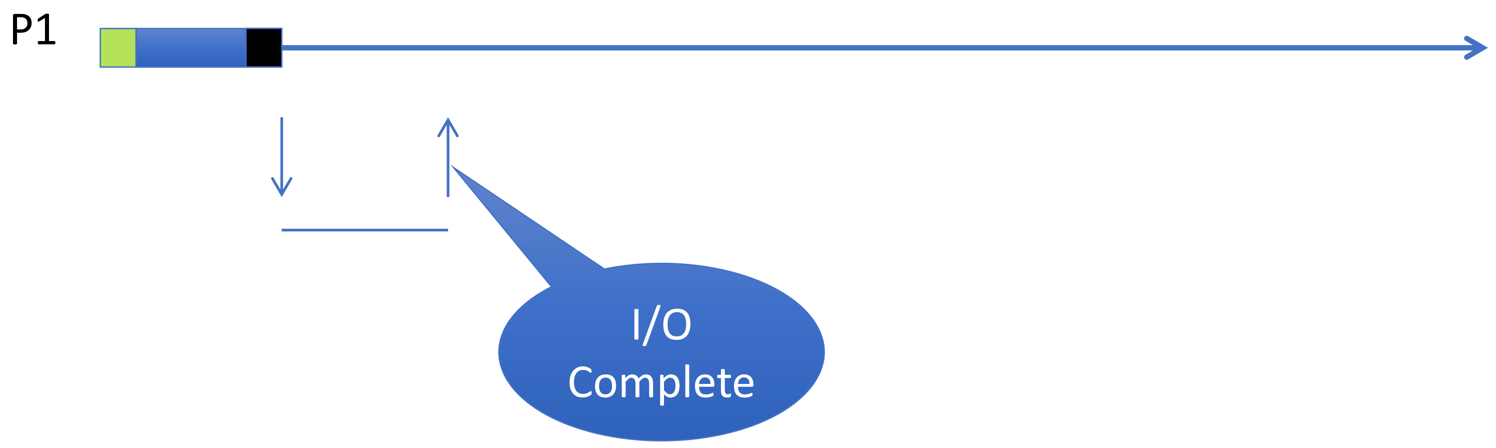
- Either it computes (uses the CPU)
- Or it does I/O (uses a device)

Single Process System

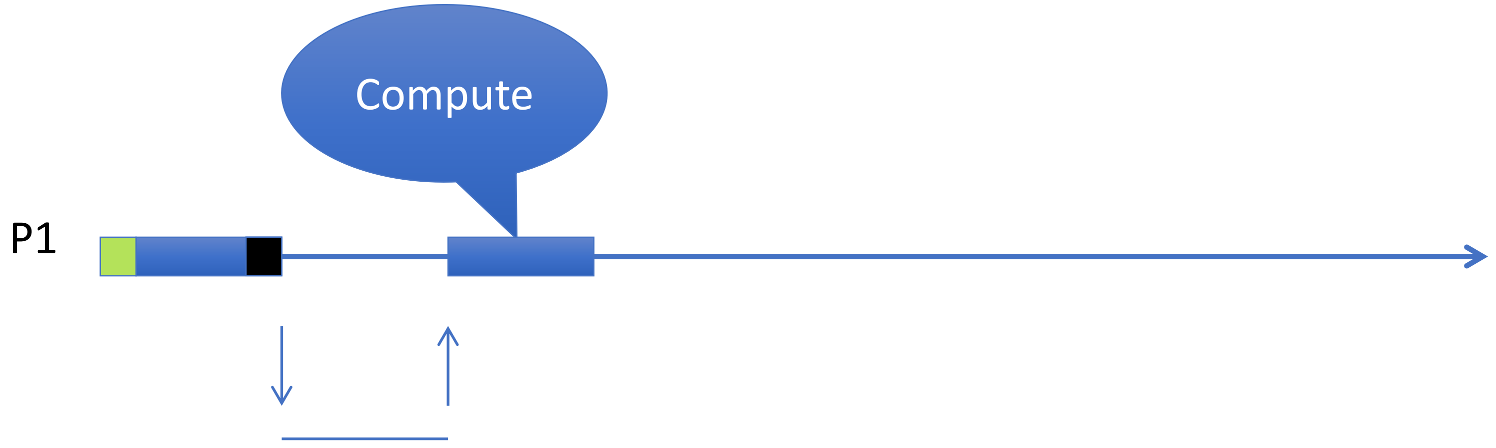
Single Process System

Single Process System

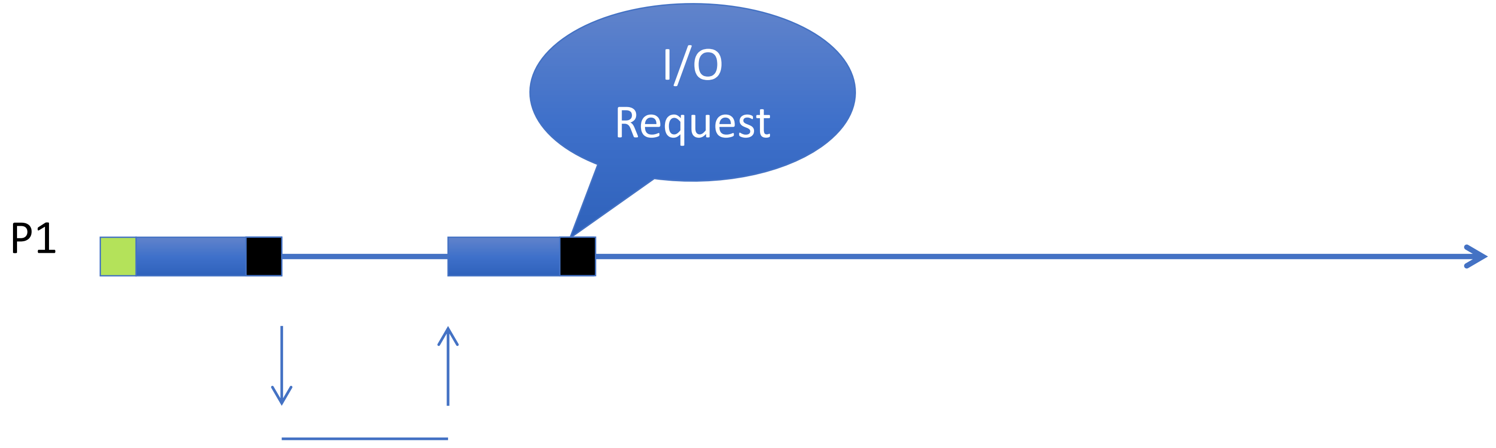
Single Process System



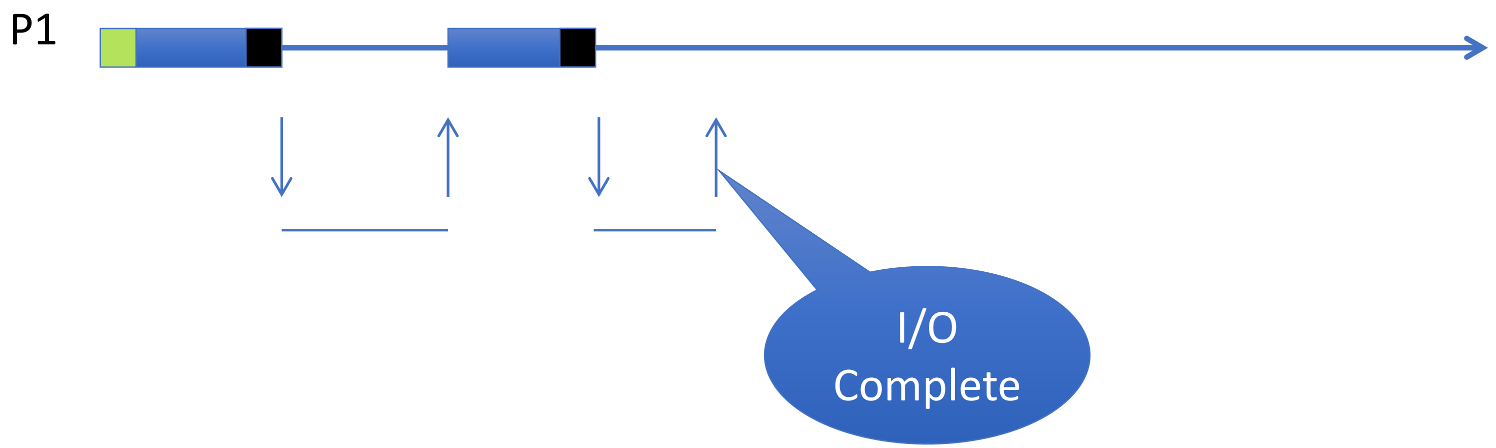
Single Process System



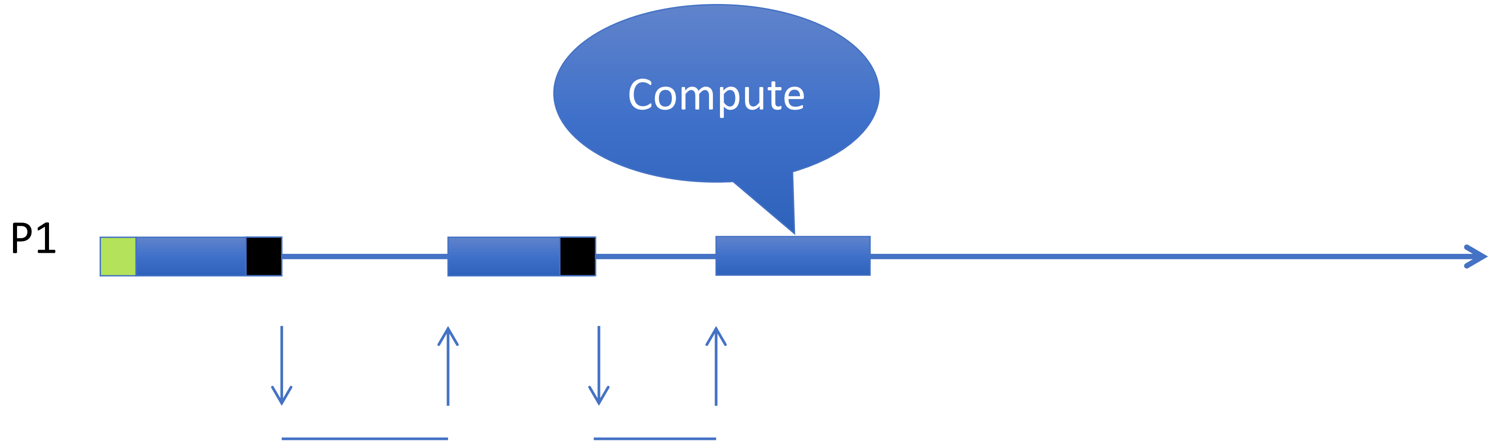
Single Process System



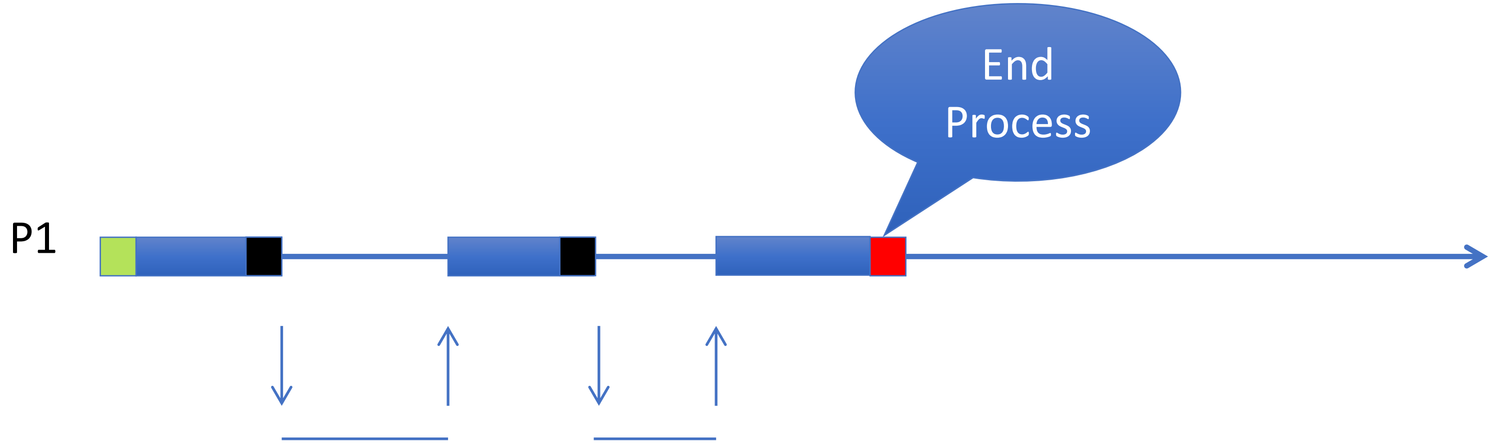
Single Process System



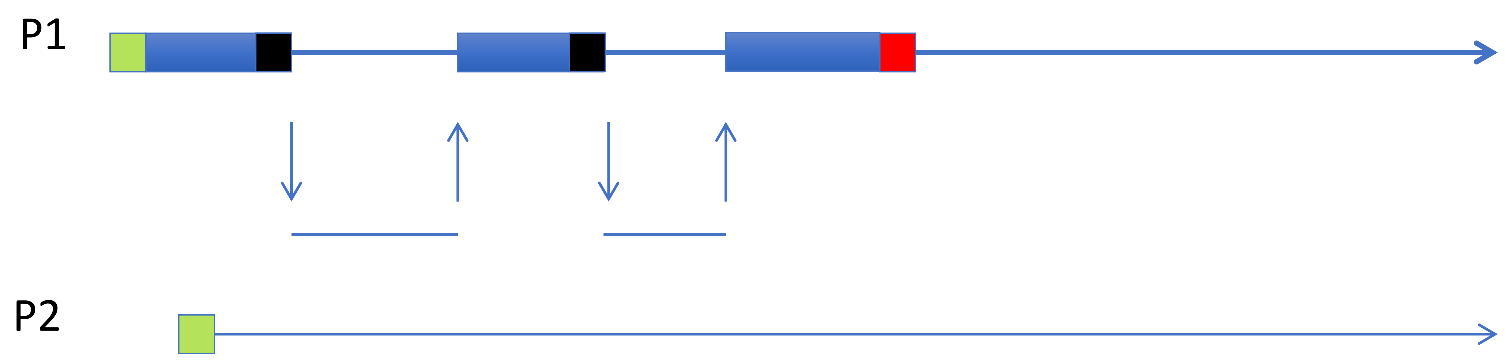
Single Process System



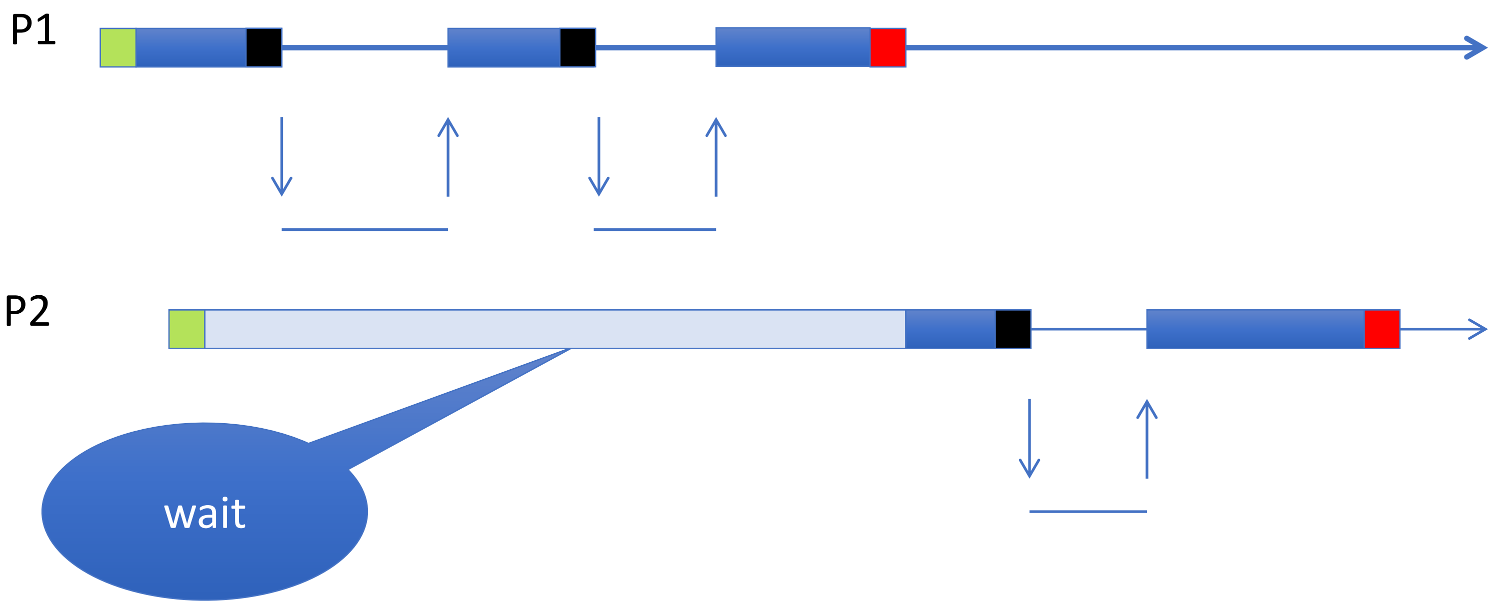
Single Process System



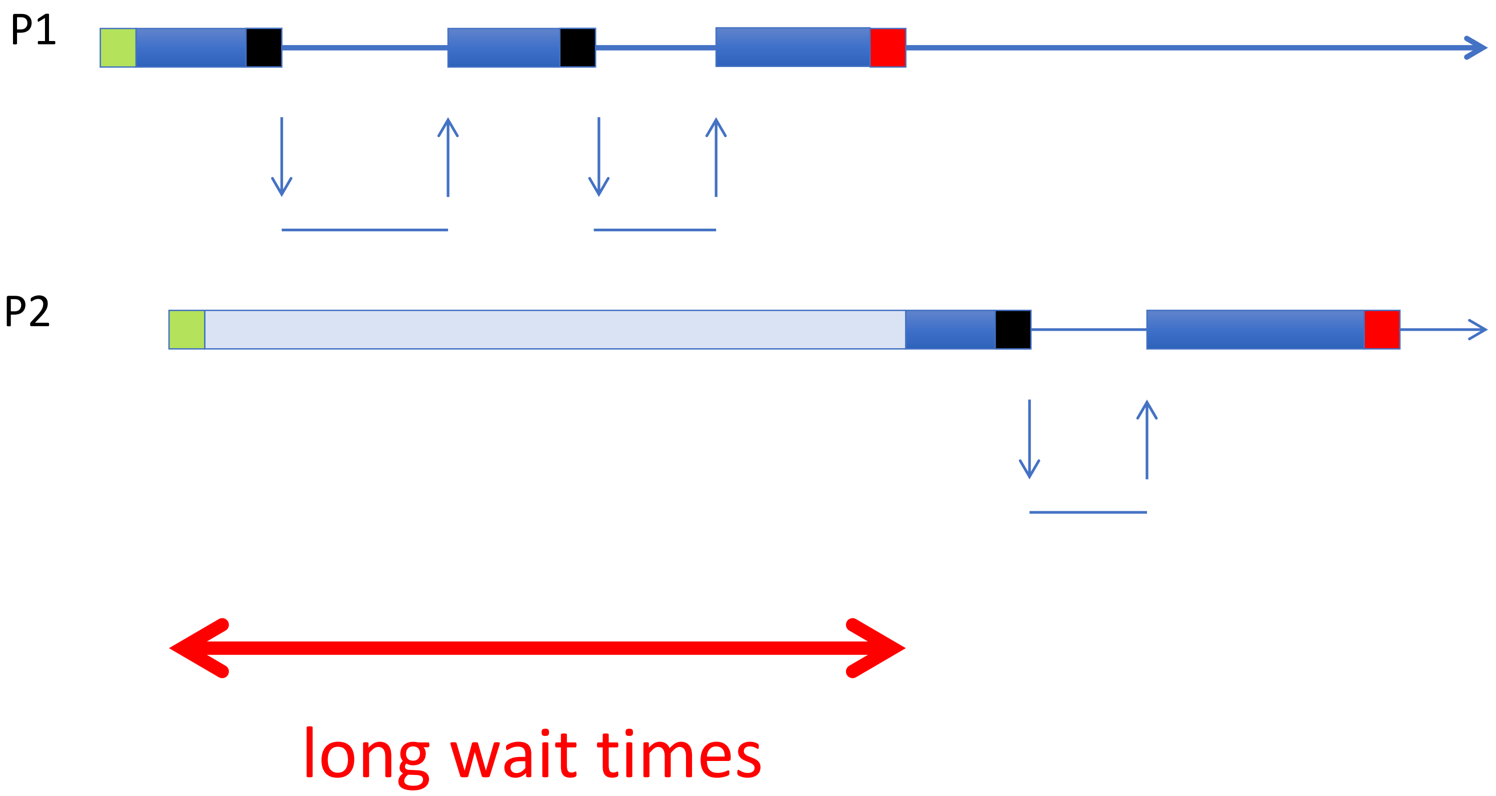
A Second Process



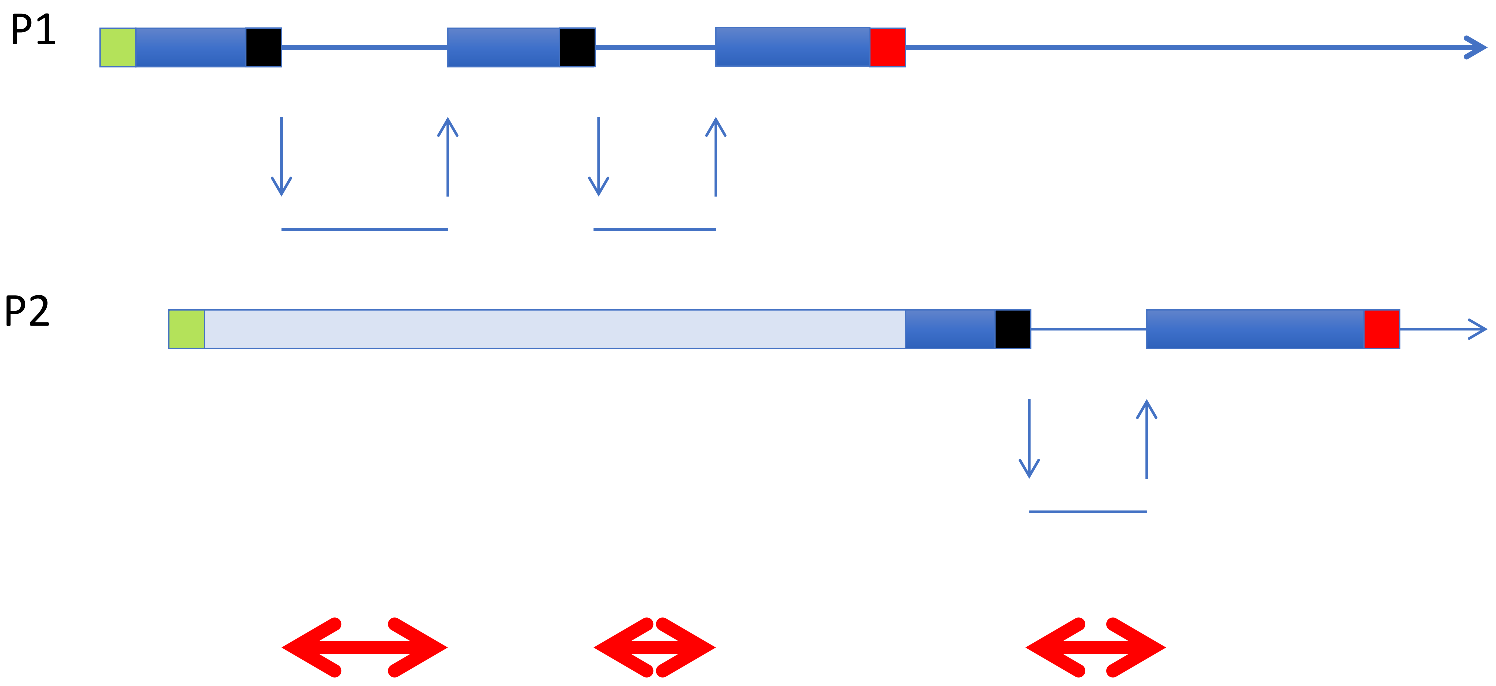
A Second Process



Two Issues



Two Issues



low utilization (long CPU idle times)

Single Process System

- Is very inefficient
  - Very poor CPU utilization
- Is very annoying
  - You can't do anything else

Multiprocess System

- Many processes in the system
- One uses the CPU
- When it does an I/O
  - It waits for the I/O to complete
  - It leaves the CPU idle
- *Another process gets the CPU*

Multiprocess System

Multiprocess System

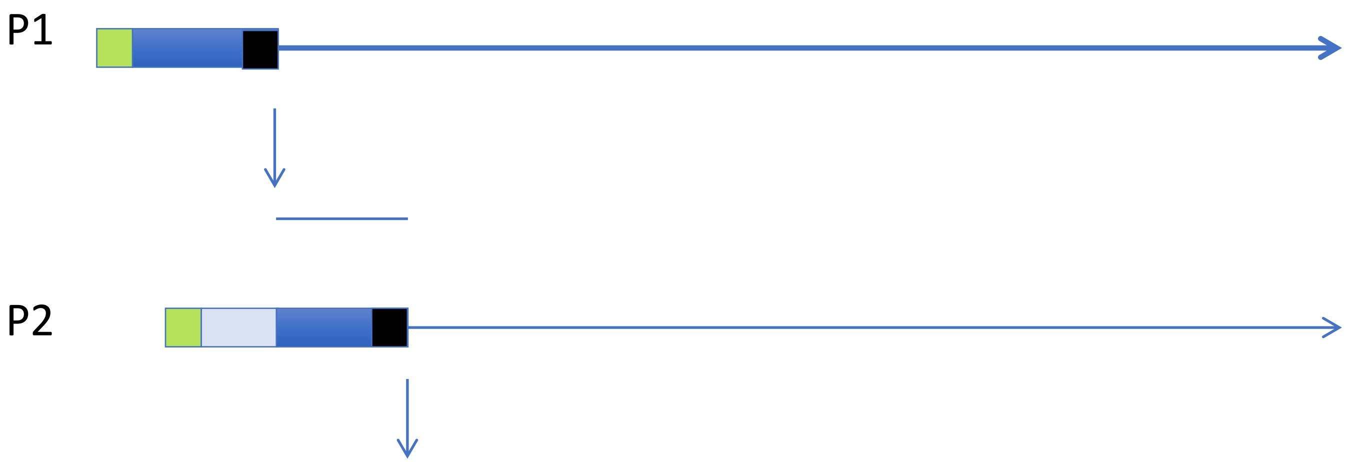
Multiprocess System

Multiprocess System

Multiprocess System

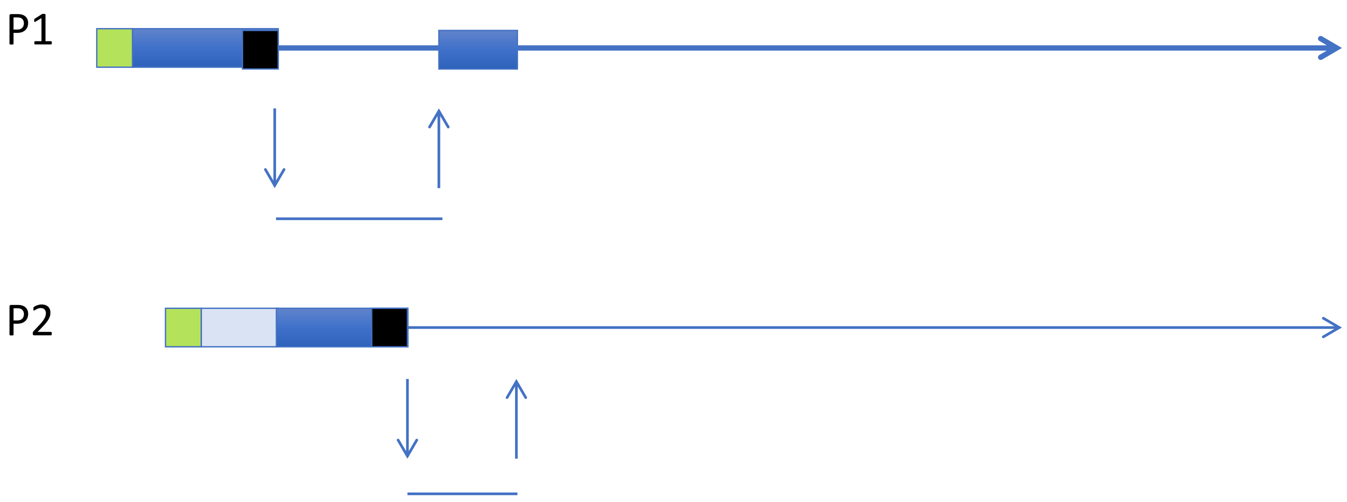
Multiprocess System

Multiprocess System

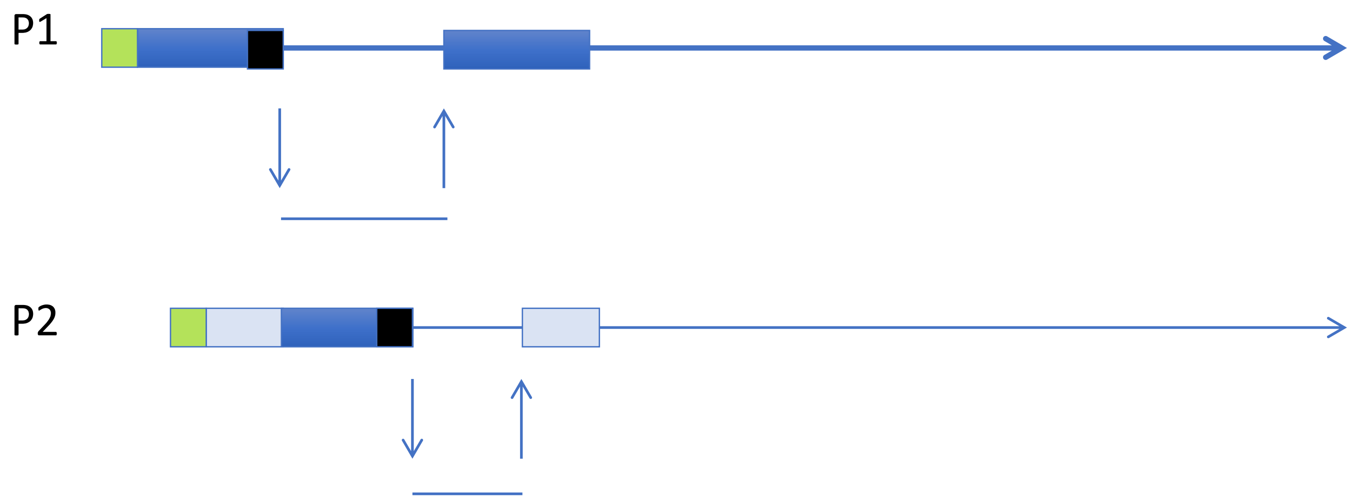


Multiprocess System

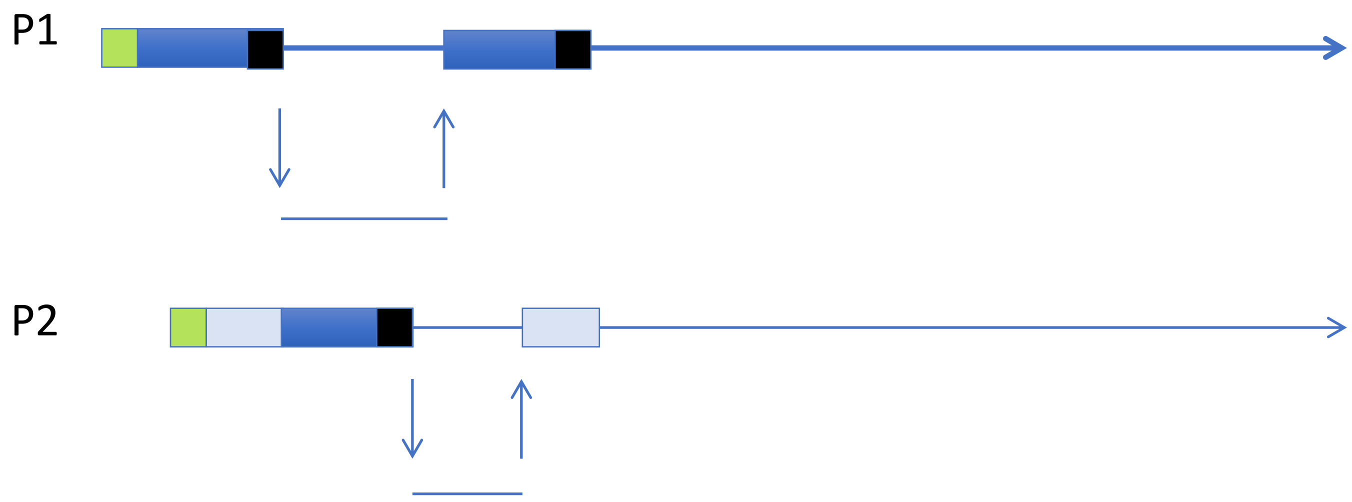
Multiprocess System



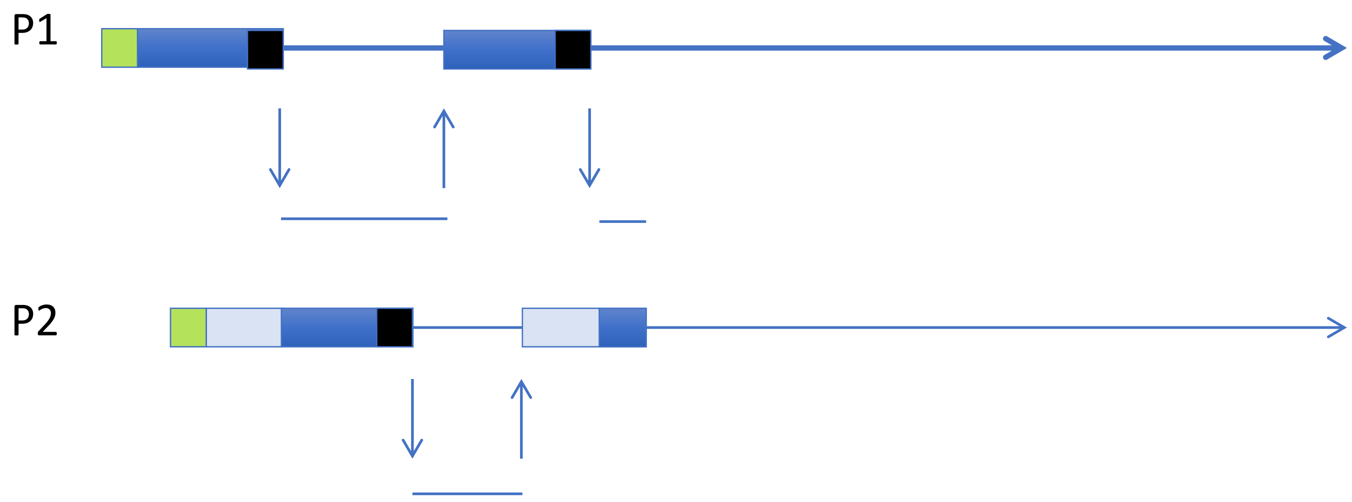
Multiprocess System



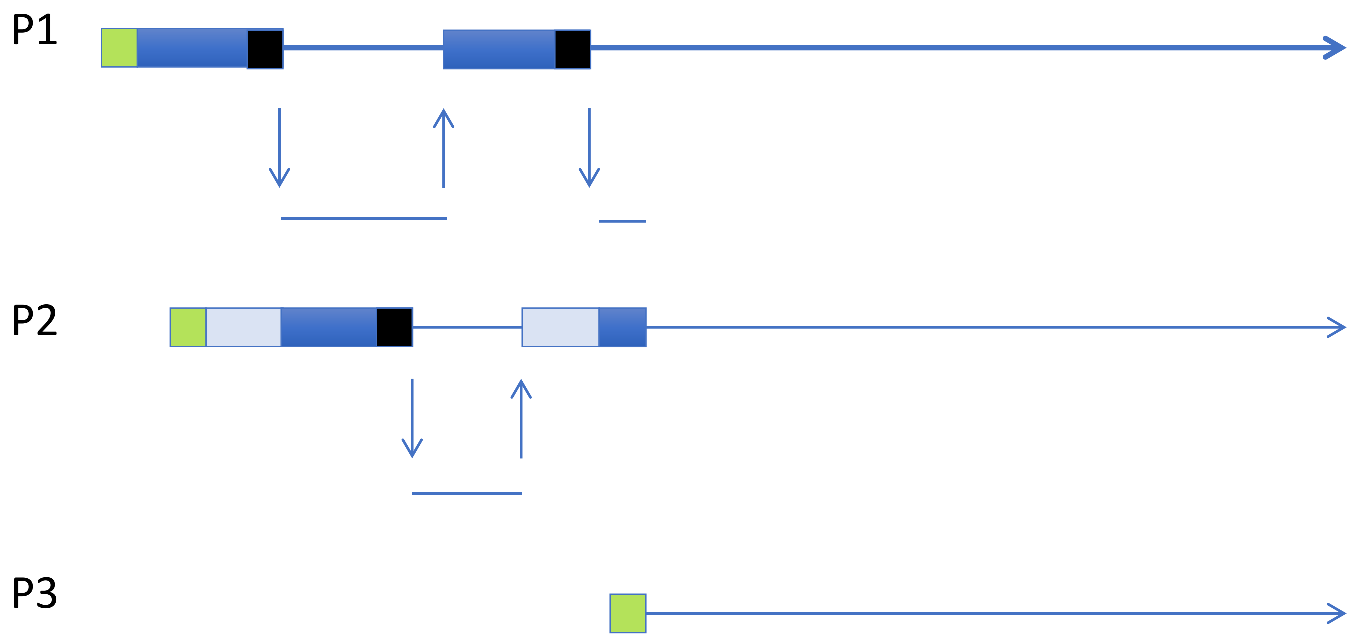
Multiprocess System



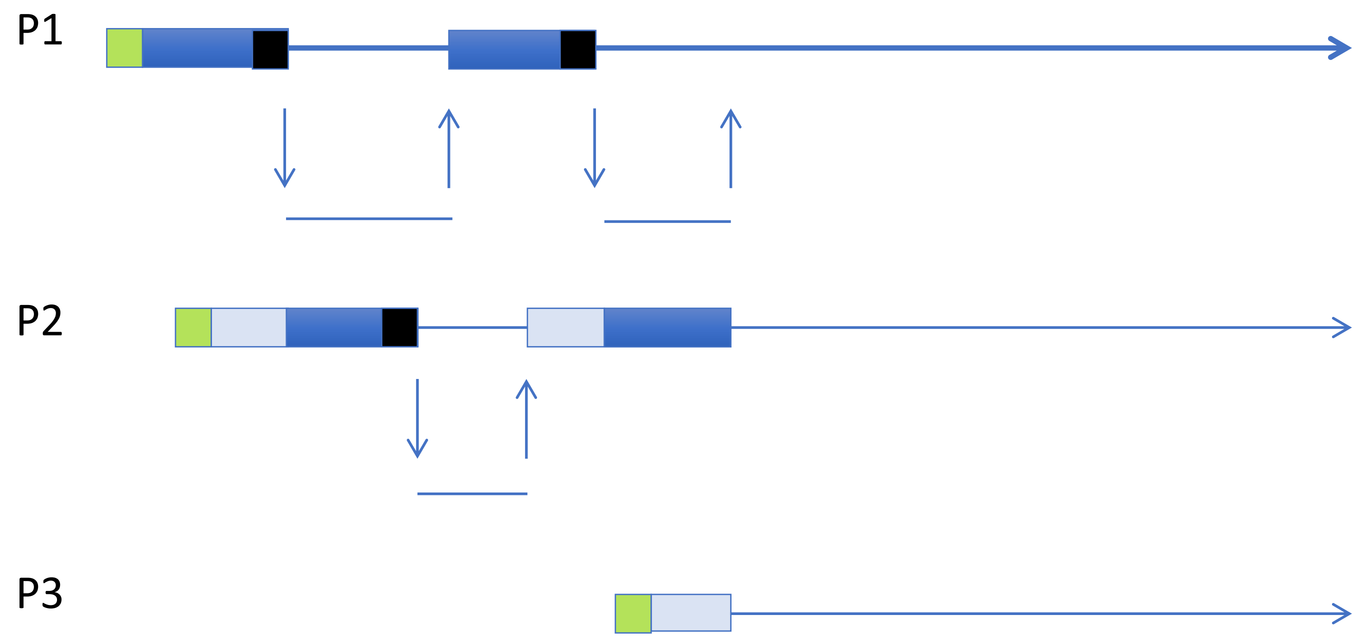
Multiprocess System



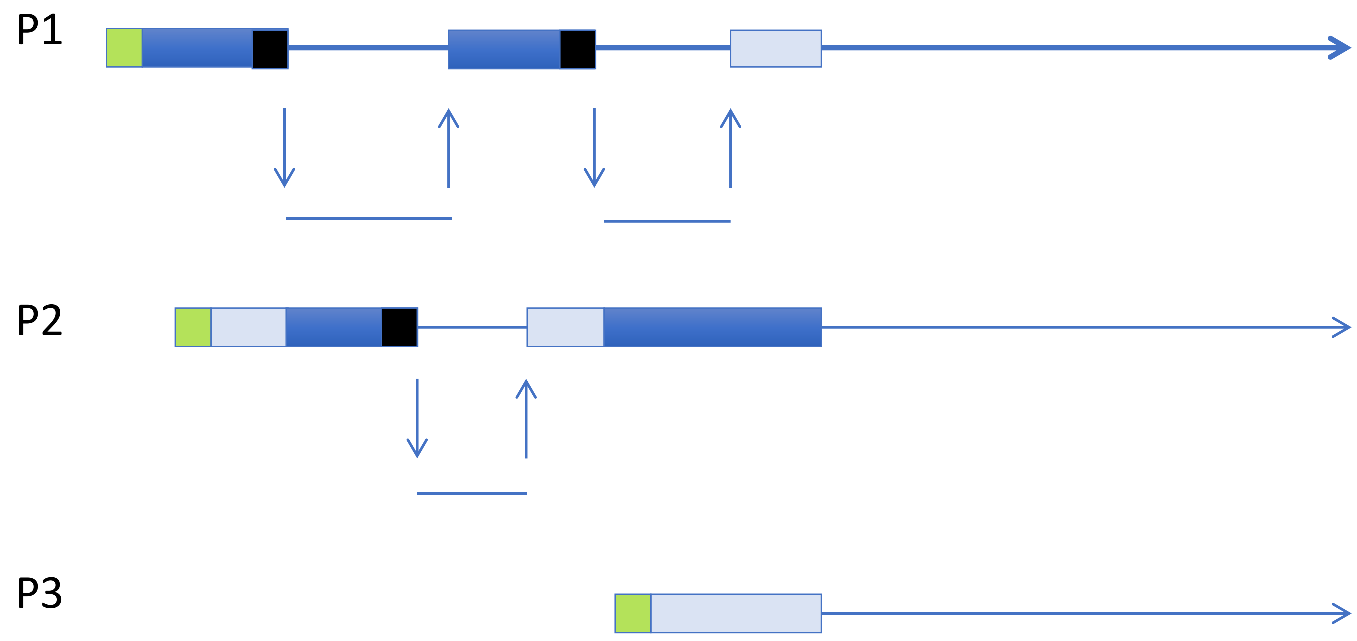
Multiprocess System



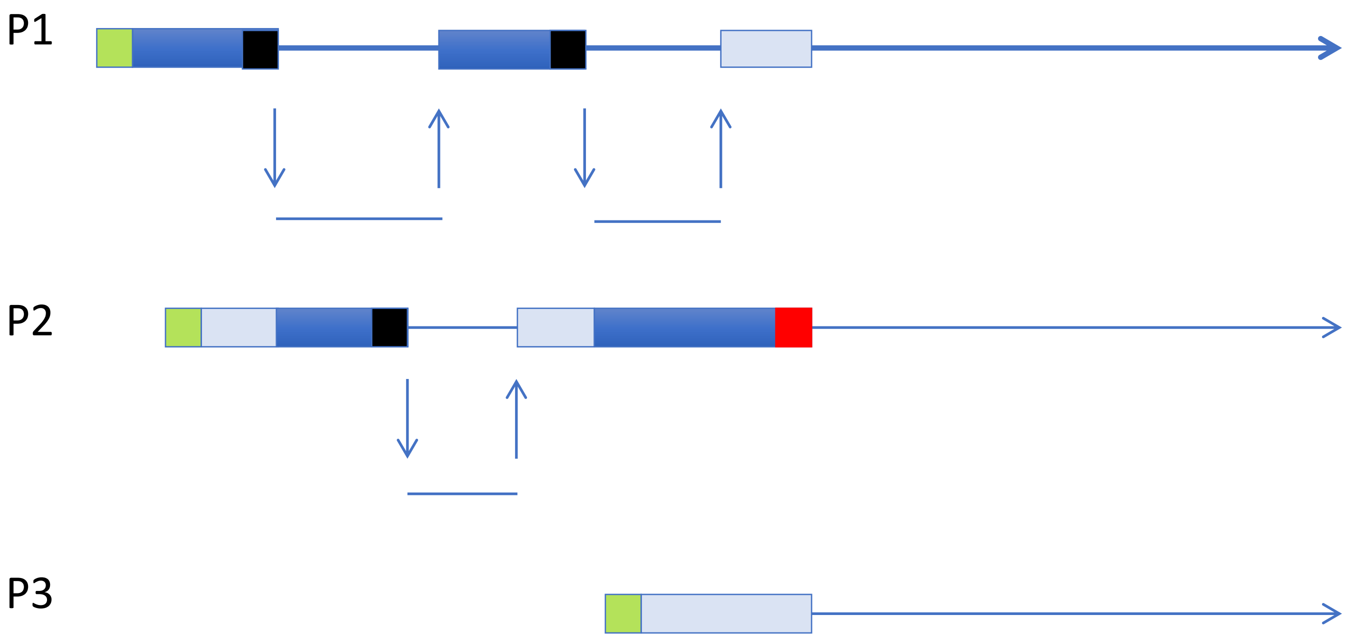
Multiprocess System



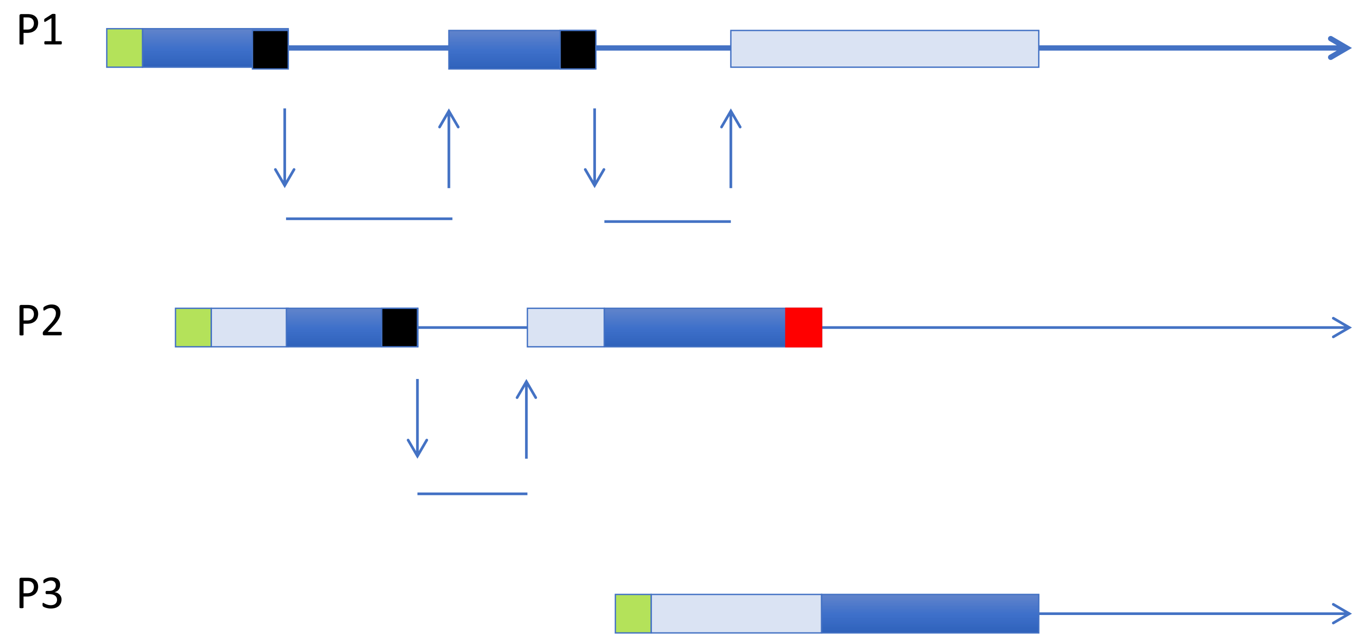
Multiprocess System



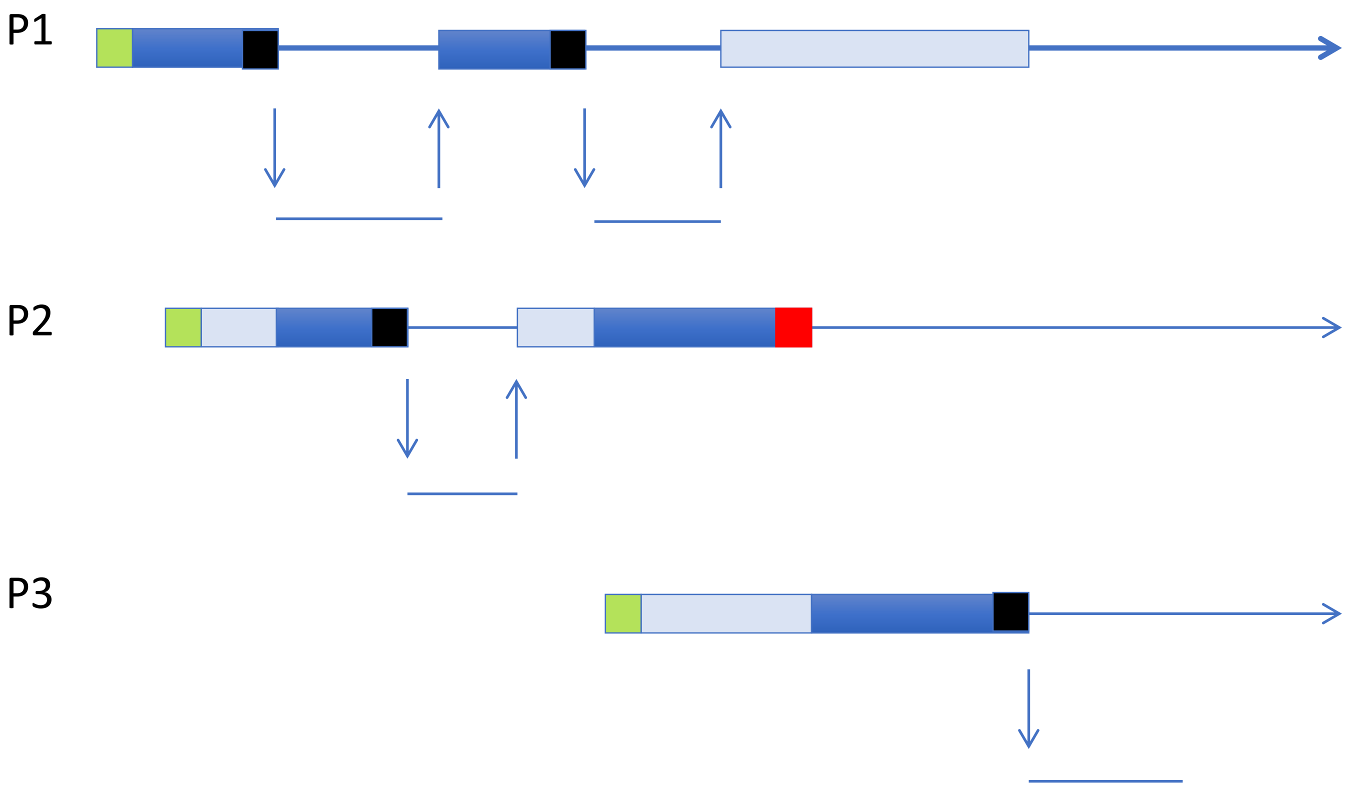
Multiprocess System



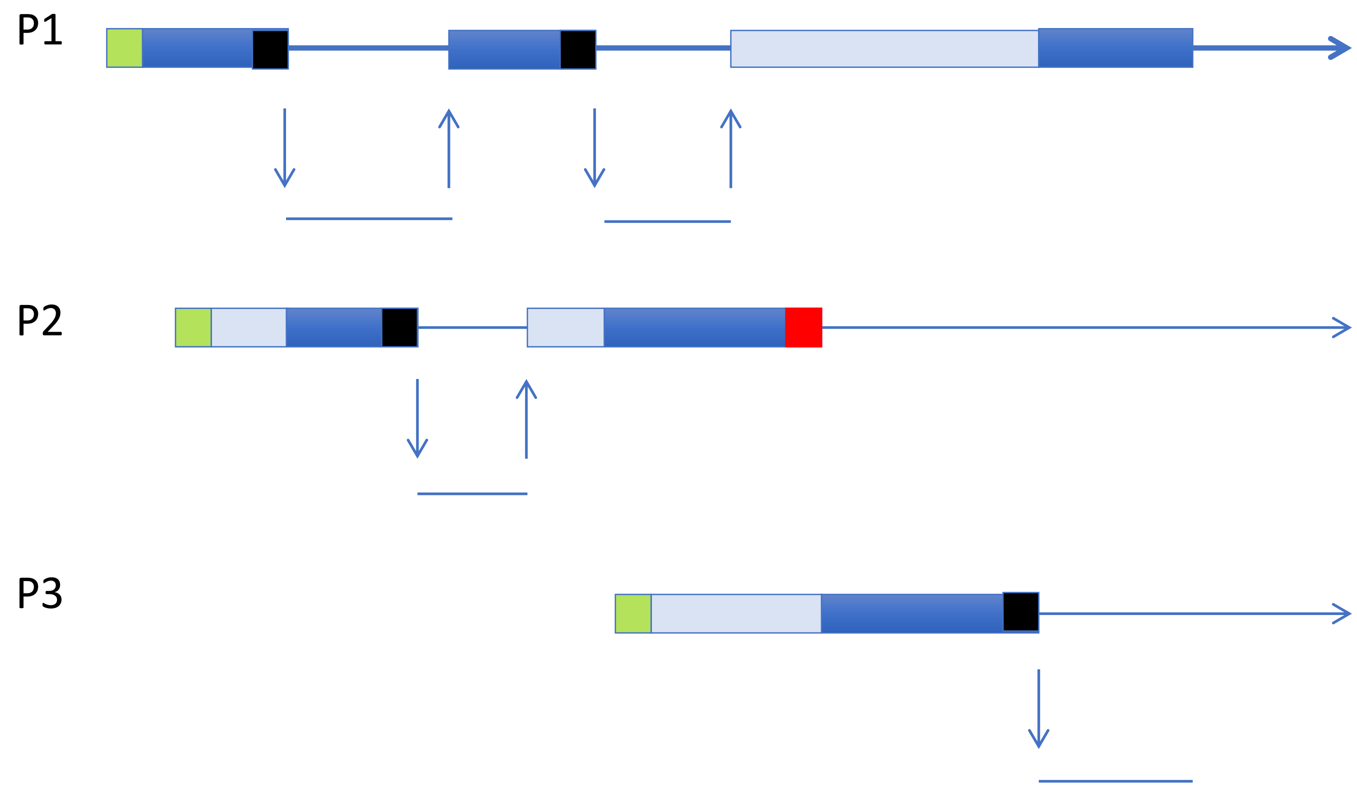
Multiprocess System



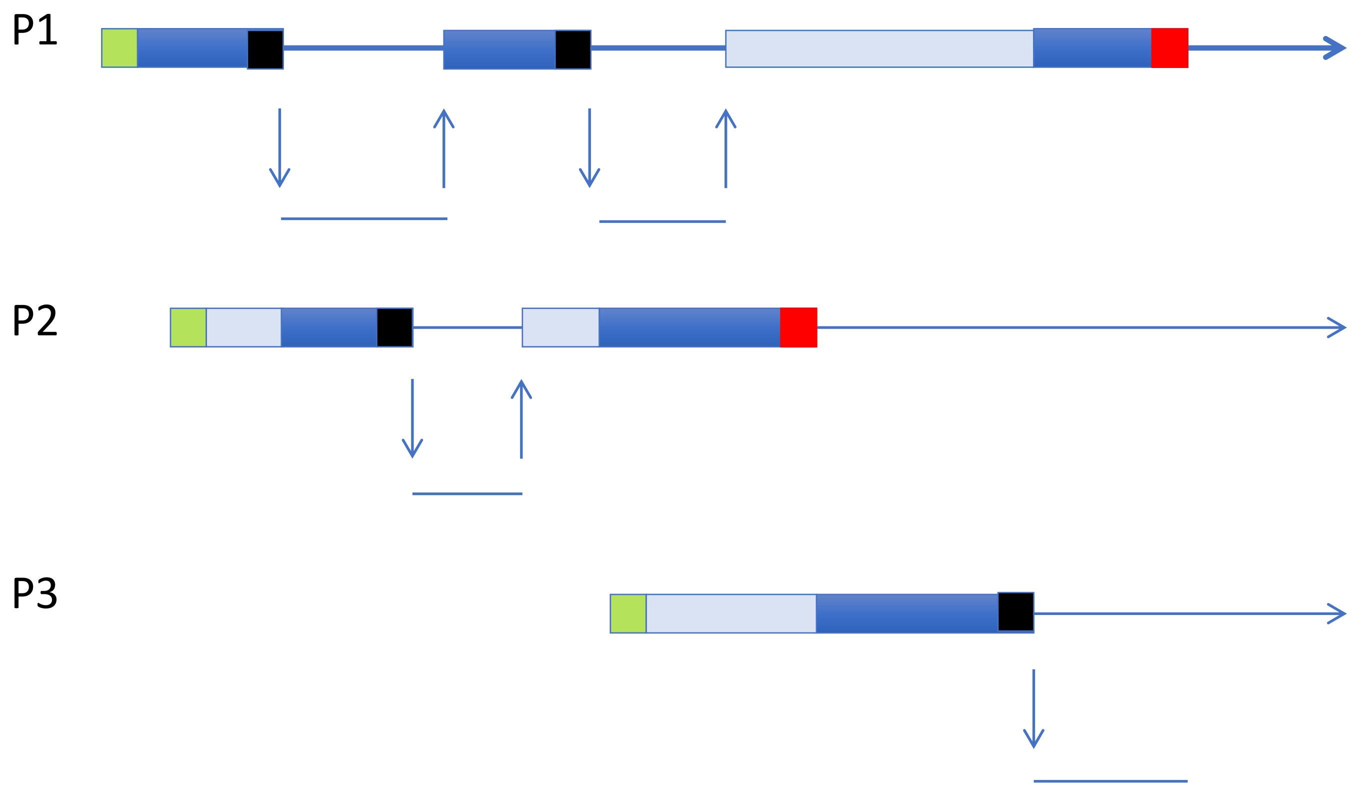
Multiprocess System



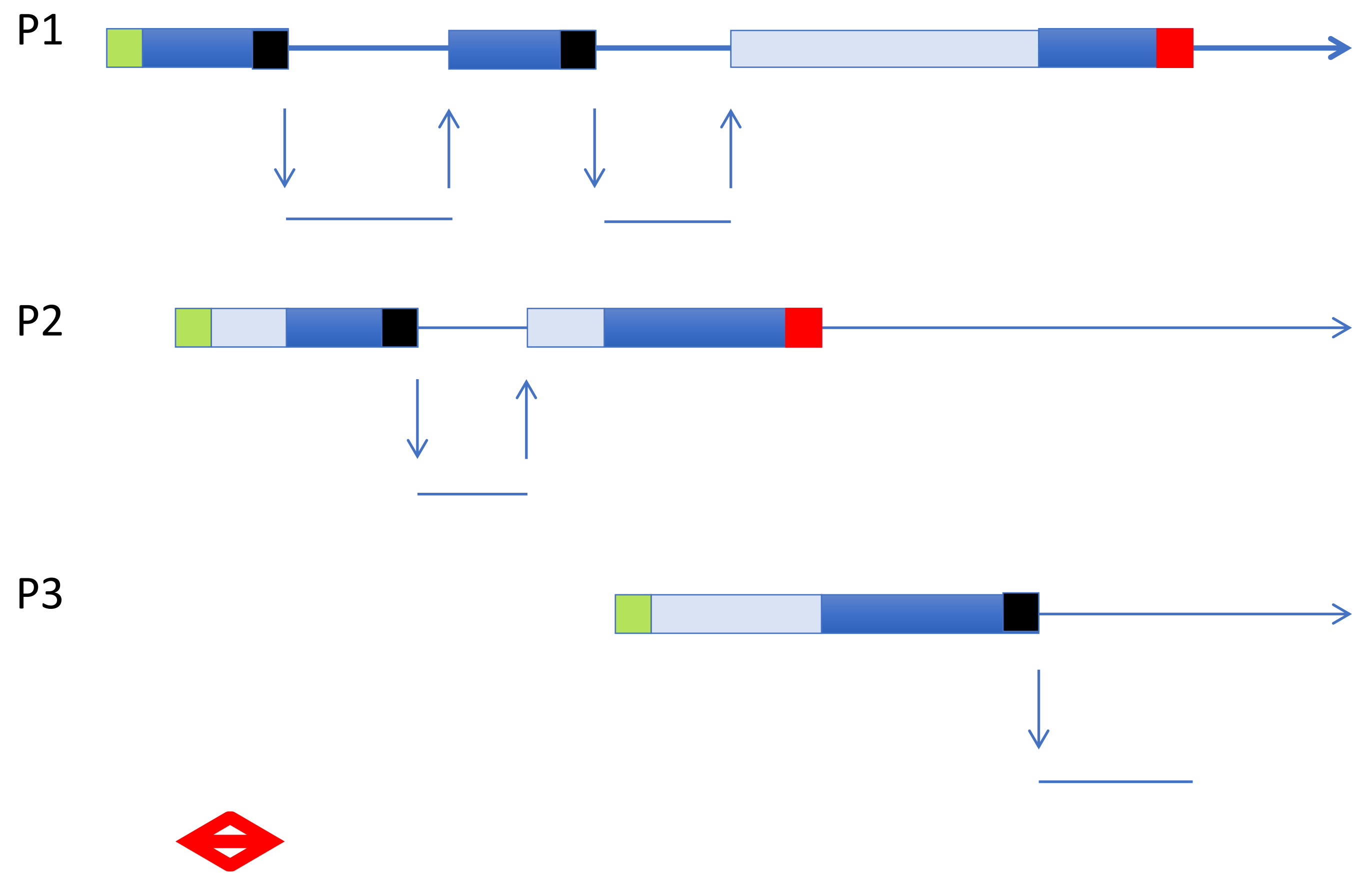
Multiprocess System



Multiprocess System

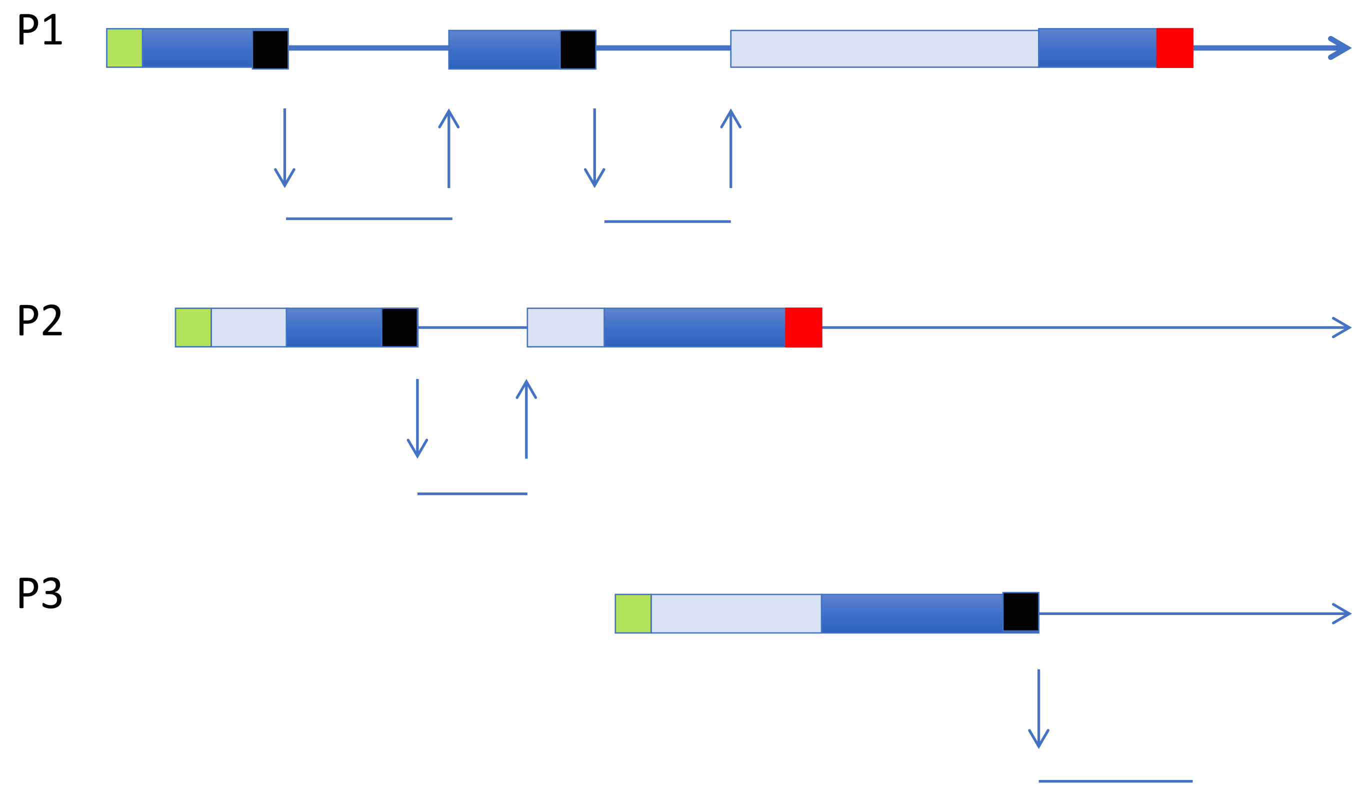


Multiprocess System



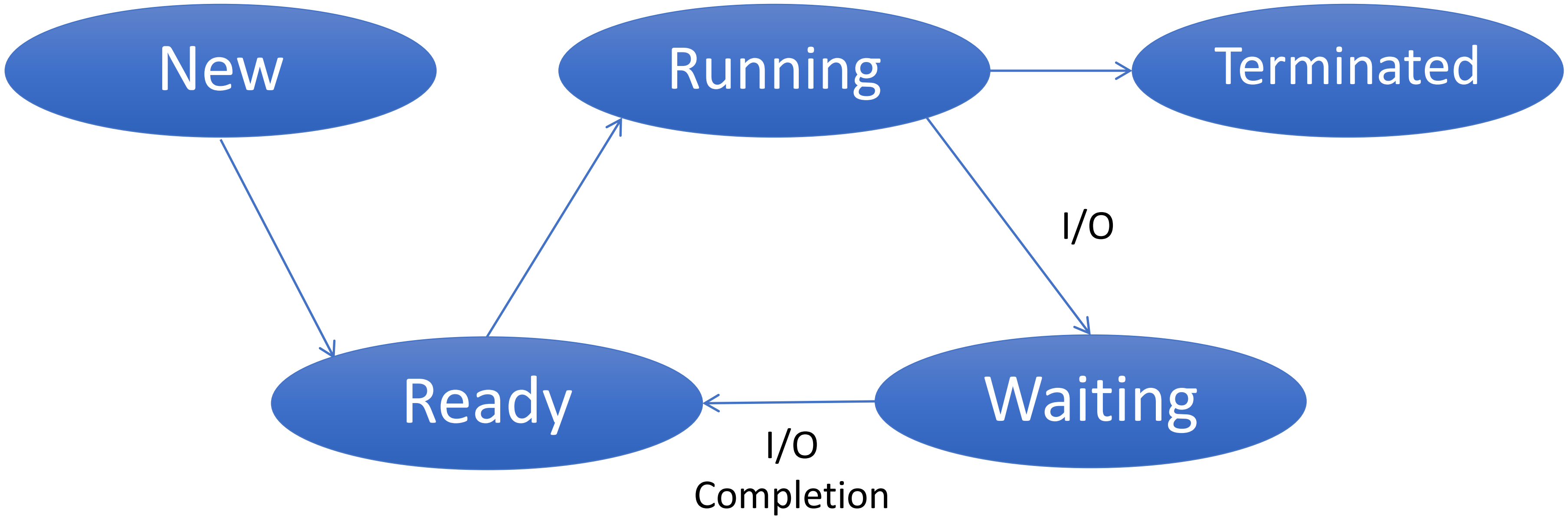
short wait time

Multiprocess System



high utilization (short CPU idle times)

Process State Diagram for Multiprocessing System



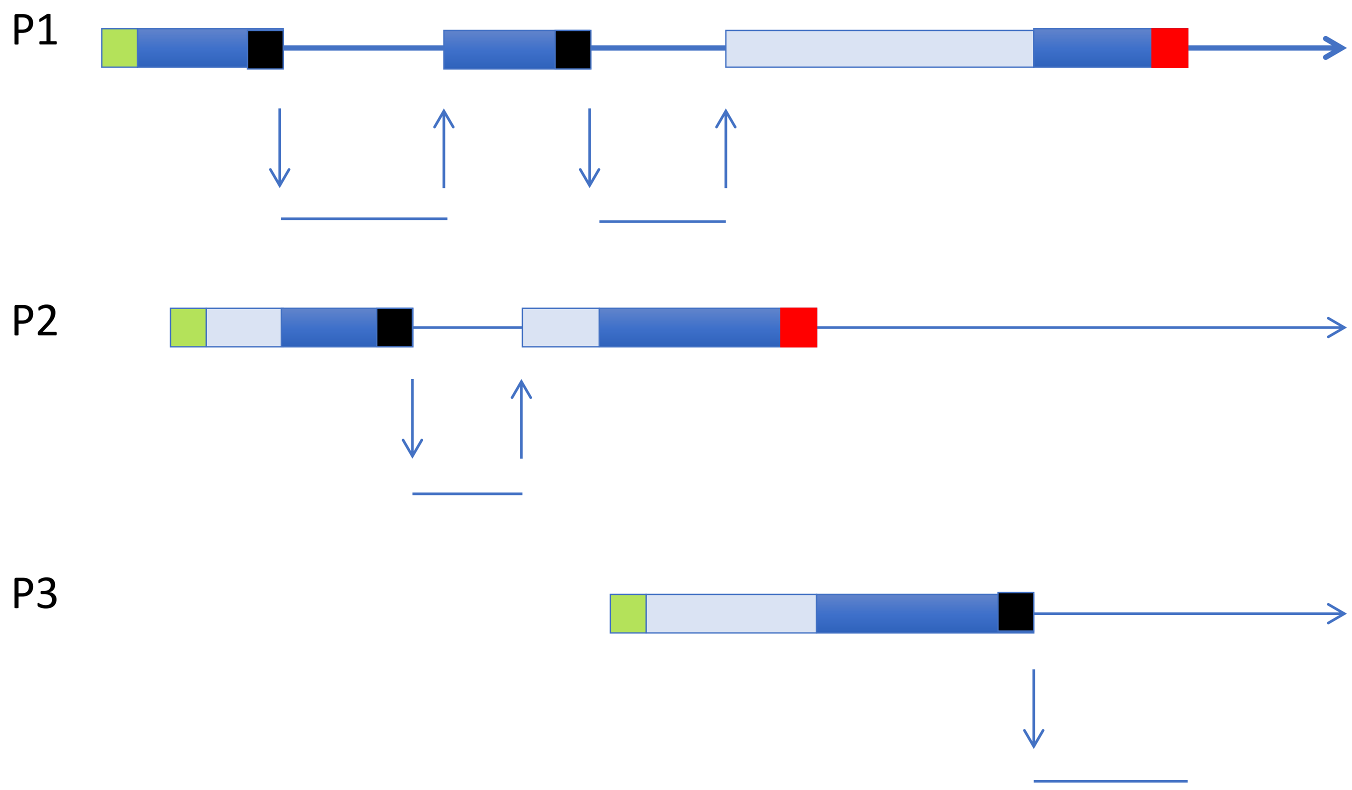
Two Important Concepts

- Process switch
- Process scheduling

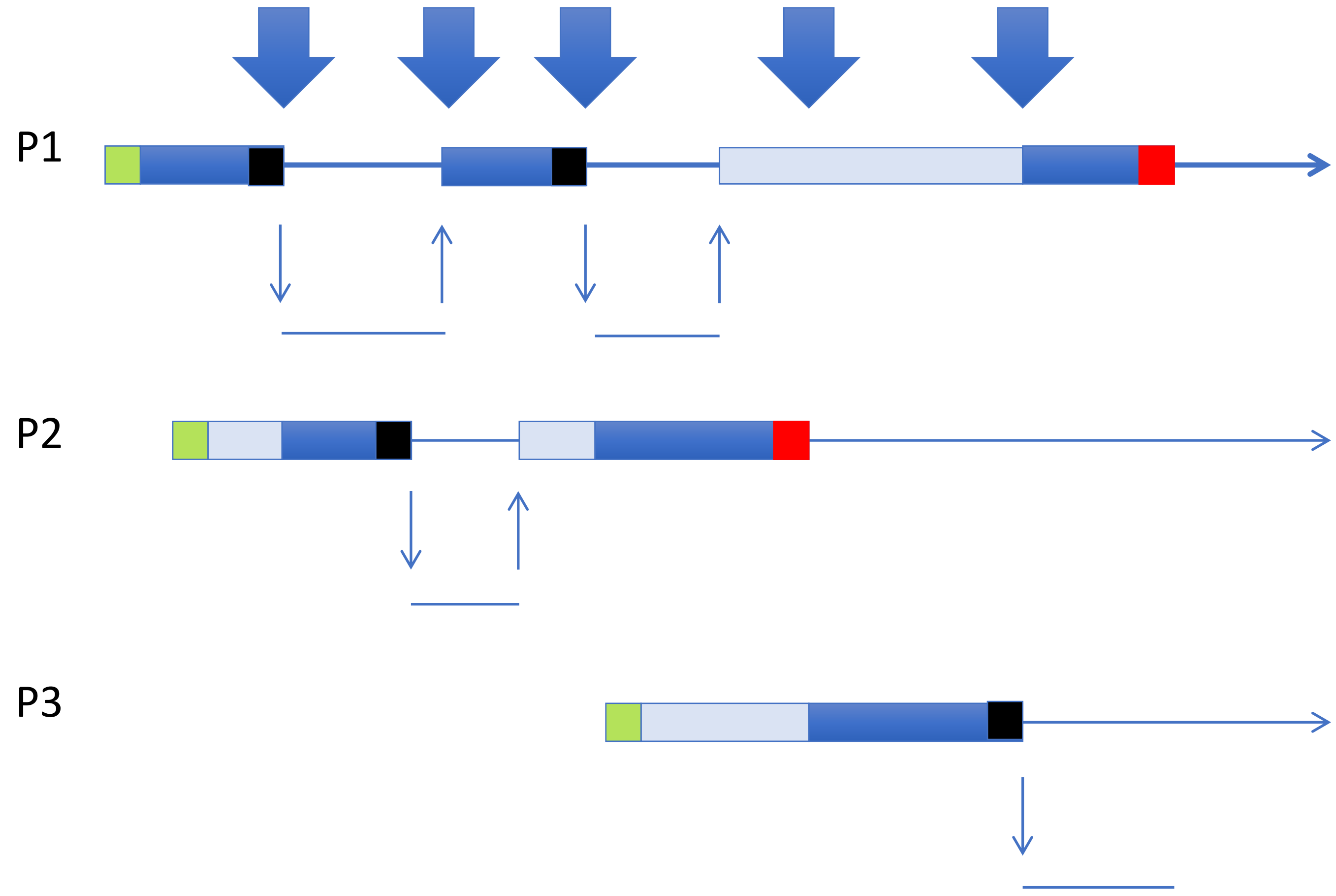
Process Switch

- Switch from one process running on the CPU to another process
- Such that you can later switch back to the process currently holding the CPU

Process Switch – where do we switch?



Answer: Process Switch



Process Switch Implementation

- Process consists of:
  - Code (including libraries)
  - Stack
  - Heap
  - Registers (including PC)
  - MMU info (ignore for now)

Process Switch Implementation

- Process:
  - Code
  - Stack
  - Heap
  - Registers
  - MMU info

Resides in process-private locations

Resides in shared locations
(ignore for now)

Process Switch P1 ➔ P2

- Save registers(P1) to somewhere
- Restore registers(P2) from somewhere

- Where to save to and restore from?

Process Control Block

- Kernel must remember processes
- Each process has a process control block (PCB)
- Process control block contains
  - Process identifier (unique id)
  - Process state
  - *Space to support process switch (save area)*
- Process Control Block Array
  - Indexed by hash(pid)

Process Switch P1 ➔ P2

- Save registers ➔ PCB[P1].SaveArea
- Restore PCB[P2].SaveArea ➔ registers

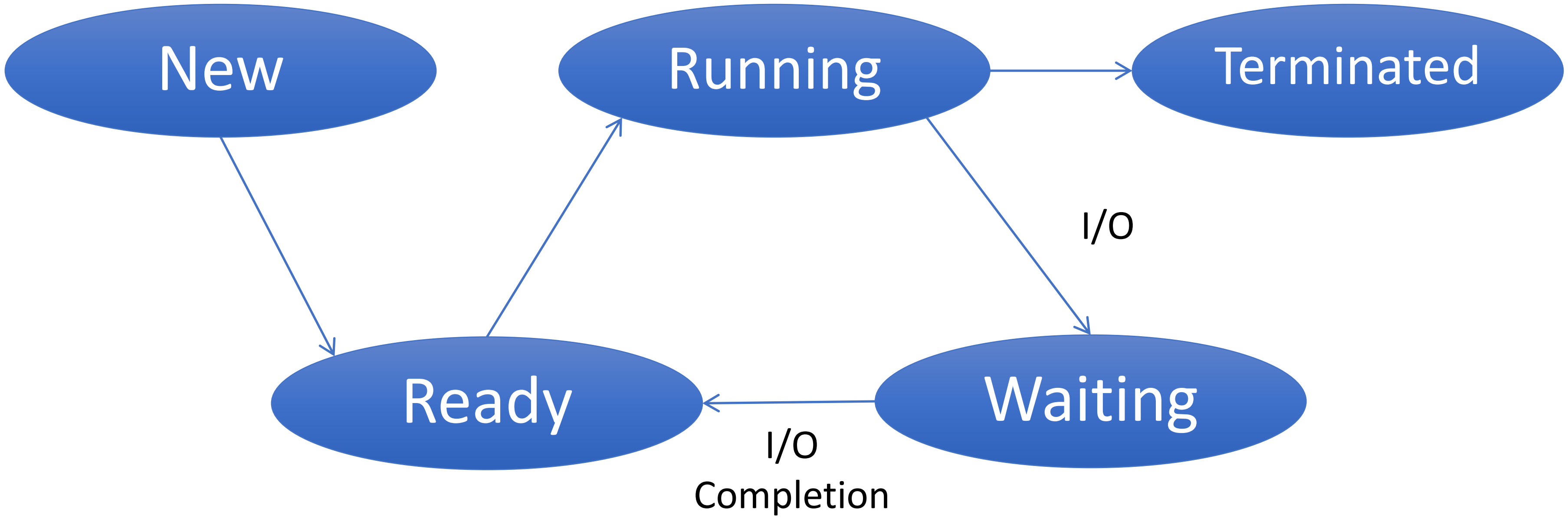
Process Switch - Caveat

- A process switch is an expensive operation!
- Requires saving and restoring lots of stuff
  - Not just registers
  - Also MMU information
- Has to be implemented very efficiently
- Has to be used with care

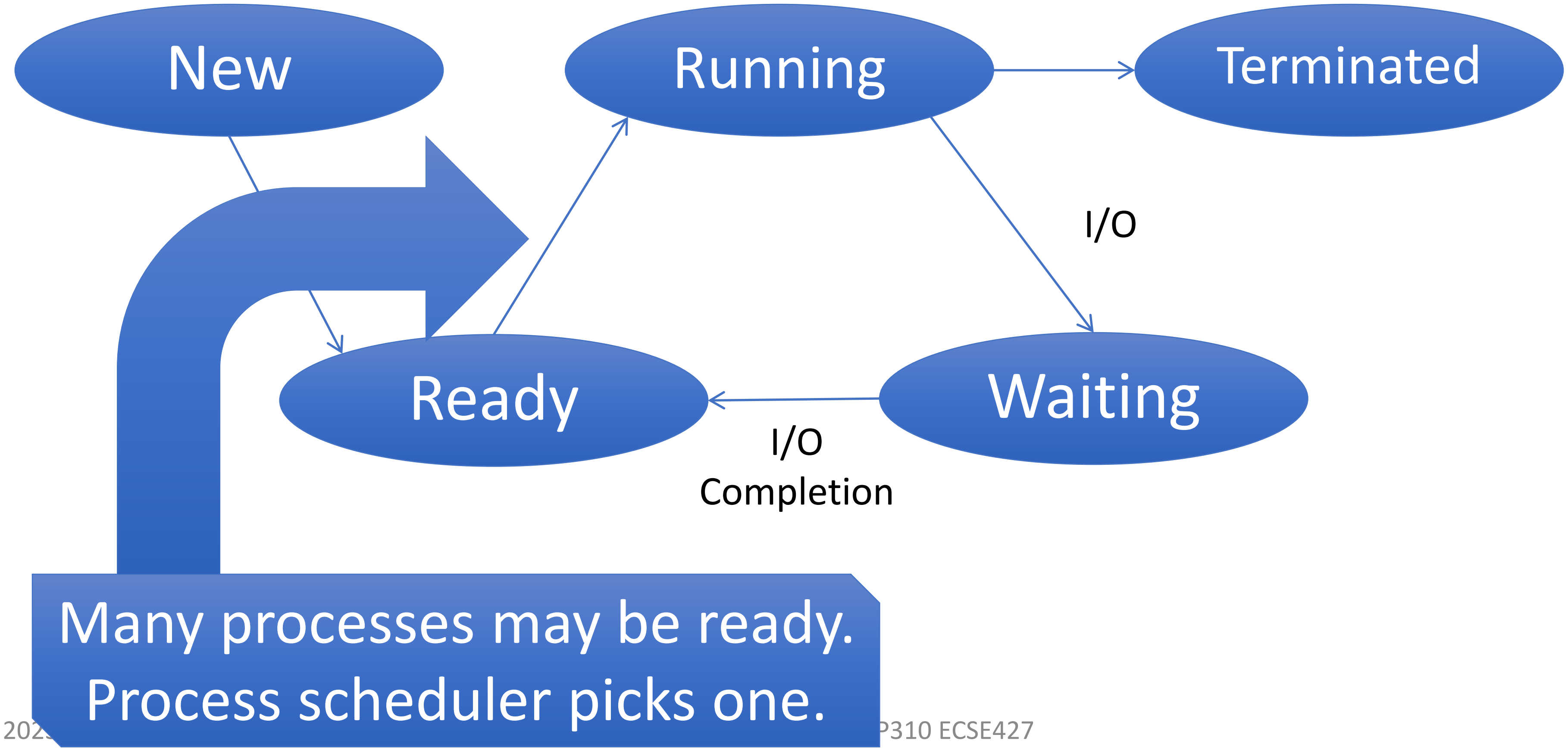
Two Important Concepts

- Process switch
- ***Process scheduling***

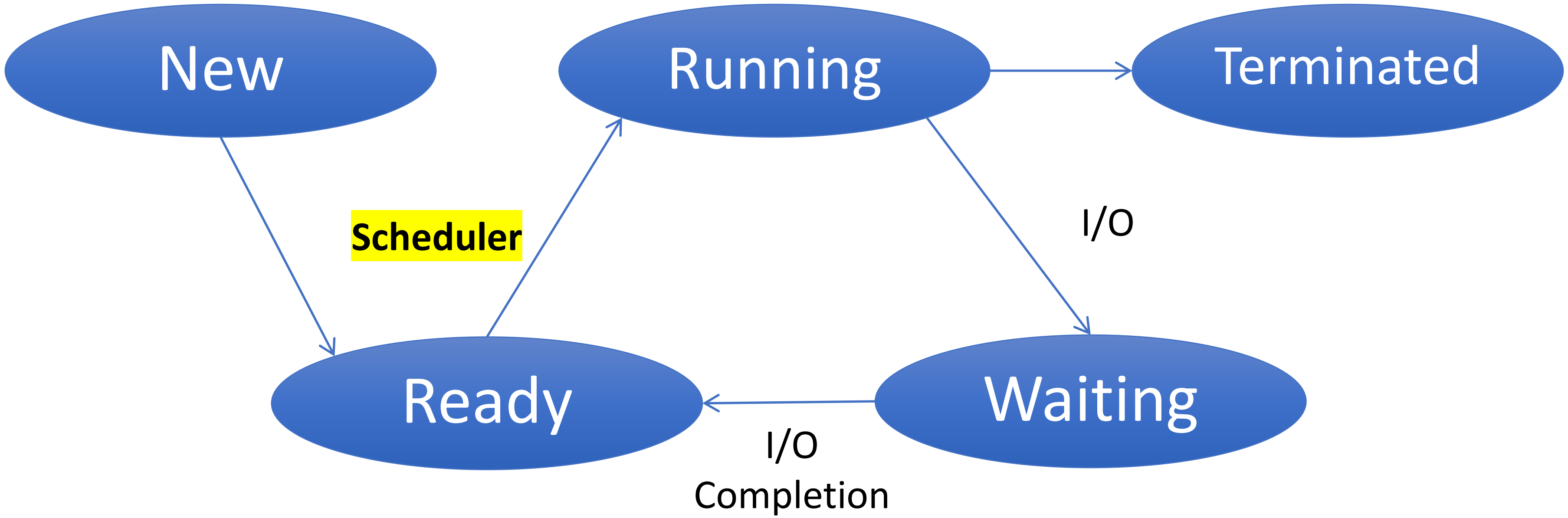
Process Scheduling



Process Scheduling



Process Scheduling

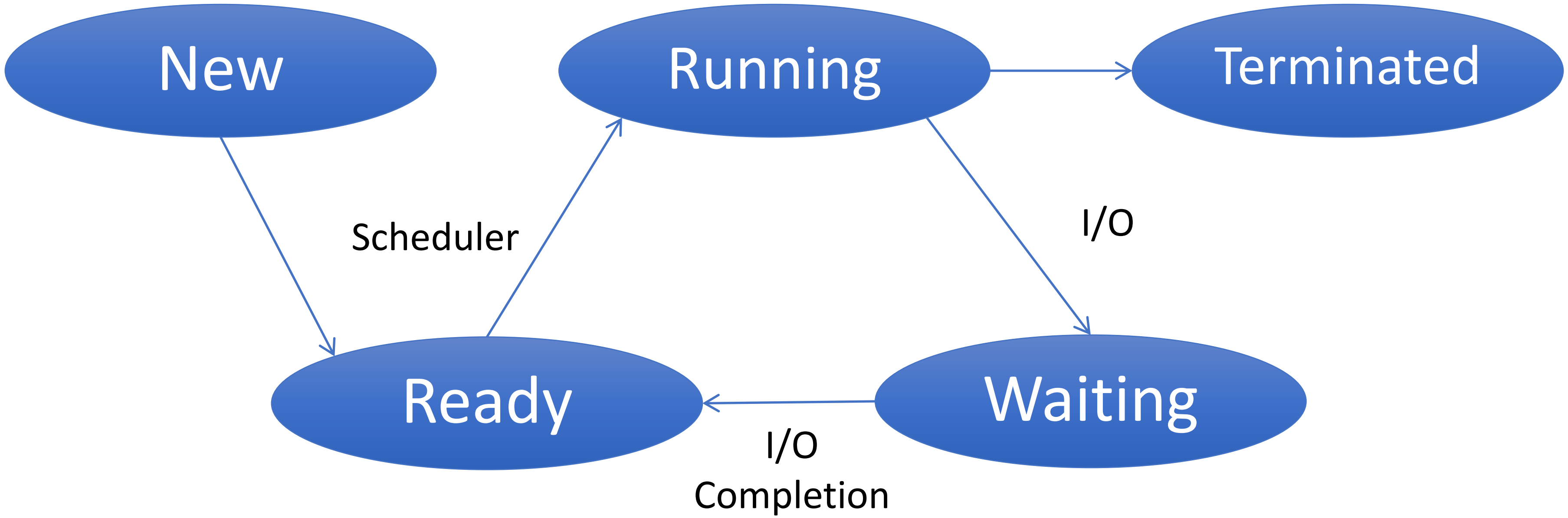


What is the role of the scheduler in an OS?

- Think of scheduler as managing a queue
- Process ready: insert it into queue
  - According to scheduling policy
- Scheduling decision: run head of queue

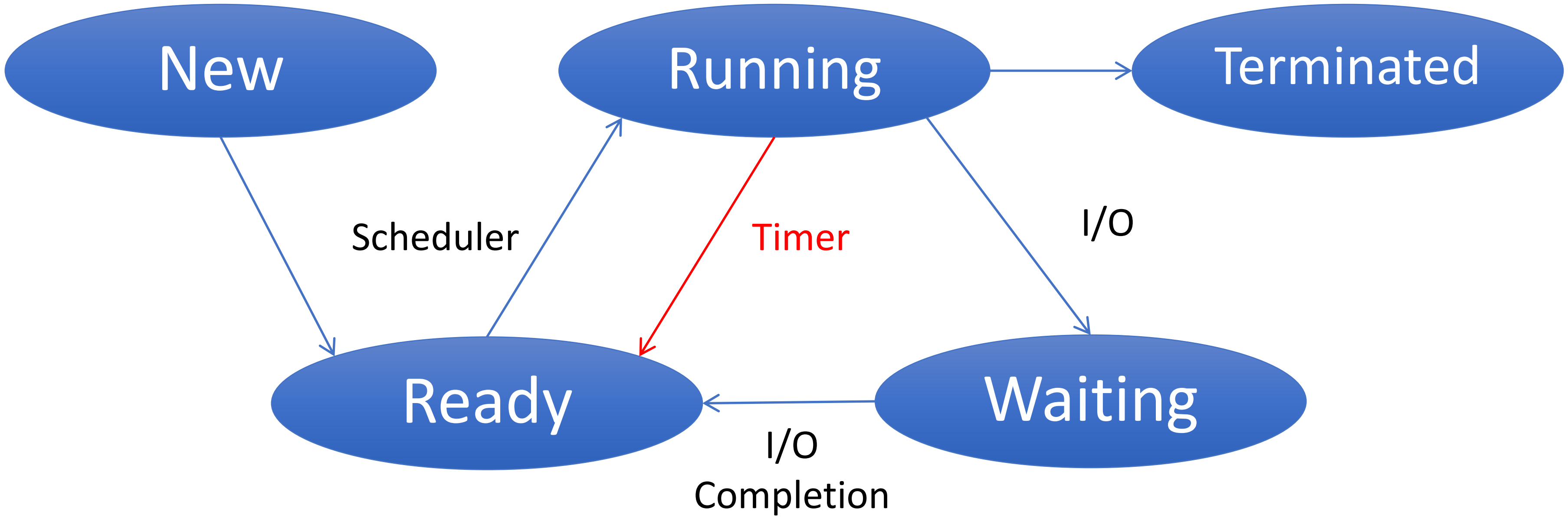
- Not always implemented this way!!

Problem



A process could run forever, locking all other processes out

Solution


New
Running
Terminated
Scheduler
Timer
I/O
Ready
Waiting
I/O
Completion

Preemptive vs Non-preemptive Scheduler

- Non-preemptive:
  - Process only voluntarily relinquishes CPU
- Preemptive
  - Process may be forced off CPU

Advantages - Disadvantages

- Non-preemptive

- Preemptive

- Intermediate solutions are possible

Advantages - Disadvantages

- Non-preemptive
  - Process can monopolize CPU
  - Only useful in special circumstances
- Preemptive
  - Process can be thrown out at any time
  - Usually not a problem, but sometimes it is
- Intermediate solutions are possible

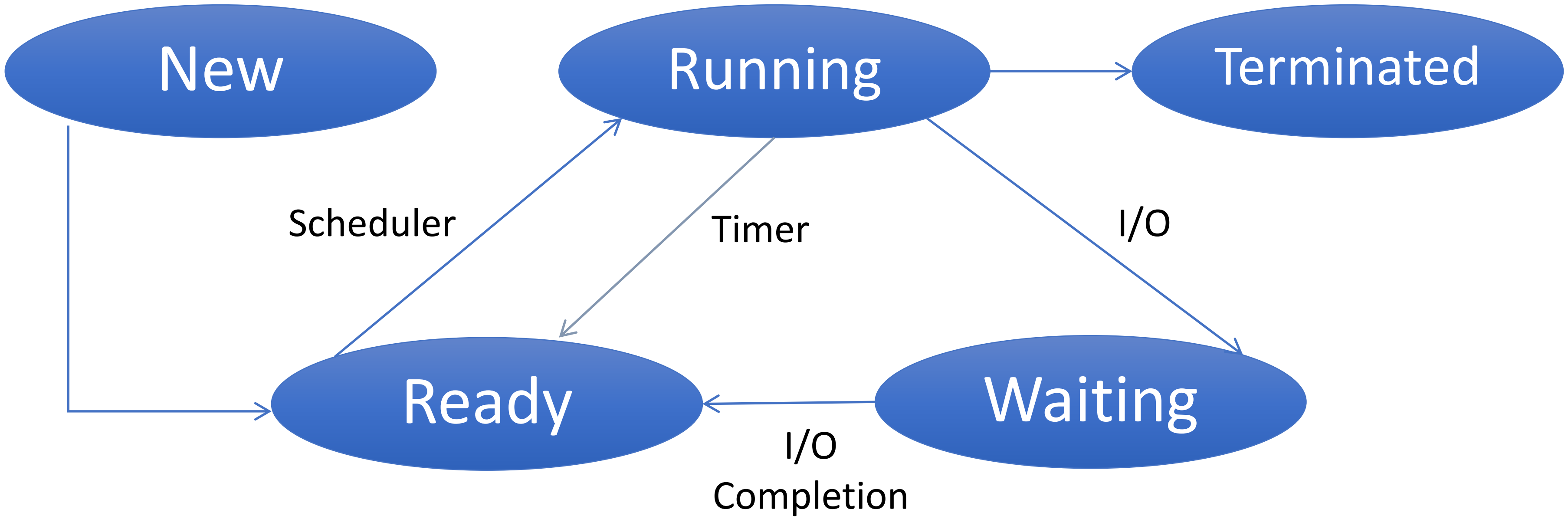
Process Scheduling Implementation

- Remember running process
- Maintain sets of queues
  - (CPU) ready queue
  - I/O device queue (one per device)
- PCBs sit in queues

How does the Scheduler run?

- Scheduler is part of the kernel
- How does kernel run?

How does Scheduler run?



The scheduler runs when
1) process starts or terminates (system call)
2) running process performs an I/O (system call)
3) I/O completes (I/O interrupt)
4) timer expires (timer interrupt)

How does the Scheduler Run?

- At end of handlers for
  - System calls
  - Interrupts
  - Traps
- Scheduler runs: decides on process to run
- Switches to a new process
- Sets another timer

Scheduling Algorithm

- Decides which ready process gets to run

What makes a good scheduling algorithm?

What makes a good scheduling algorithm?

- It depends …

Scheduling Performance Metrics

- Minimize turnaround time
  - Do not want to wait long for job to complete
  - Completion_time – arrival_time
- Minimize response time
  - Schedule interactive jobs promptly so users see output quickly
  - Initial_schedule_time – arrival_time
- Minimize waiting time
  - Do not want to spend much time in Ready queue
- Maximize throughput
  - Want many jobs to complete per unit of time
- Maximize resource utilization
  - Keep expensive devices busy
- Minimize overhead
  - Reduce number of context switches
- Maximize fairness
  - All jobs get same amount of CPU over some time interval

Answer: Scheduling Performance Metrics

- Minimize turnaround time
  - Do not want to wait long for job to complete
  - Completion_time – arrival_time
- Minimize response time
  - Schedule interactive jobs promptly so users see output quickly
  - Initial_schedule_time – arrival_time
- Minimize waiting time
  - Do not want to spend much time in Ready queue
- Maximize throughput
  - Want many jobs to complete per unit of time
- Maximize resource utilization
  - Keep expensive devices busy
- Minimize overhead
  - Reduce number of context switches
- Maximize fairness
  - All jobs get same amount of CPU over some time interval

Conflicting goals

→ A good scheduling algorithm depends on what we want to run

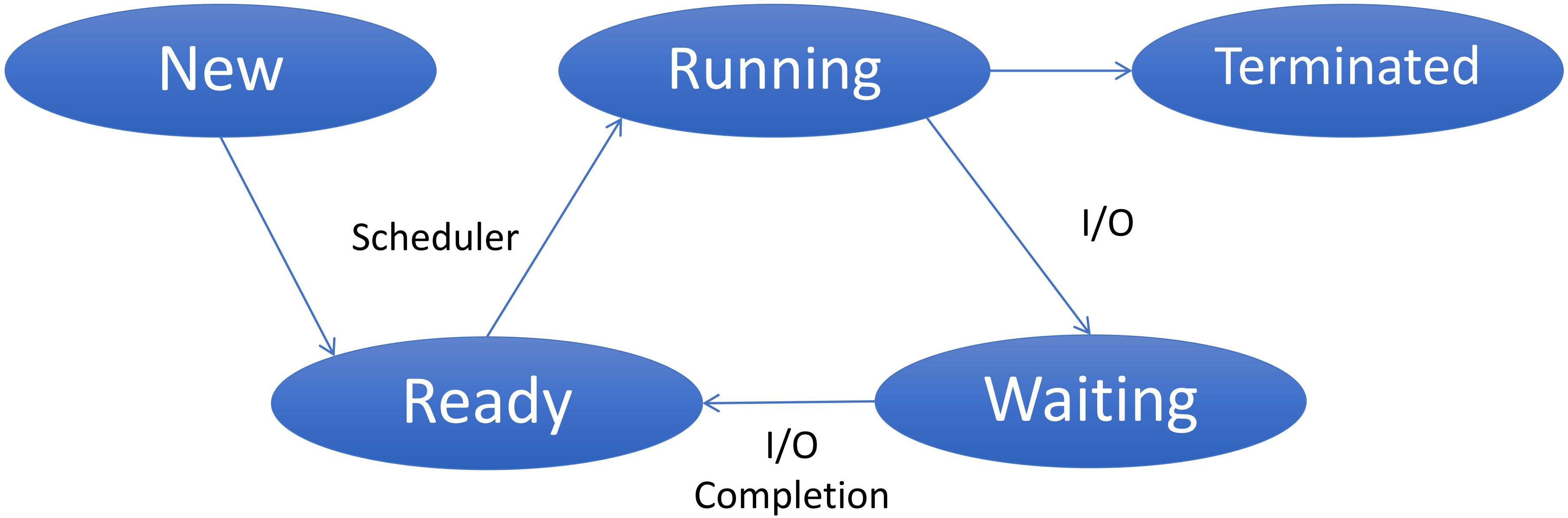
→ Most of the time, scheduler does not know in advance the job type and its needs

Question: Interactive vs. Batch Jobs

What makes a good scheduler for interactive/for batch?

- Interactive = you are waiting for the result
  - E.g., browser, editor, …
  - Tend to be short
- Batch = you will look at result later
  - E.g., supercomputing center, offline analysis, …
  - Tend to be long

Question: Interactive vs. Batch Jobs



Answer: What makes a good scheduler for interactive?

- Short response time
- Response time = wait from ready → running
  - Initial_schedule_time – arrival_time

Answer: What makes a good scheduler for batch?

- High throughput
- Throughput = number of jobs completed
  - Minimize scheduling overhead
    - Reduce number of ready → running switches

Issue: Response Time vs. Throughput

- Conflicting goals
- From throughput perspective
  - Scheduler is overhead
  - Run scheduler as little as possible
- From response time perspective
  - Want to go quickly from ready to running
  - Run scheduler often

Scheduling policies

- First come, first served (FCFS)
- Shortest job first (SJF)
- Round robin (RR)

First come first served (FCFS)

- Process ready: insert at tail of queue
- Head of queue: “oldest” ready process
- By definition, non-preemptive

Question

- Advantages and disadvantages of FCFS?

Answer: FCFS advantages and disadvantages

- Low overhead – few scheduling events
- Good throughput
- Uneven response time – stuck behind long job
- Extreme case – process monopolizes CPU

Shortest job first (SJF)

- Process ready
  - Insert in queue according to length
- Head of queue: “shortest” process
- Can be preemptive or non-preemptive
- From now on, only consider preemptive

Question

- Advantages and disadvantages of SJF?

Answer: SJF advantages and disadvantages

- Good response time for short jobs
- Can lead to starvation of long jobs
- Difficult to predict job length

Round Robin (RR)

- Define time quantum Δ
- Process ready: put at tail of queue
- Head of queue: run for Δ time
- After Δ
  - Put running process at the tail of the queue
  - Re-schedule

Question

- Advantages and disadvantages of RR?

Answer: RR advantages and disadvantages

- Good compromise for long and short jobs
- Short jobs finish quickly (a few rounds)
- Long jobs are not postponed forever

- No need to know job length
  - Discover length by how many Δ's it needs

RR Issue – How to pick Δ

- Too small
  - Many scheduling events
  - Good response time
  - Low throughput
- Too large
  - Few scheduling events
  - Good throughput
  - Poor response time
- Typical value: ~ 10 milliseconds

Scheduling Exercise

A. Describe on a timeline the order of execution of the following five processes, with arrival and execution times as shown in the table, using the following scheduling algorithms:

1. FCFS,
2. SJF – preemptive,
3. RR with a time quantum of 1.

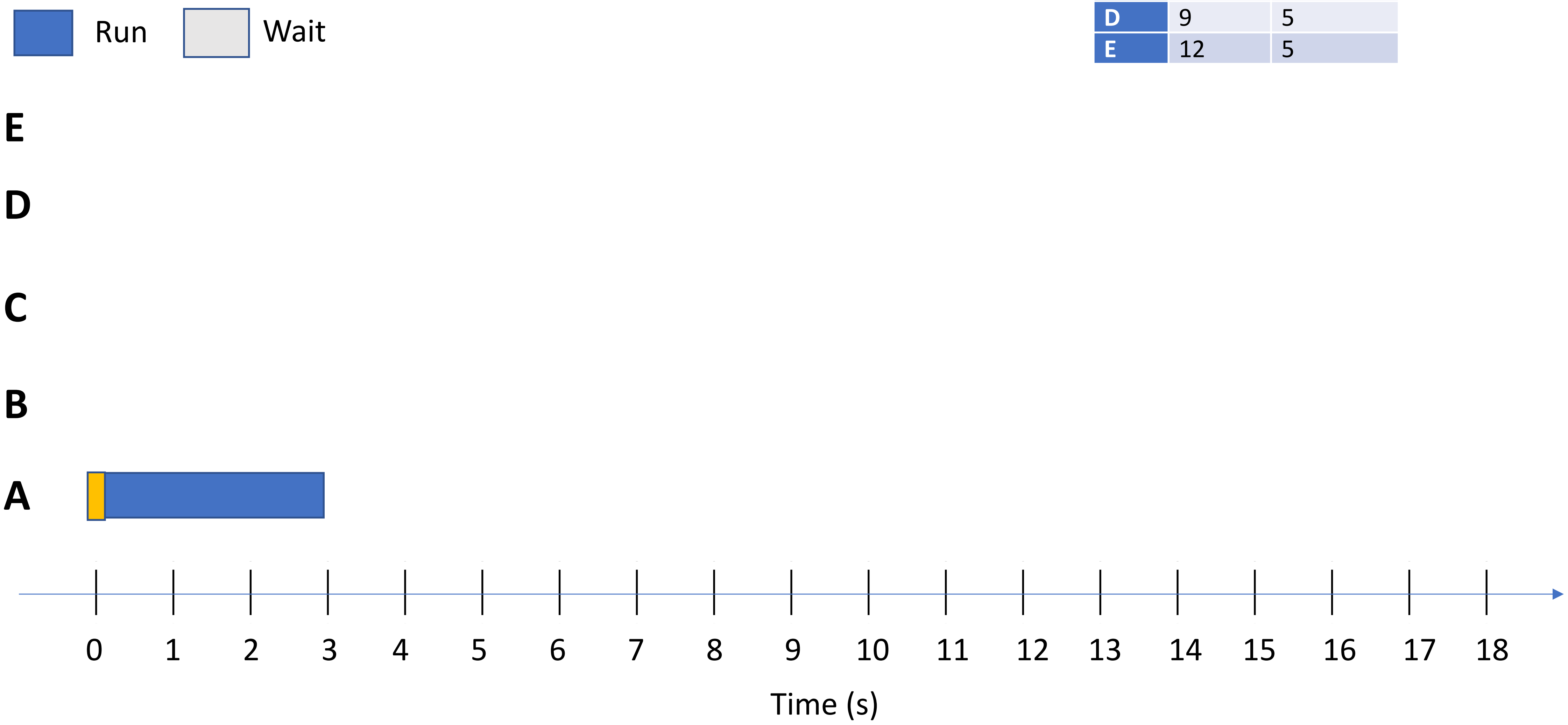
B. What is the turn-around time & response time for each process in each scenario?

Process	Arrival time	Execution time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

FCFS

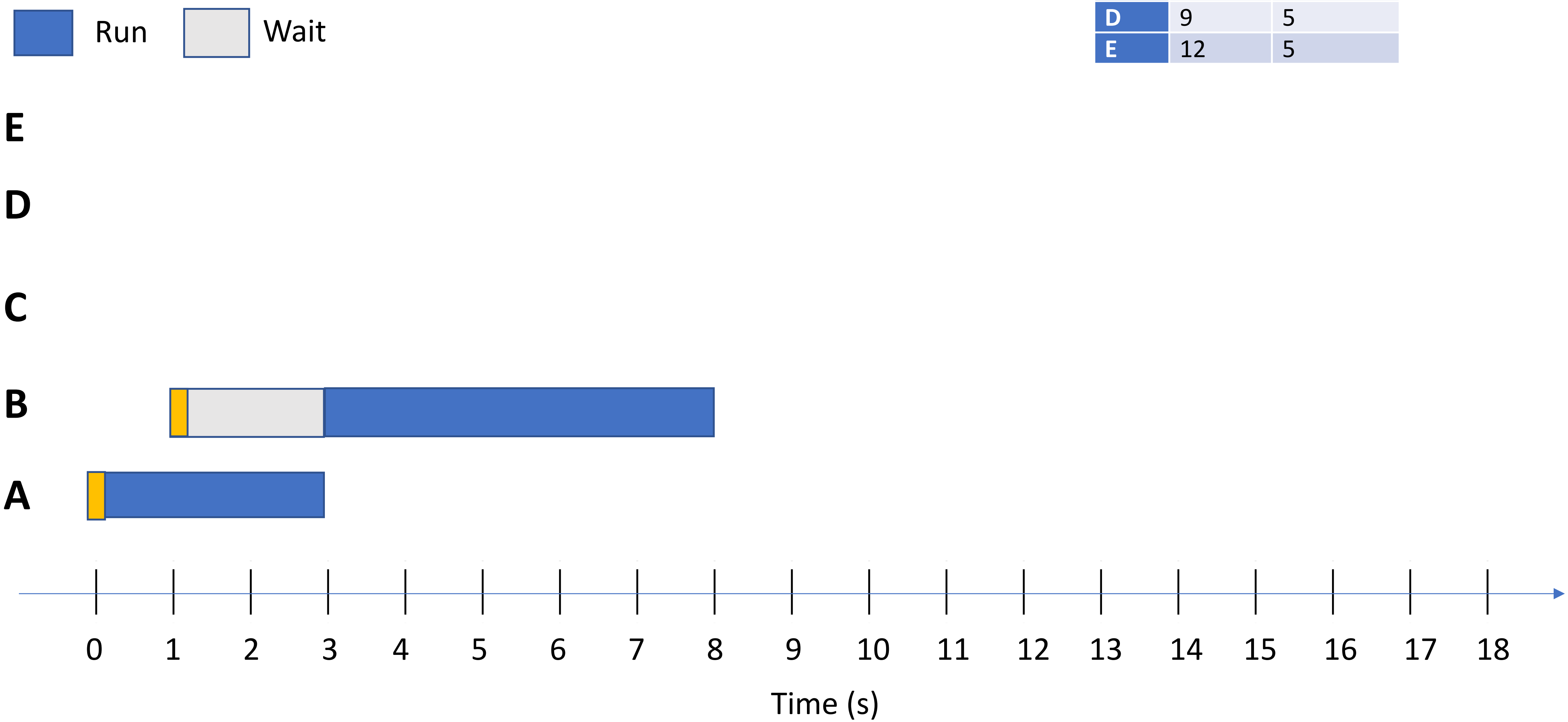
FCFS

Proc	Arrival time	Execution time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5



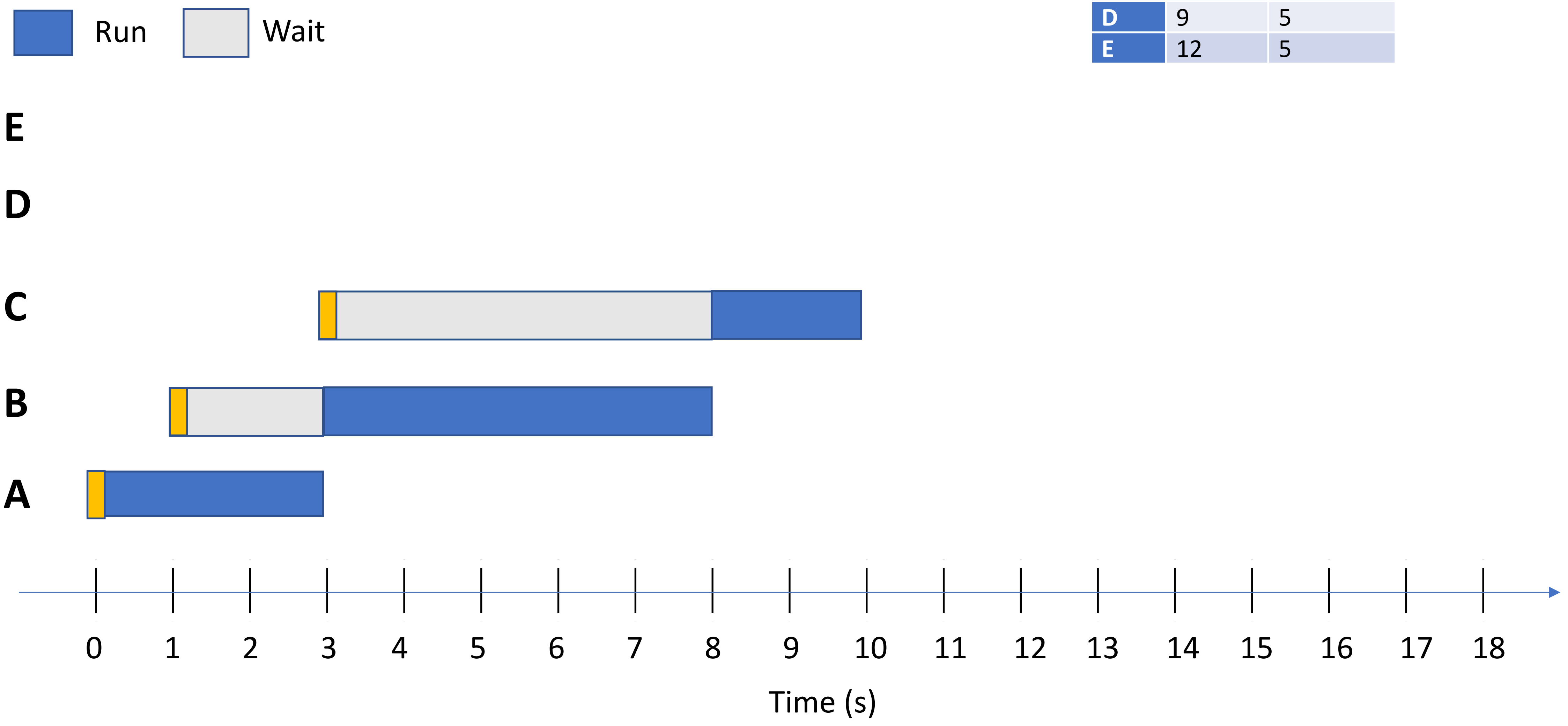
FCFS

Proc	Arrival time	Execution time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5



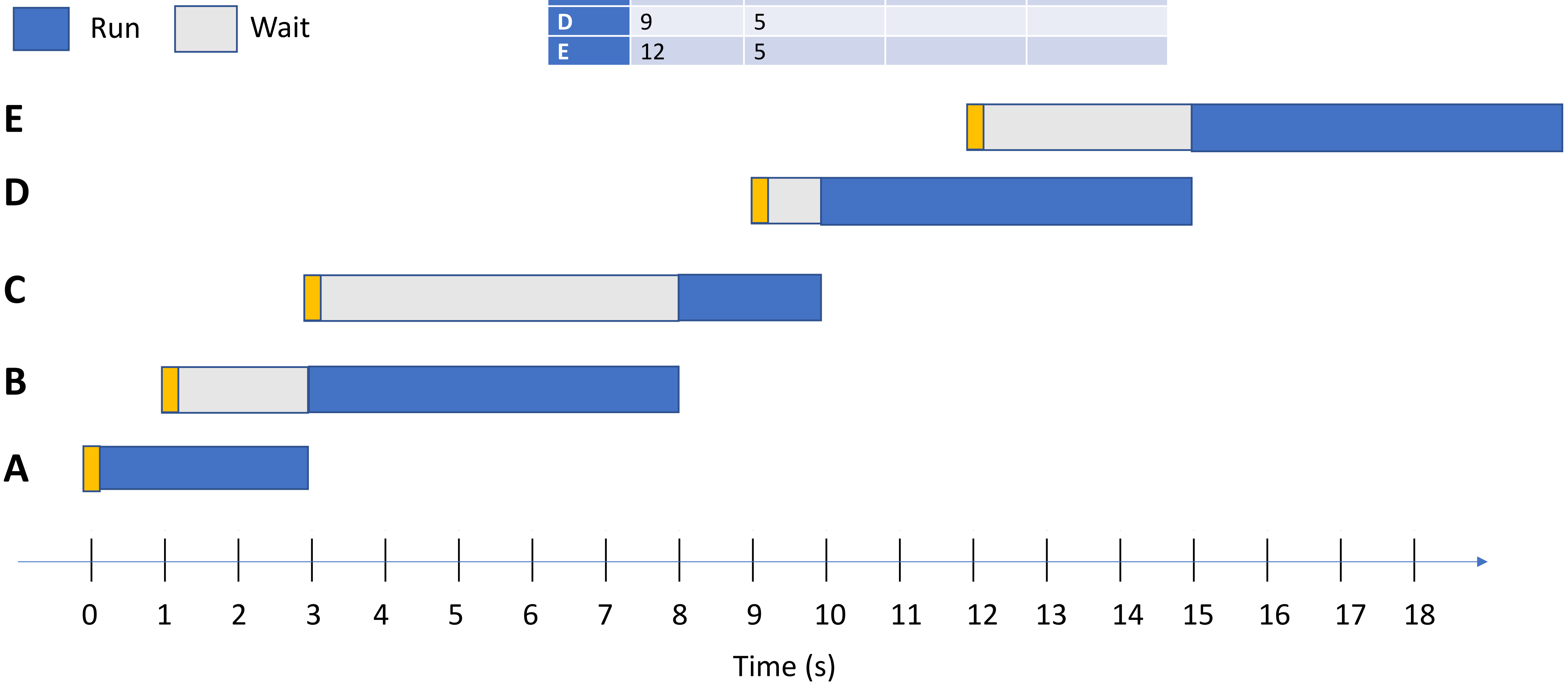
FCFS

Proc	Arrival time	Execution time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5



FCFS

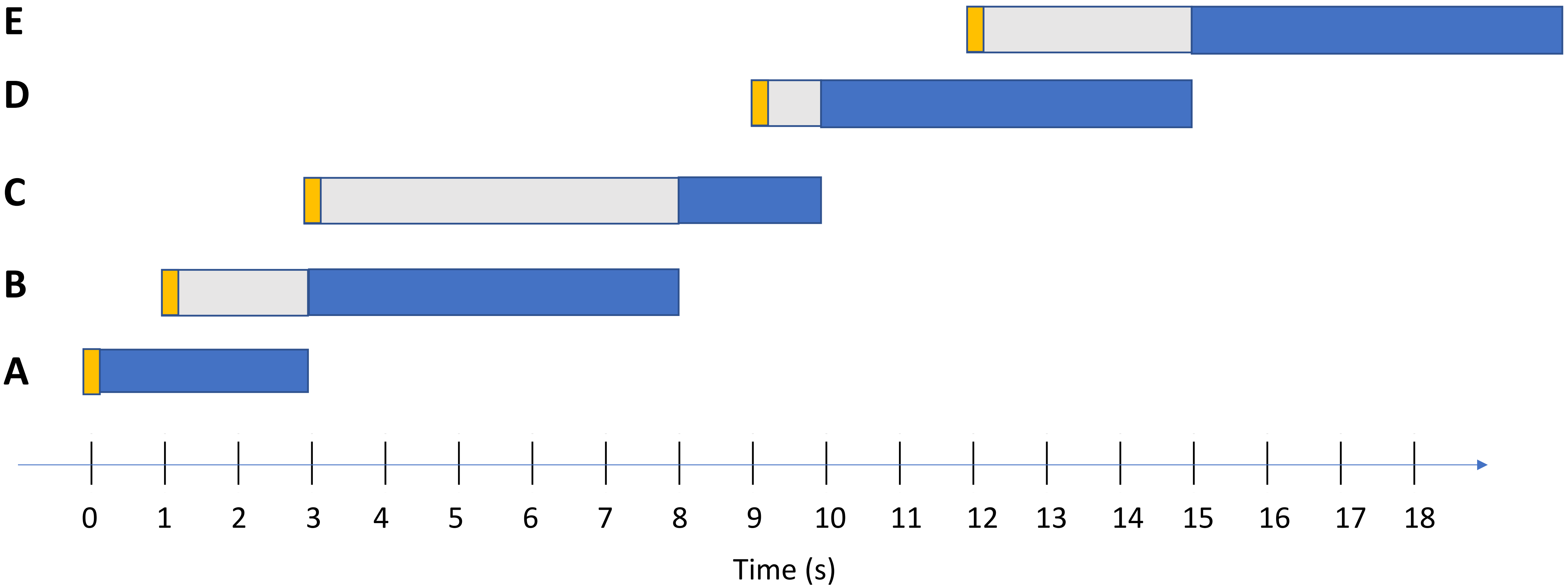
Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



FCFS

Run Wait

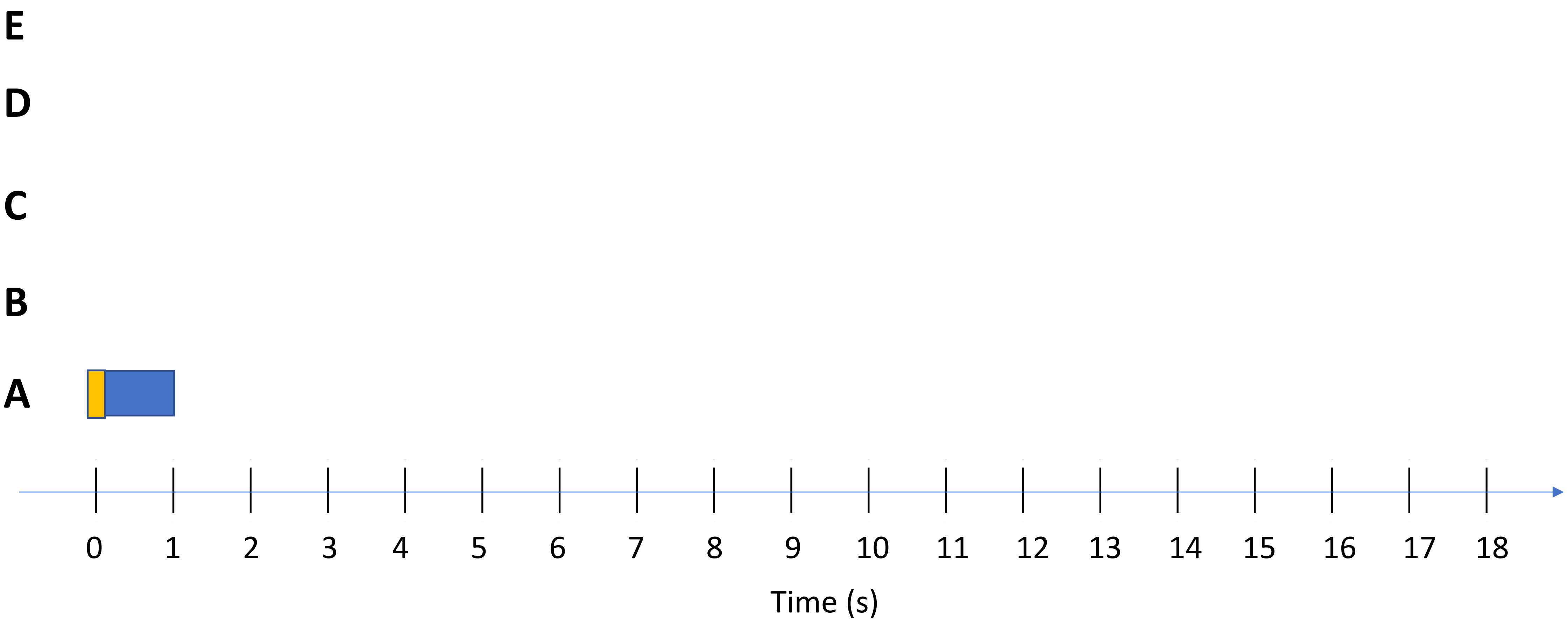
Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3	3	0
B	1	5	7	2
C	3	2	7	5
D	9	5	6	1
E	12	5	8	3



SJF – preemptive

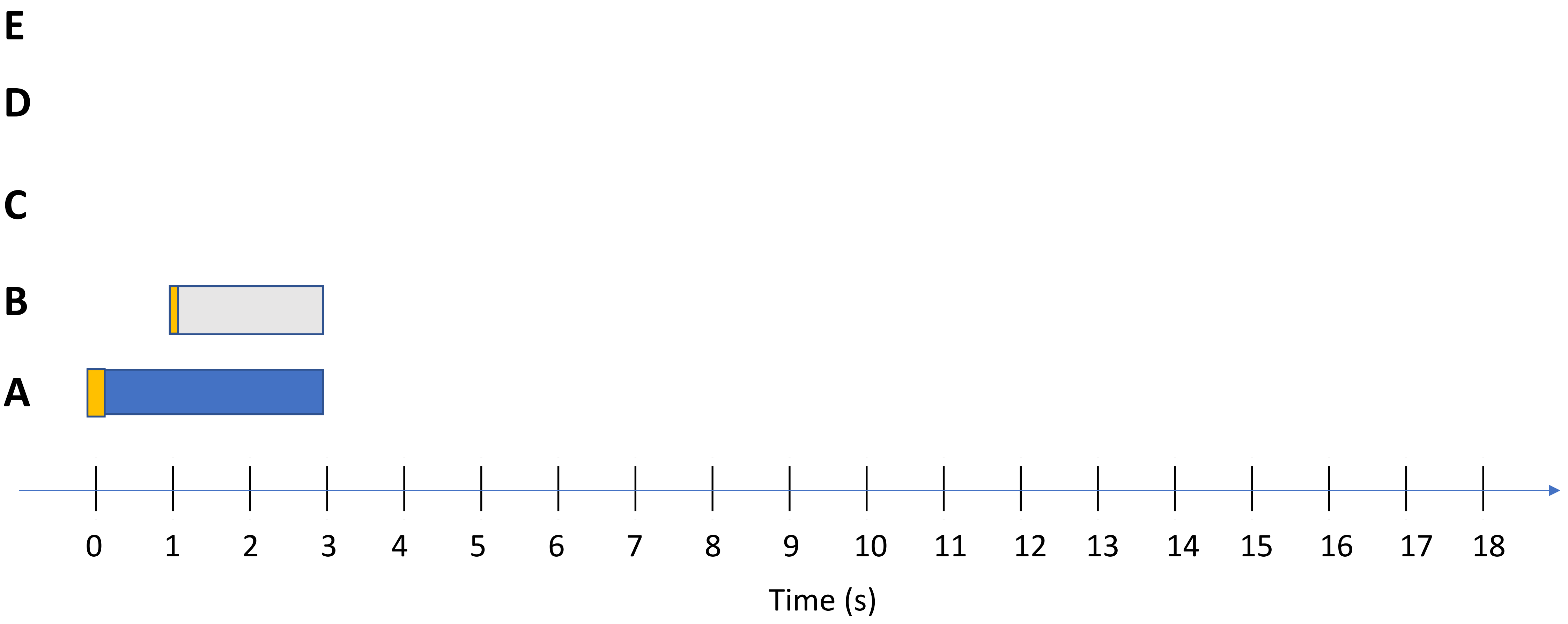
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



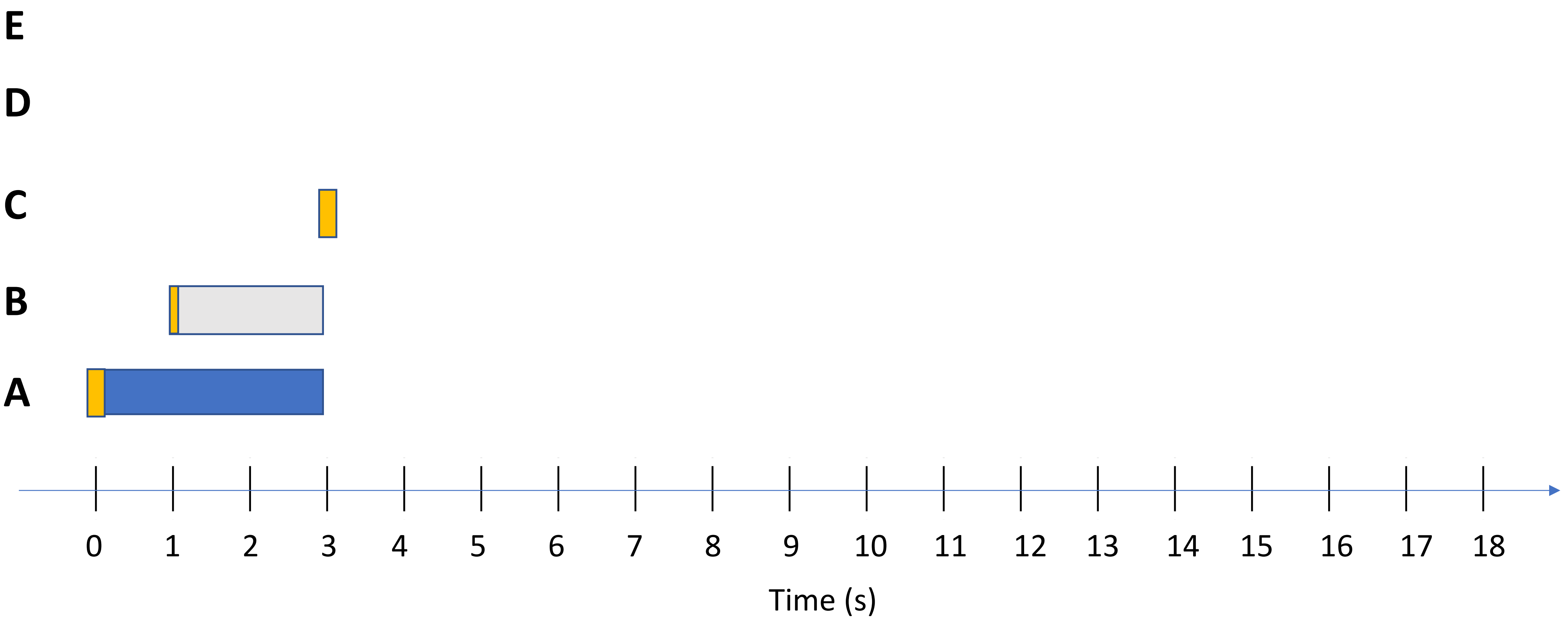
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



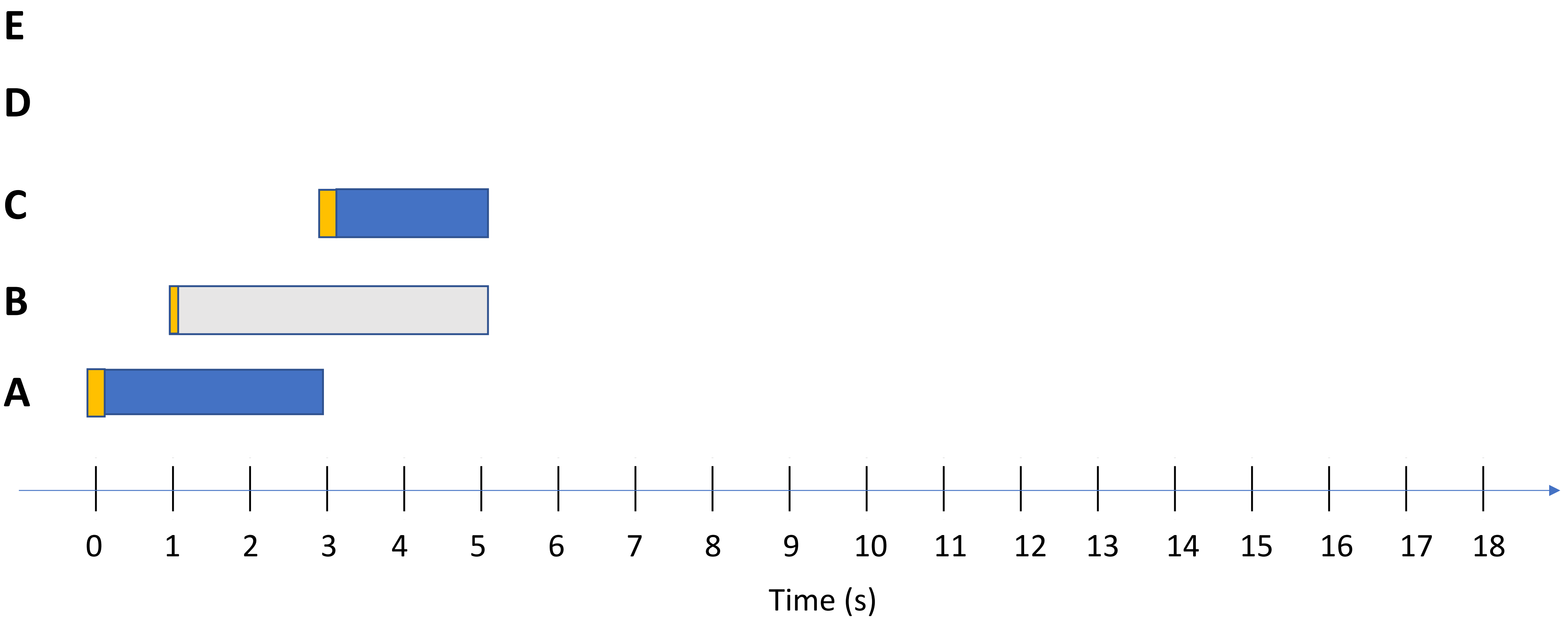
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



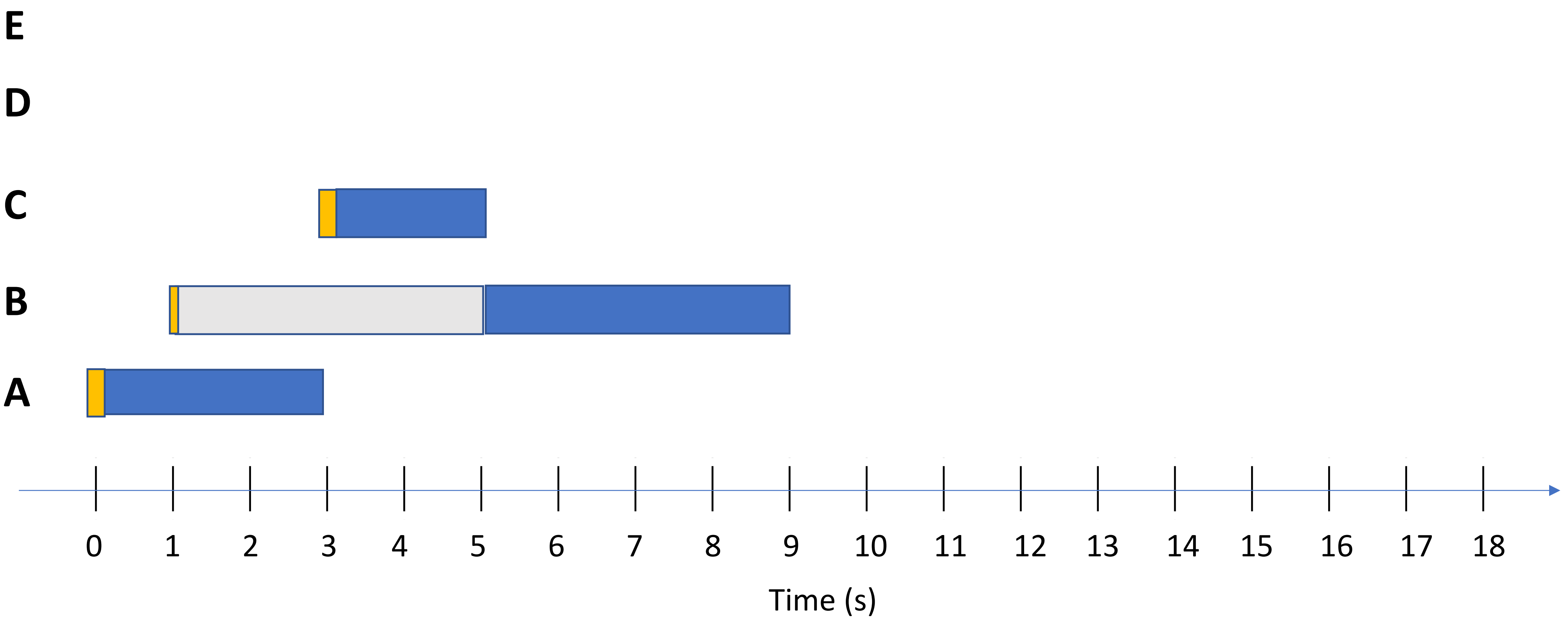
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



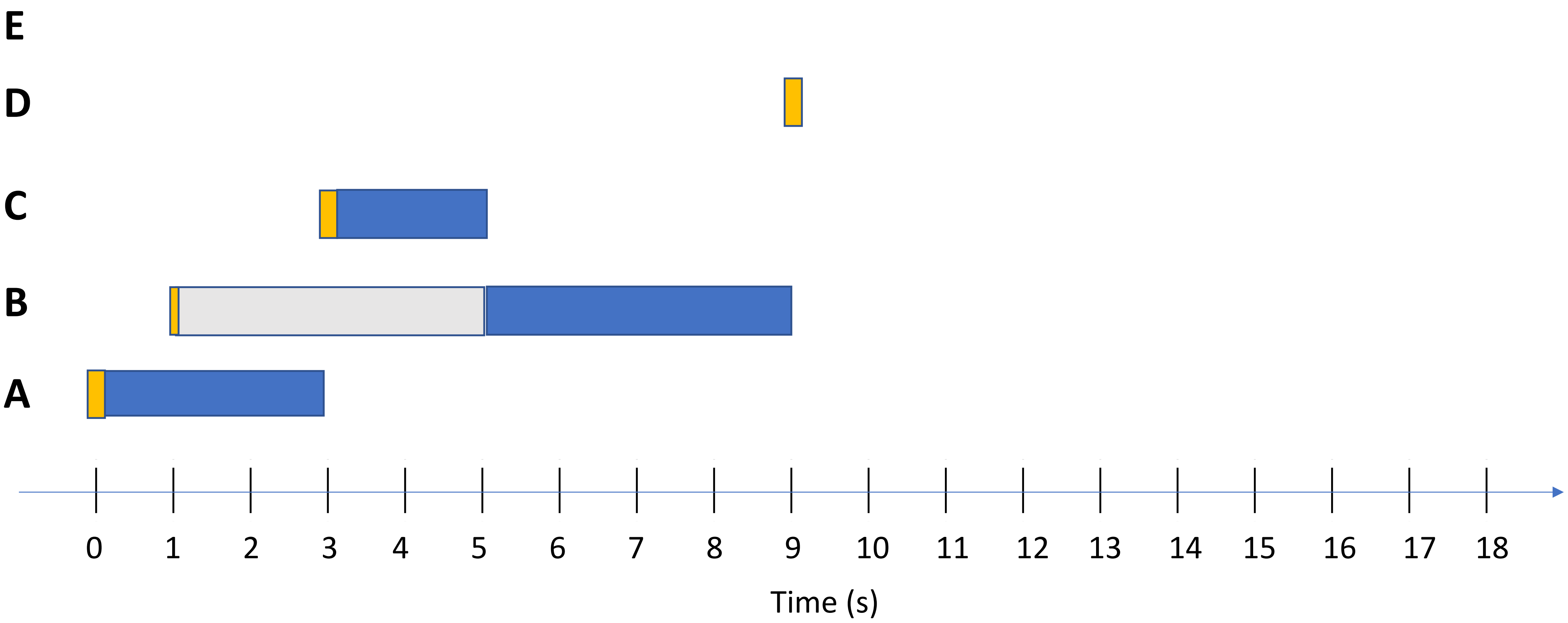
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



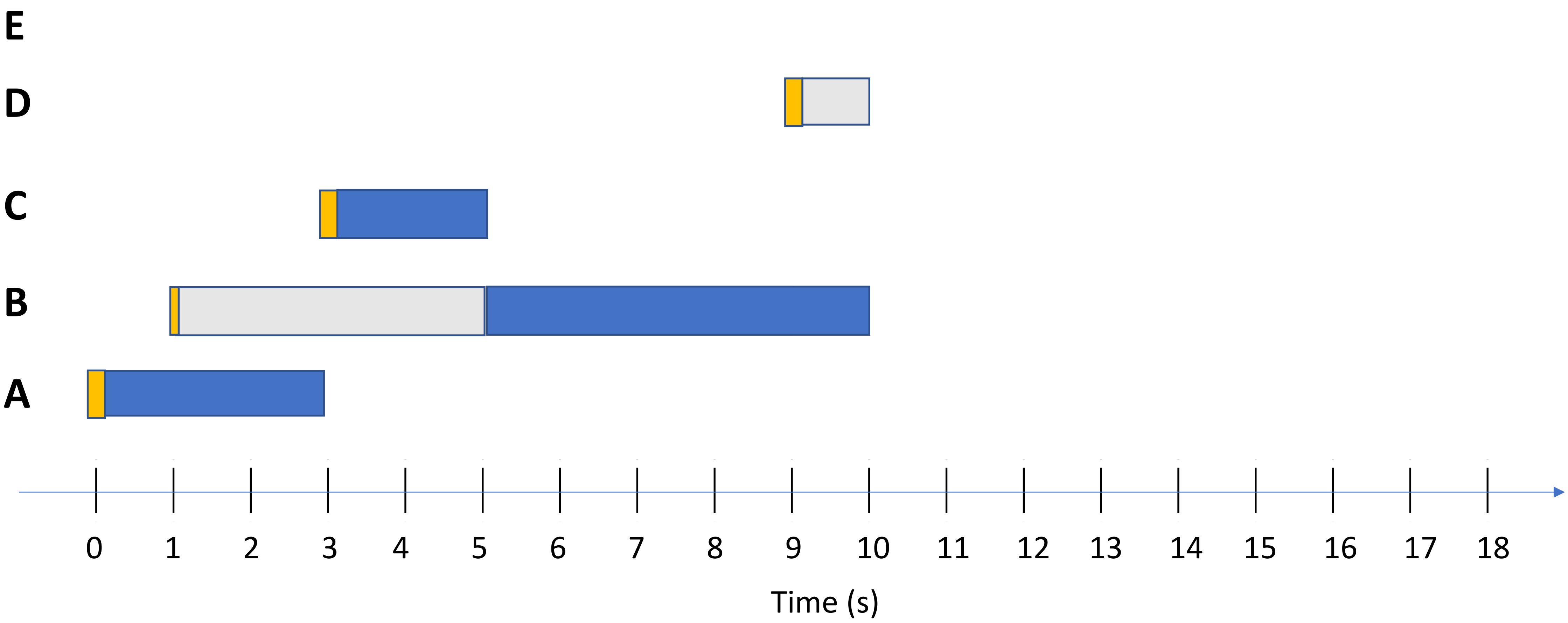
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



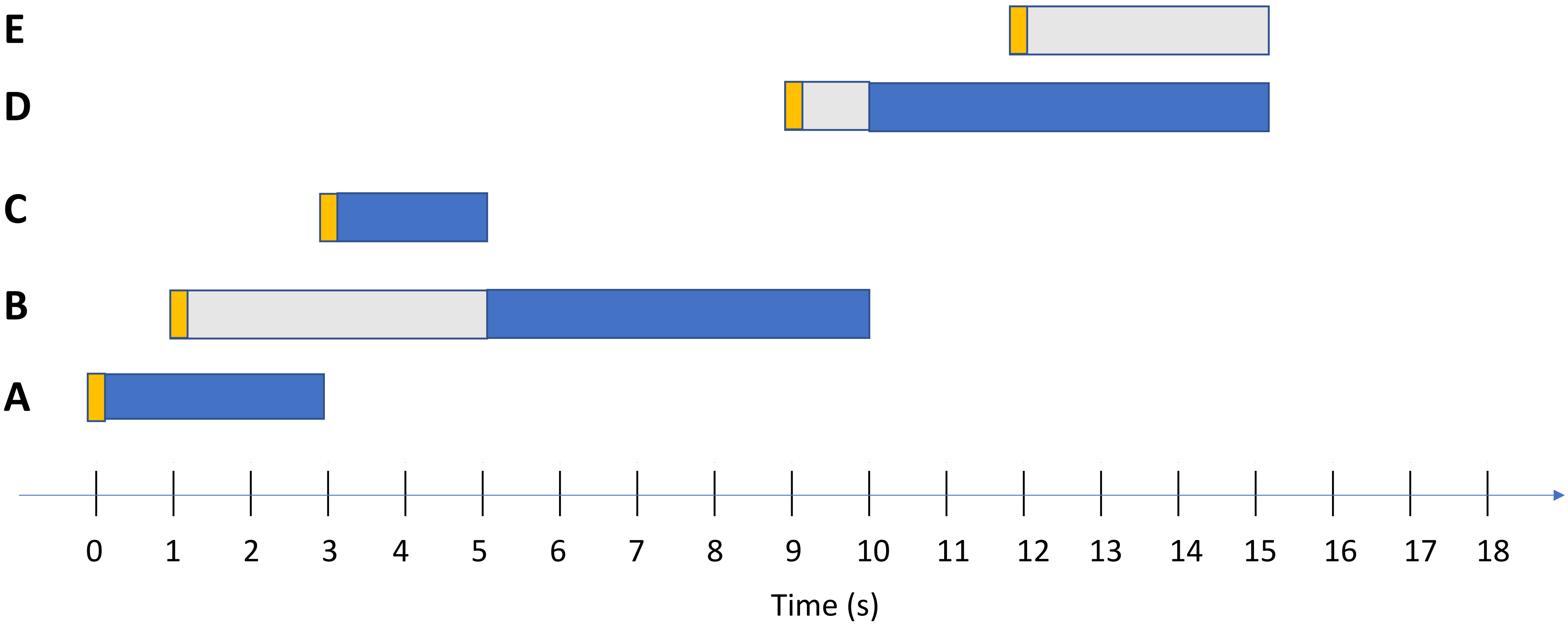
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



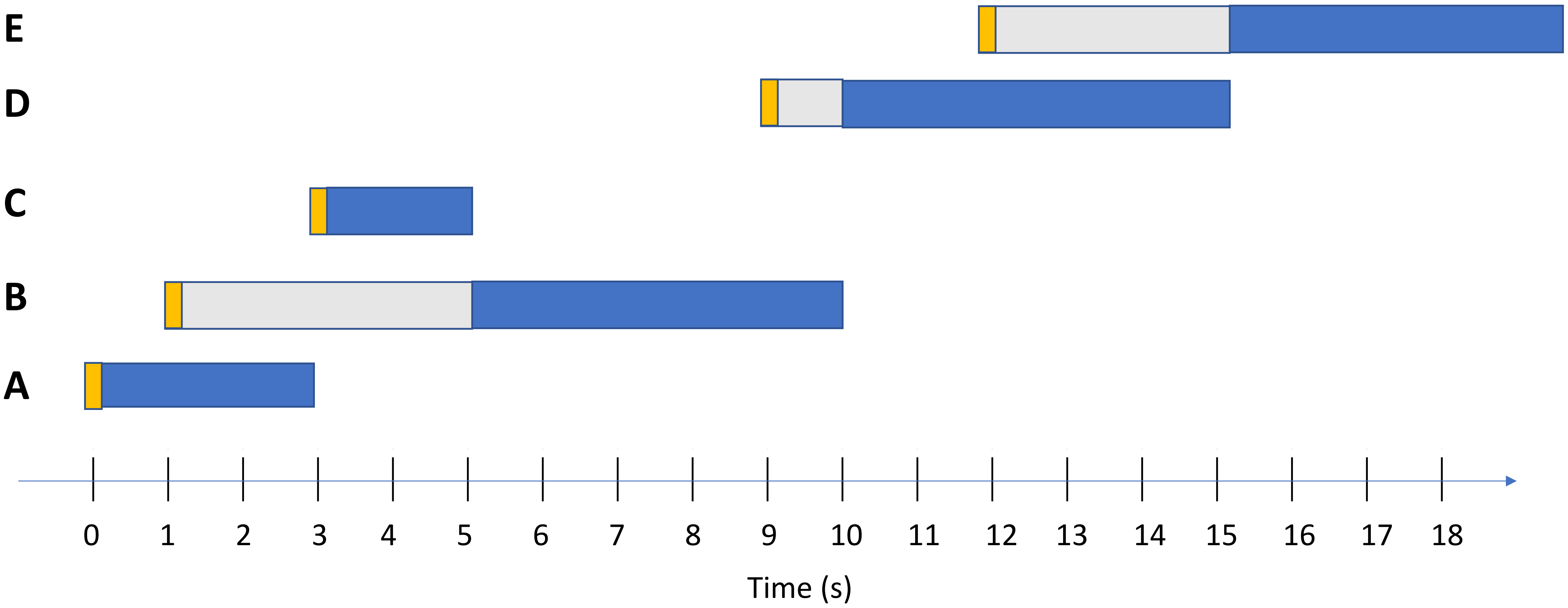
SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



SJF - preemptive

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3	3	0
B	1	5	9	4
C	3	2	2	0
D	9	5	6	1
E	12	5	8	3

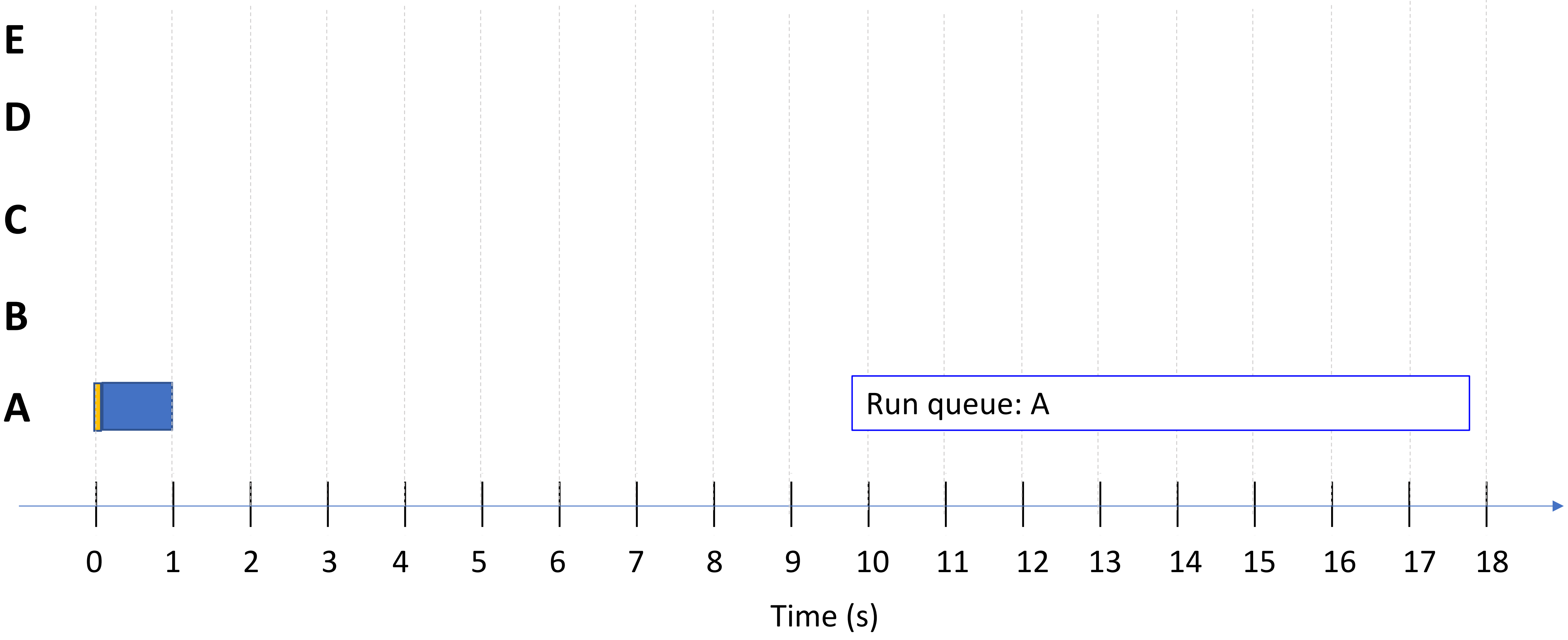


RR

- **time quantum of 1.**

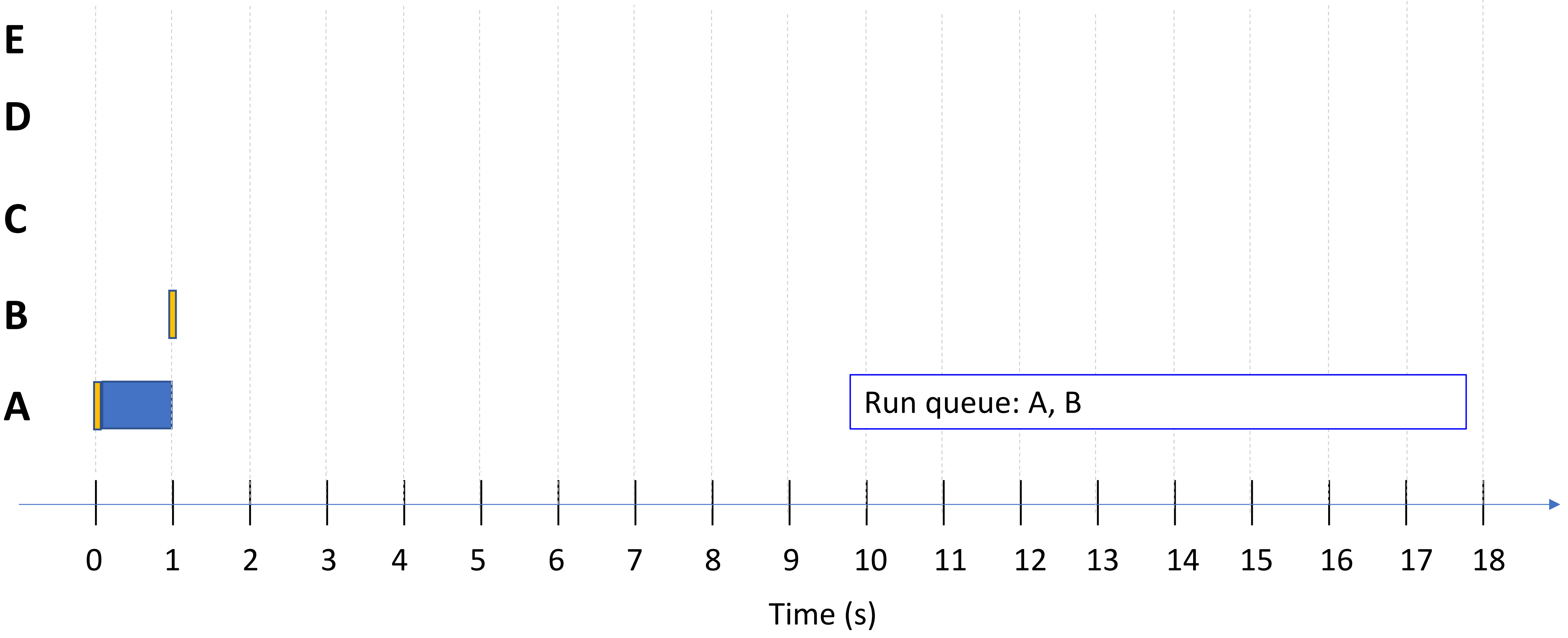
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



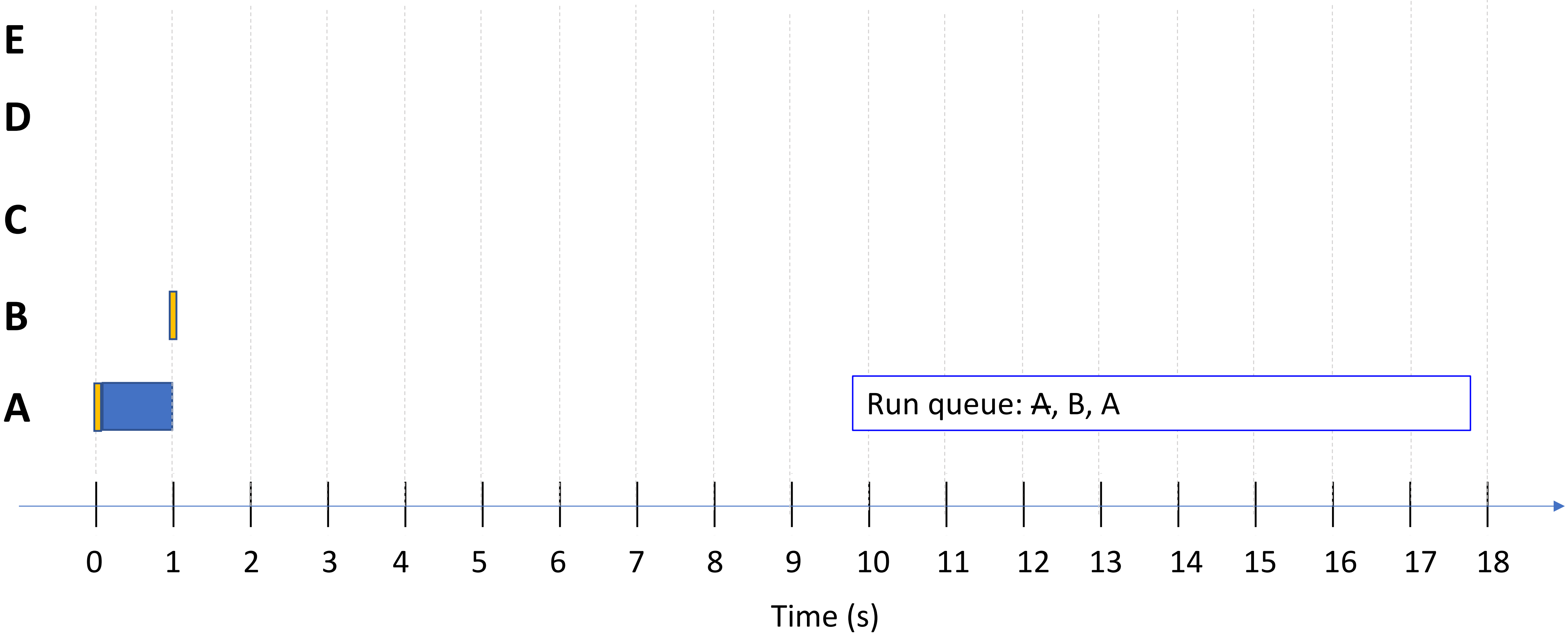
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



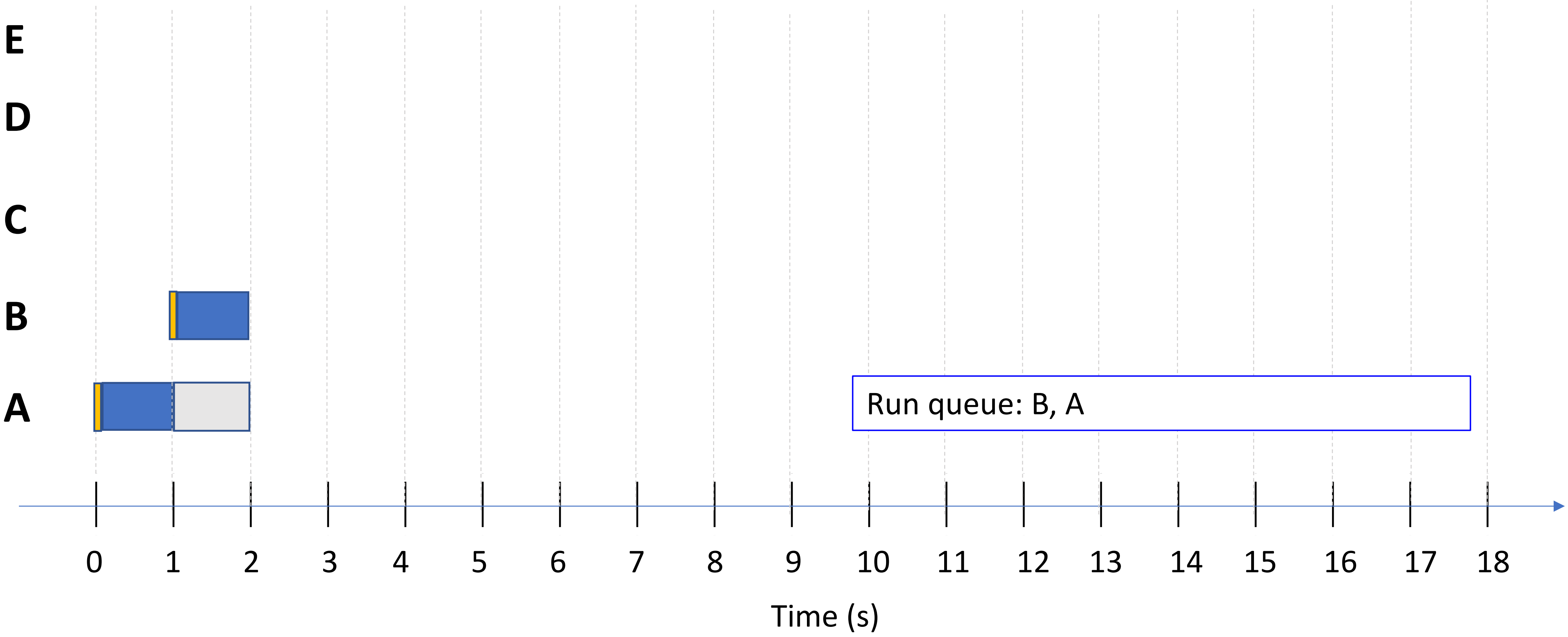
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



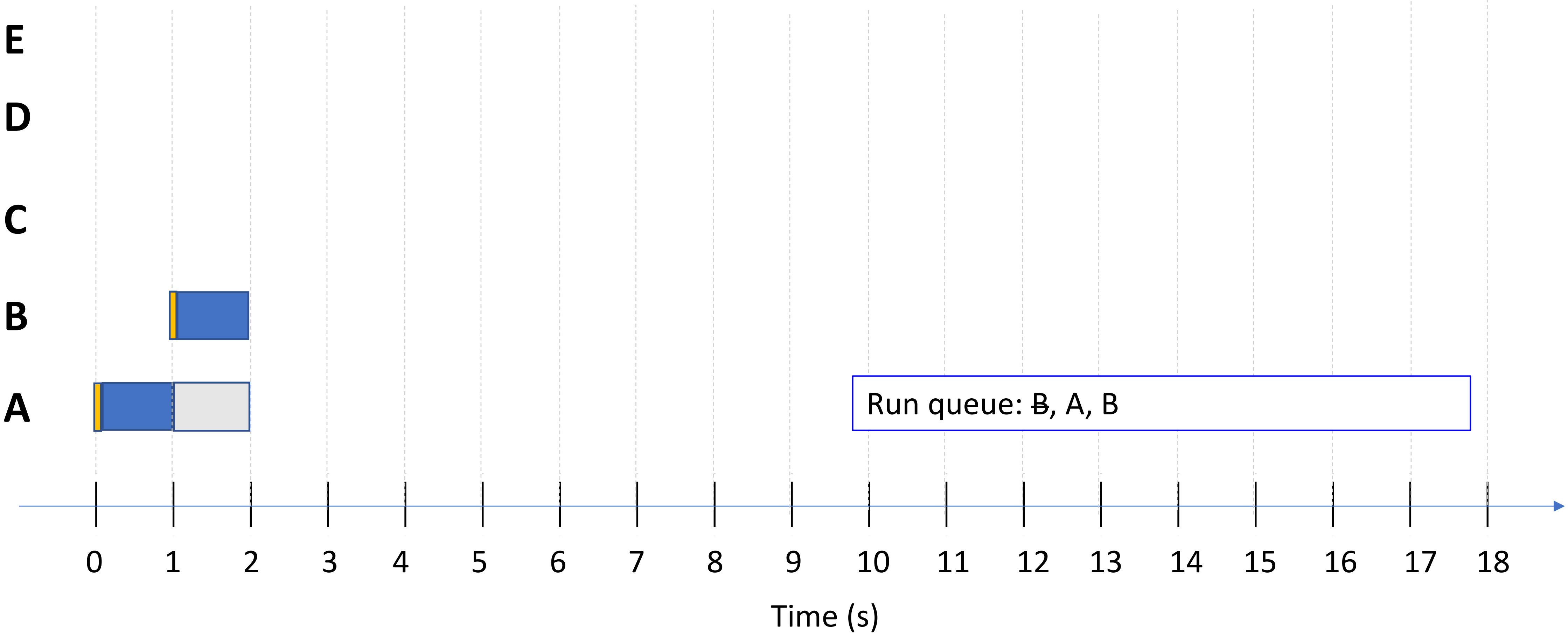
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



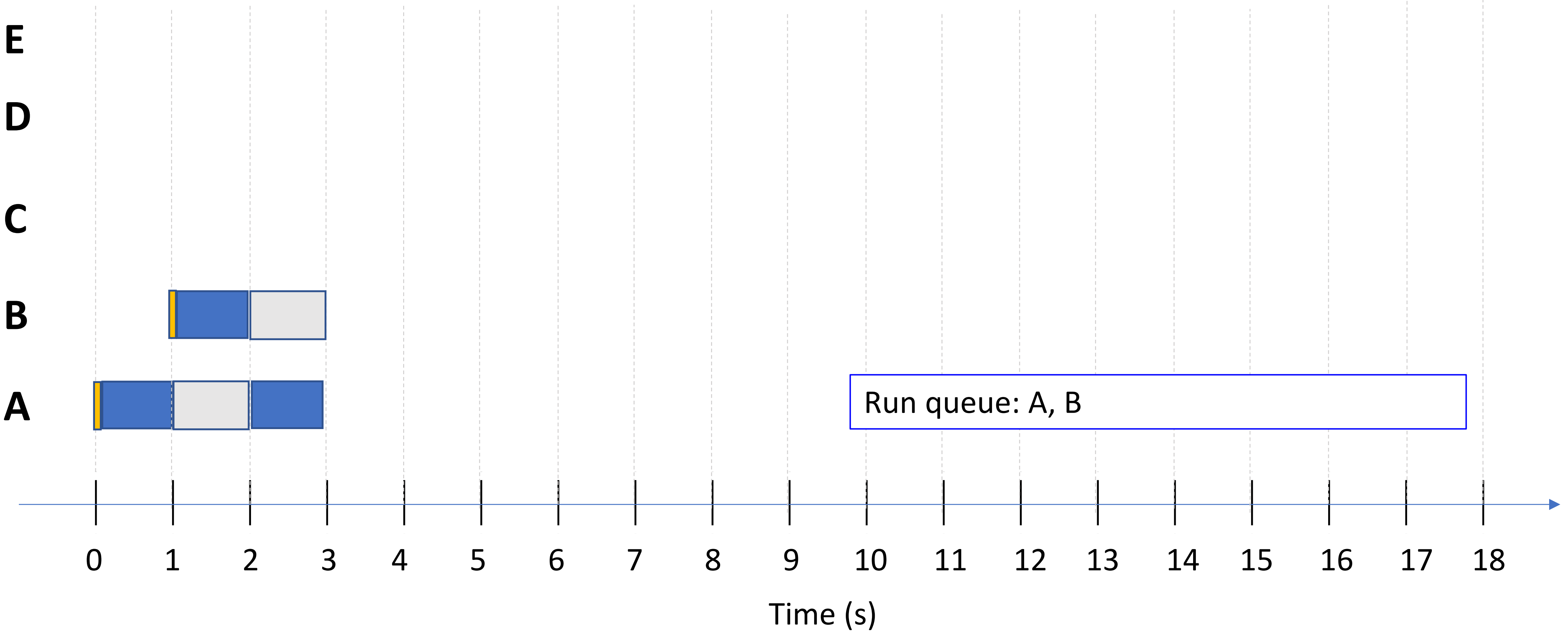
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



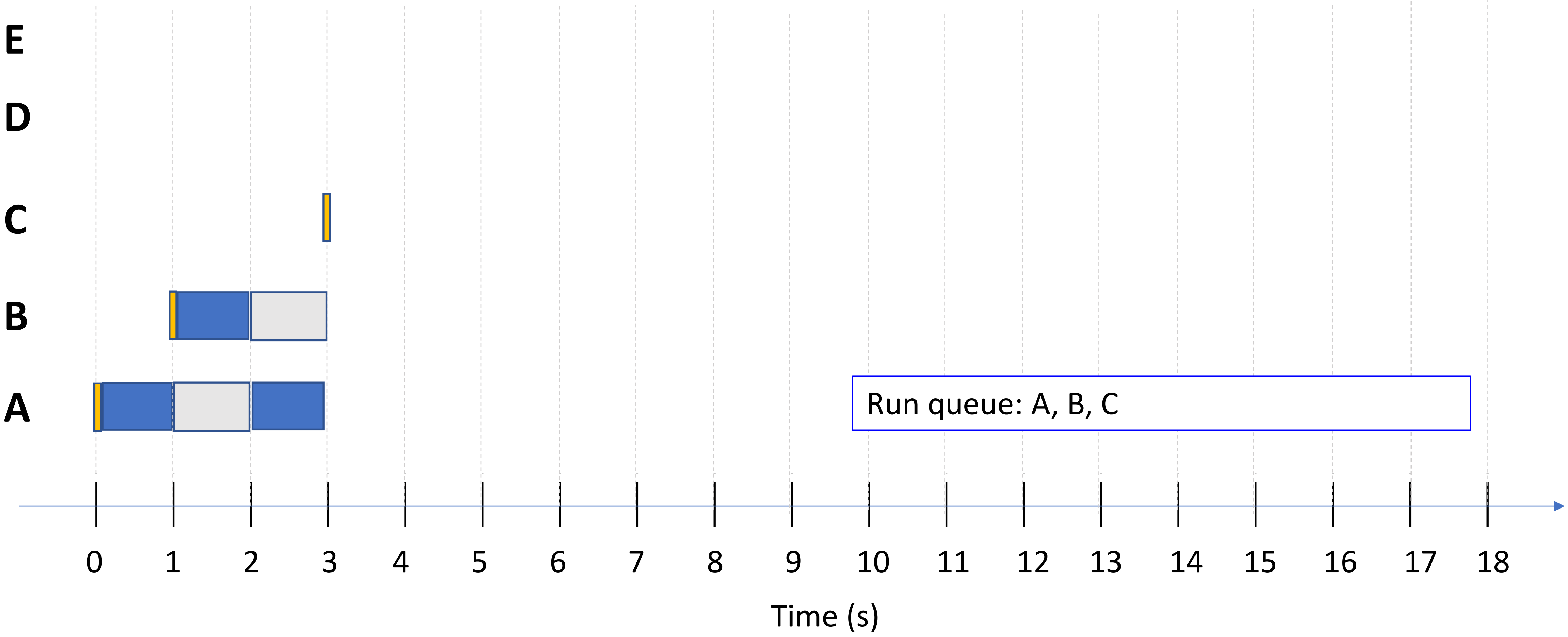
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



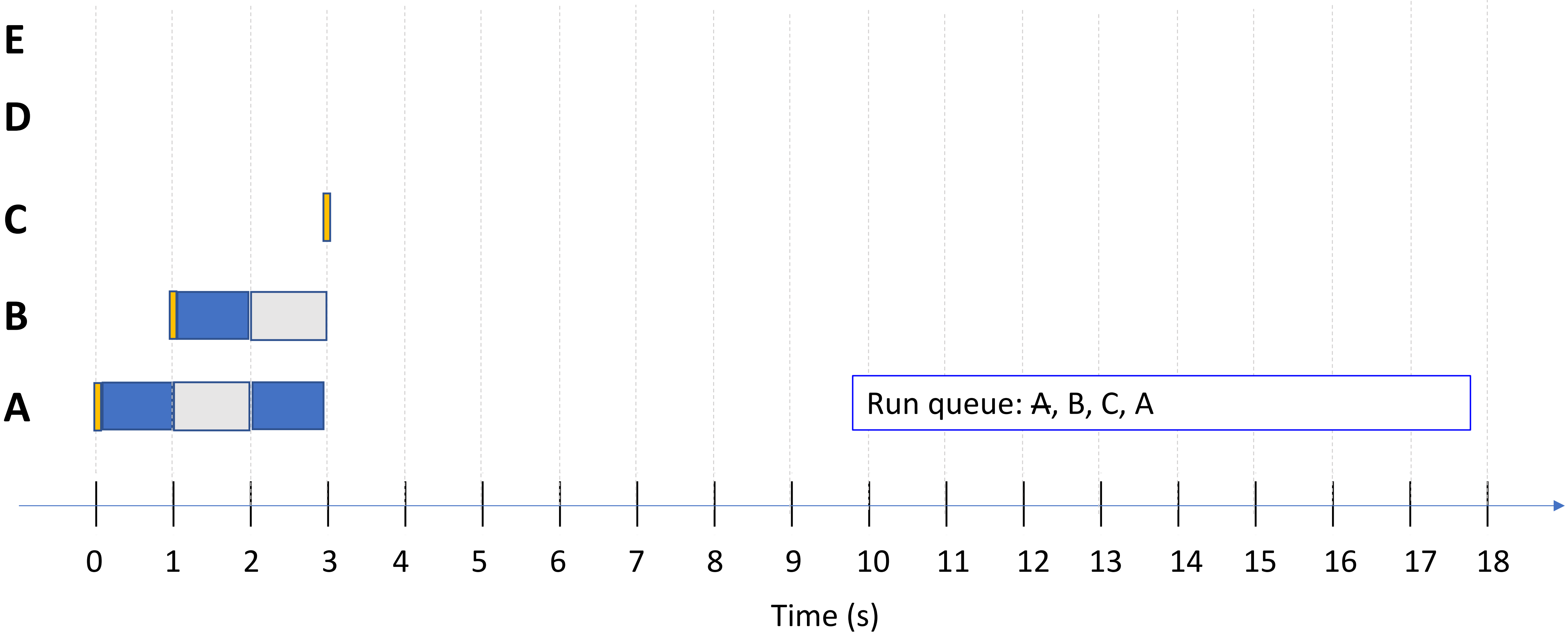
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



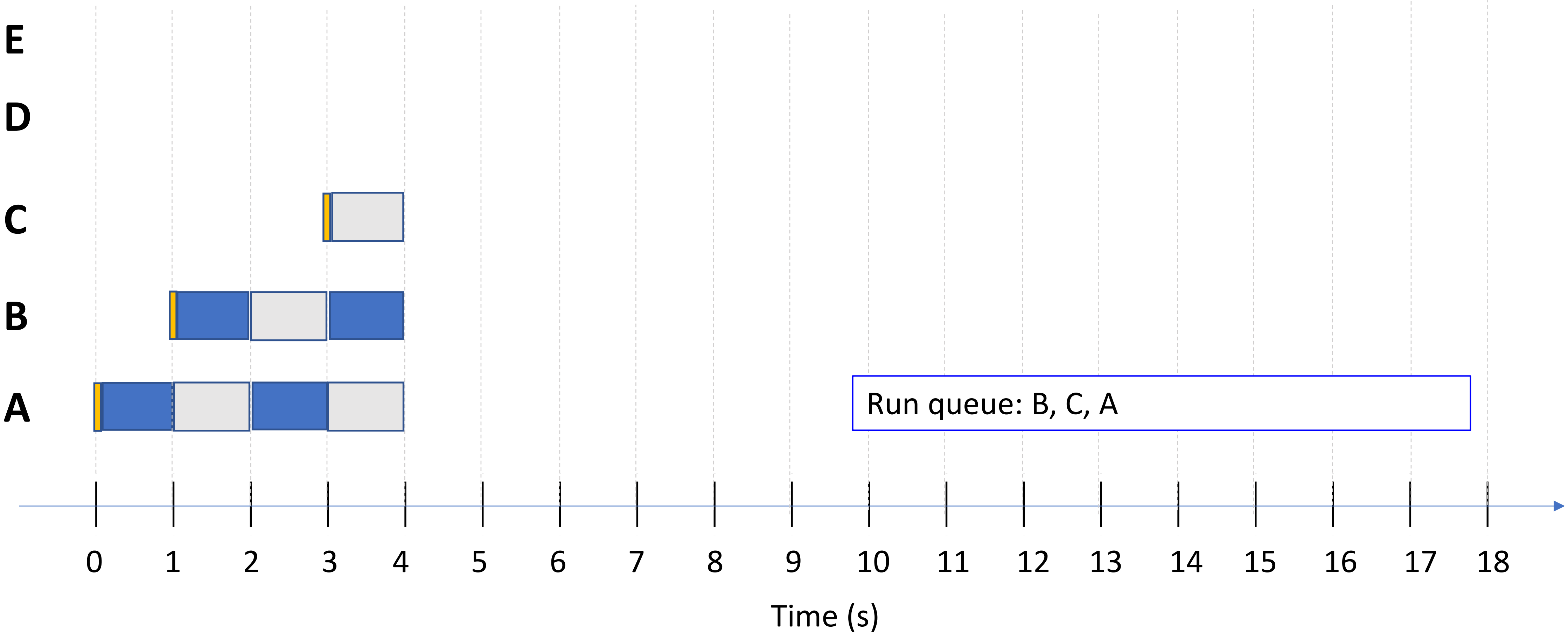
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



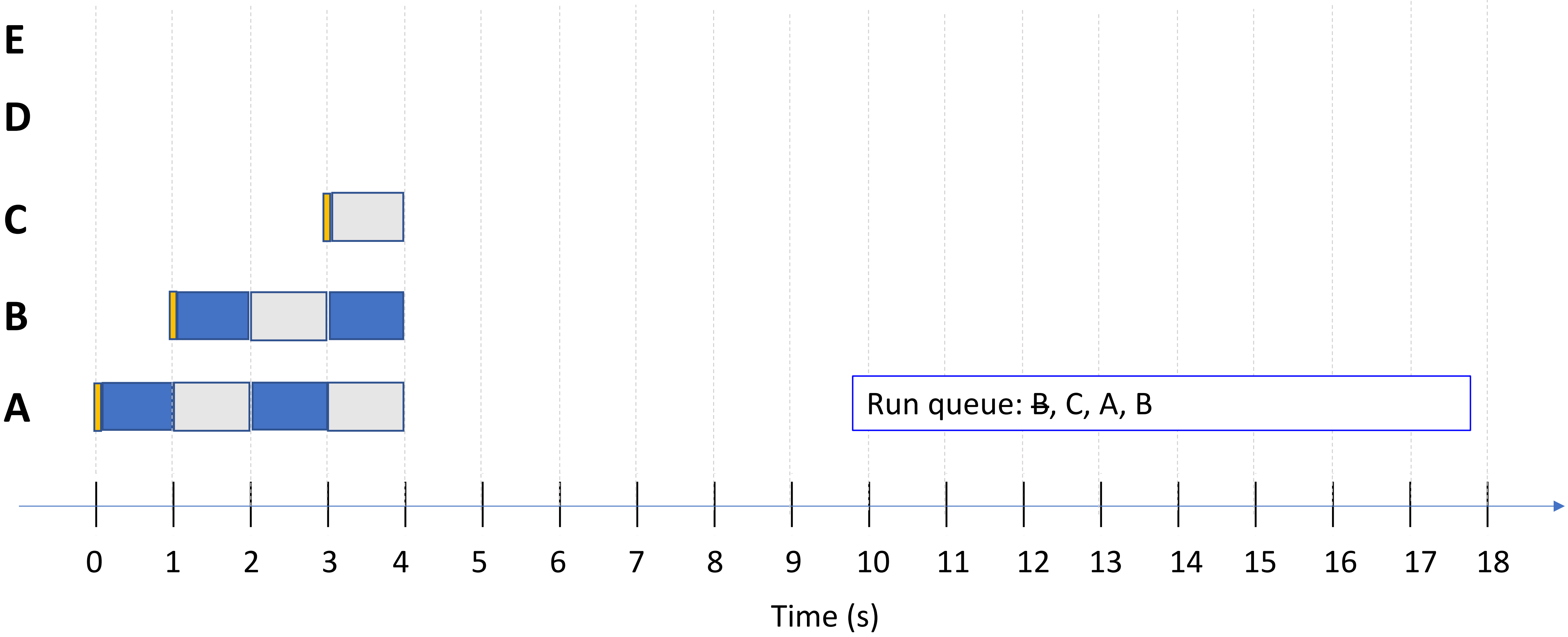
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



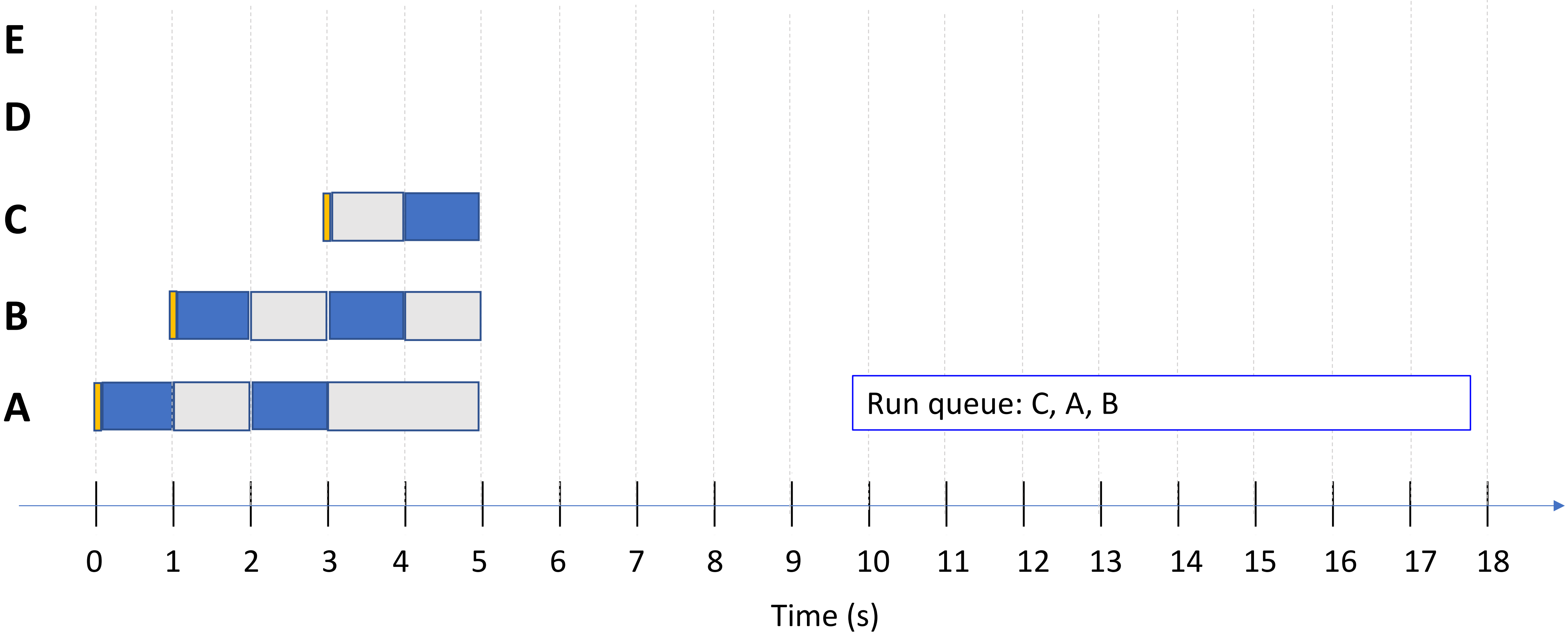
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



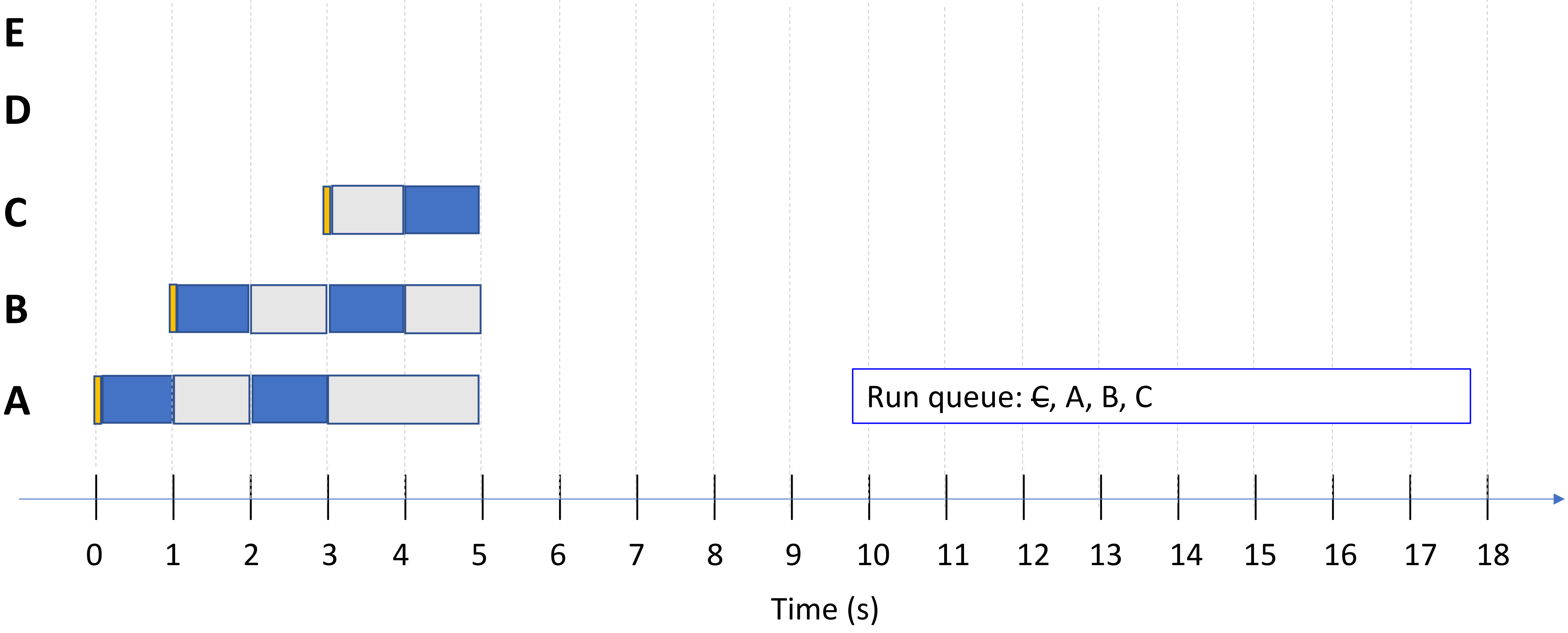
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



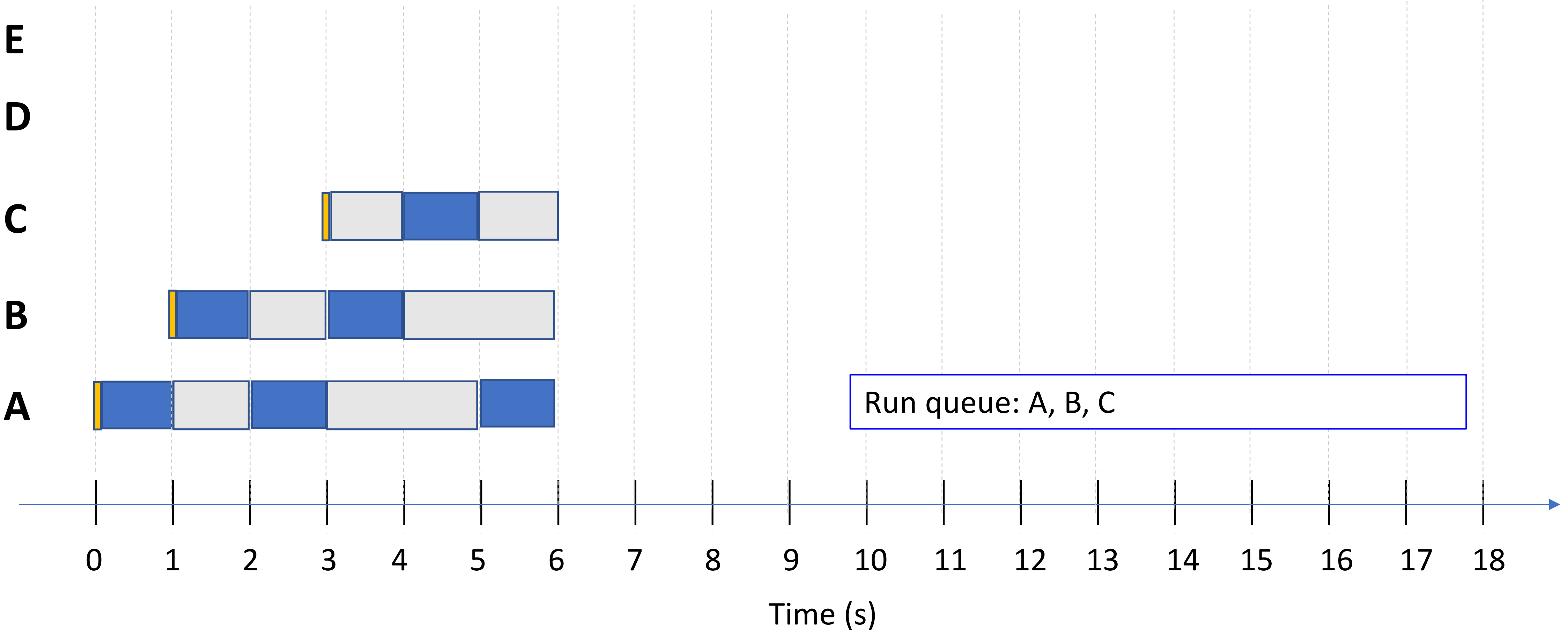
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



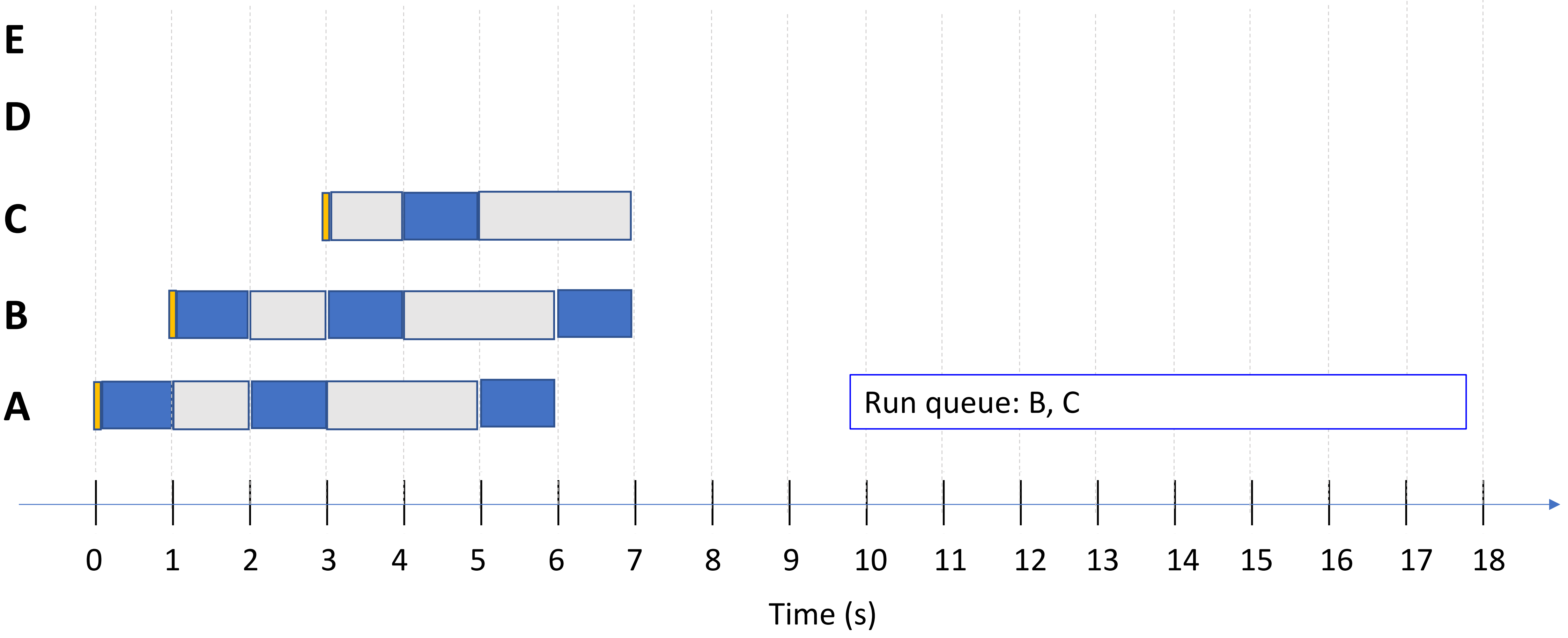
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



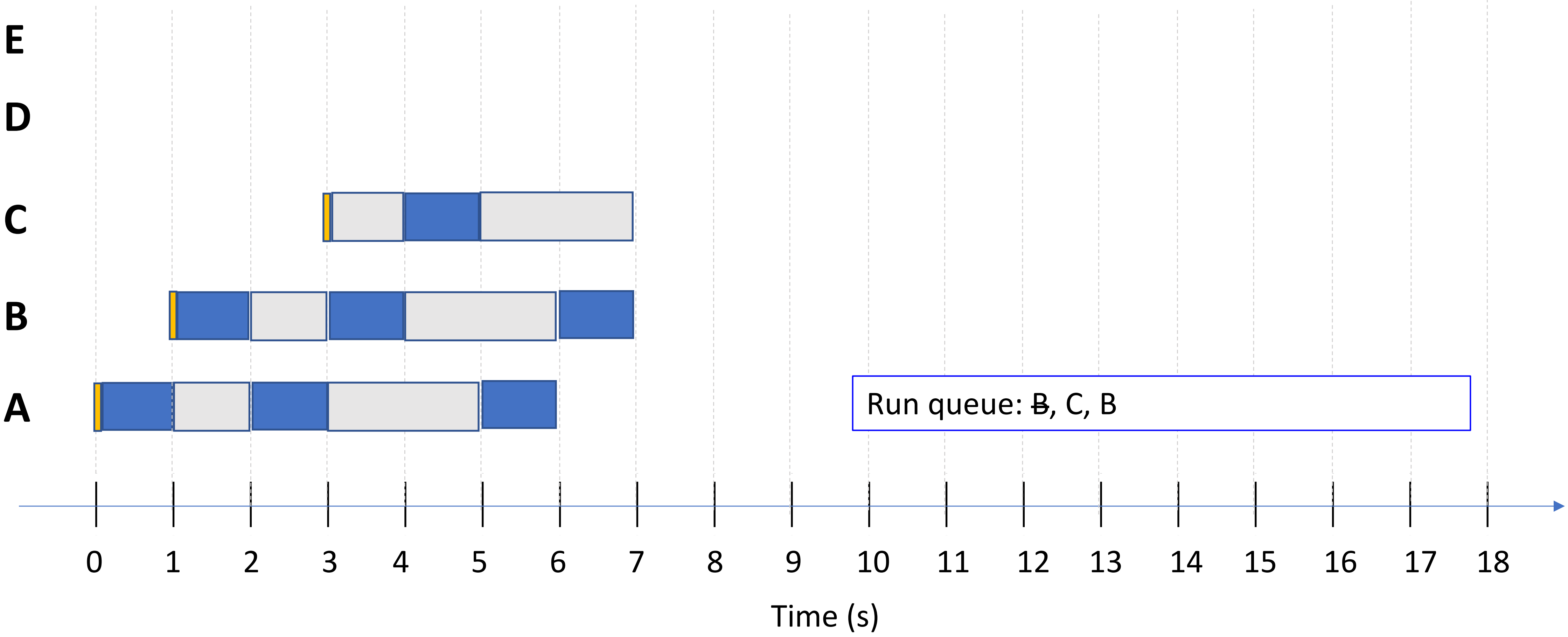
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



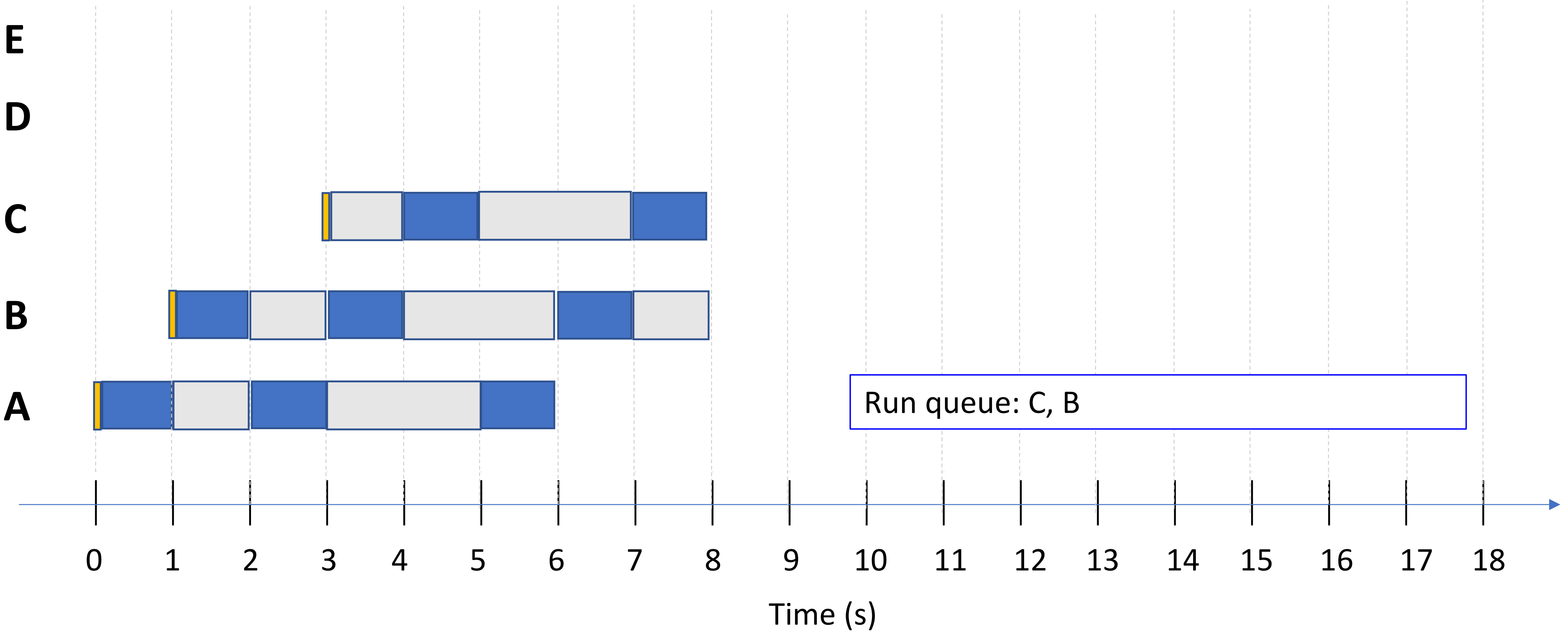
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



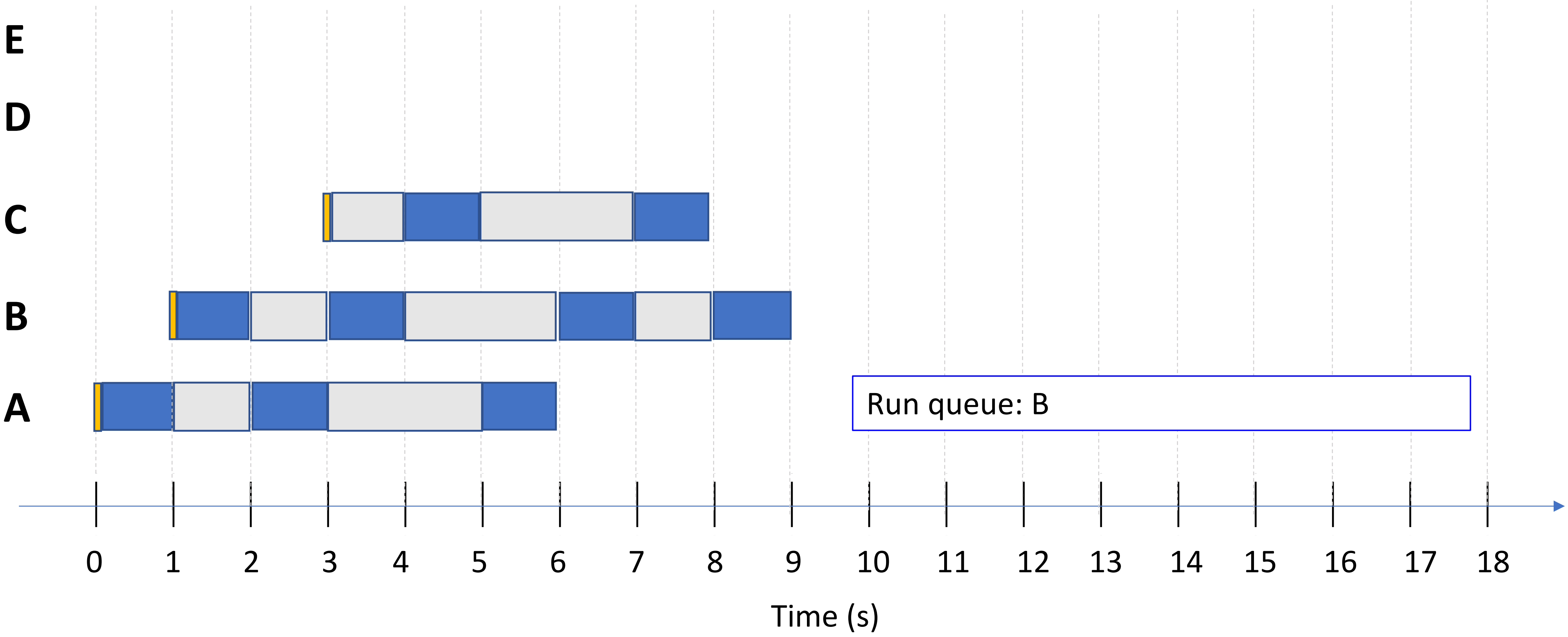
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



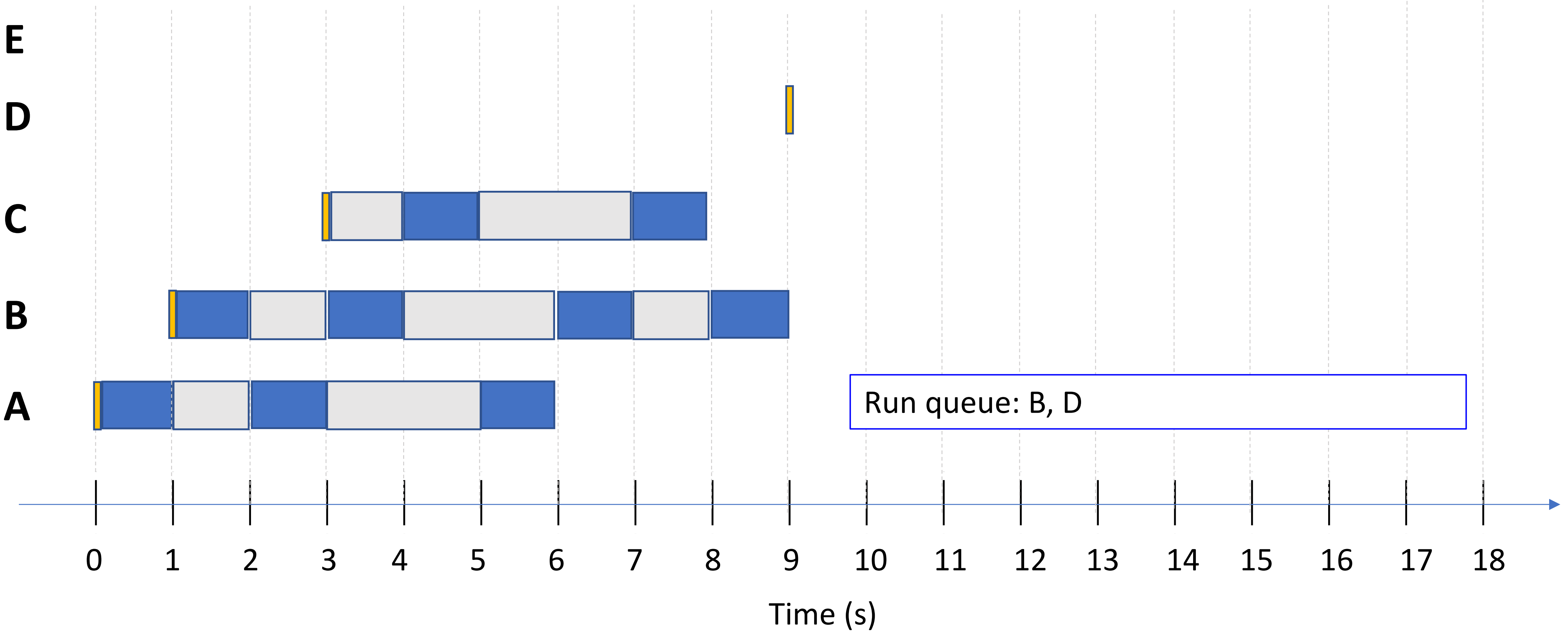
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



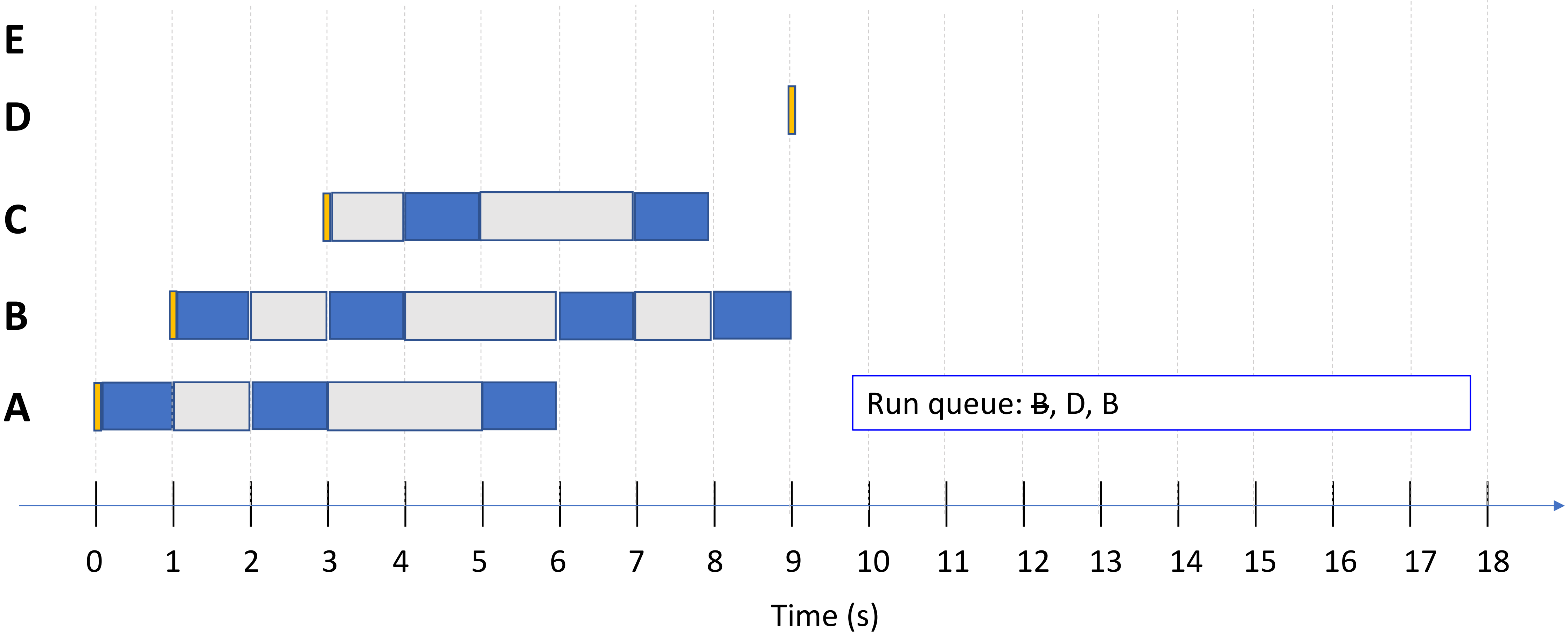
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



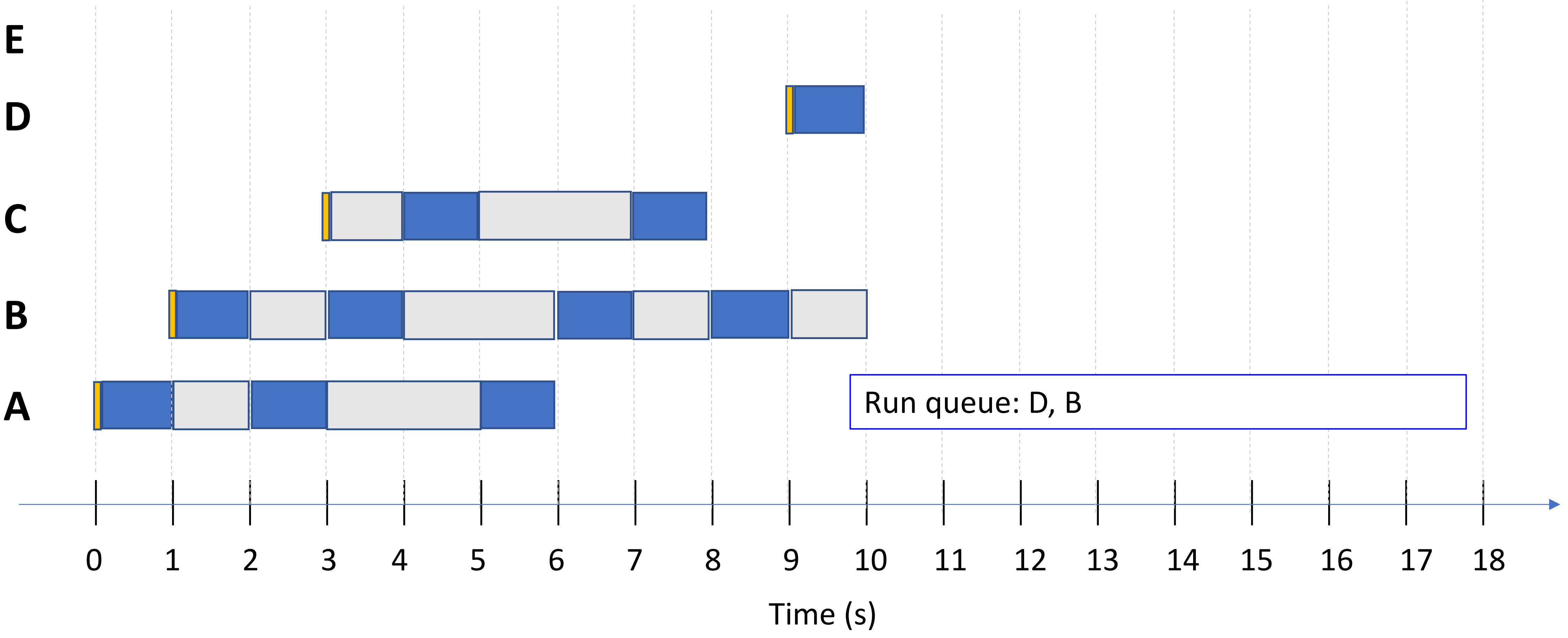
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



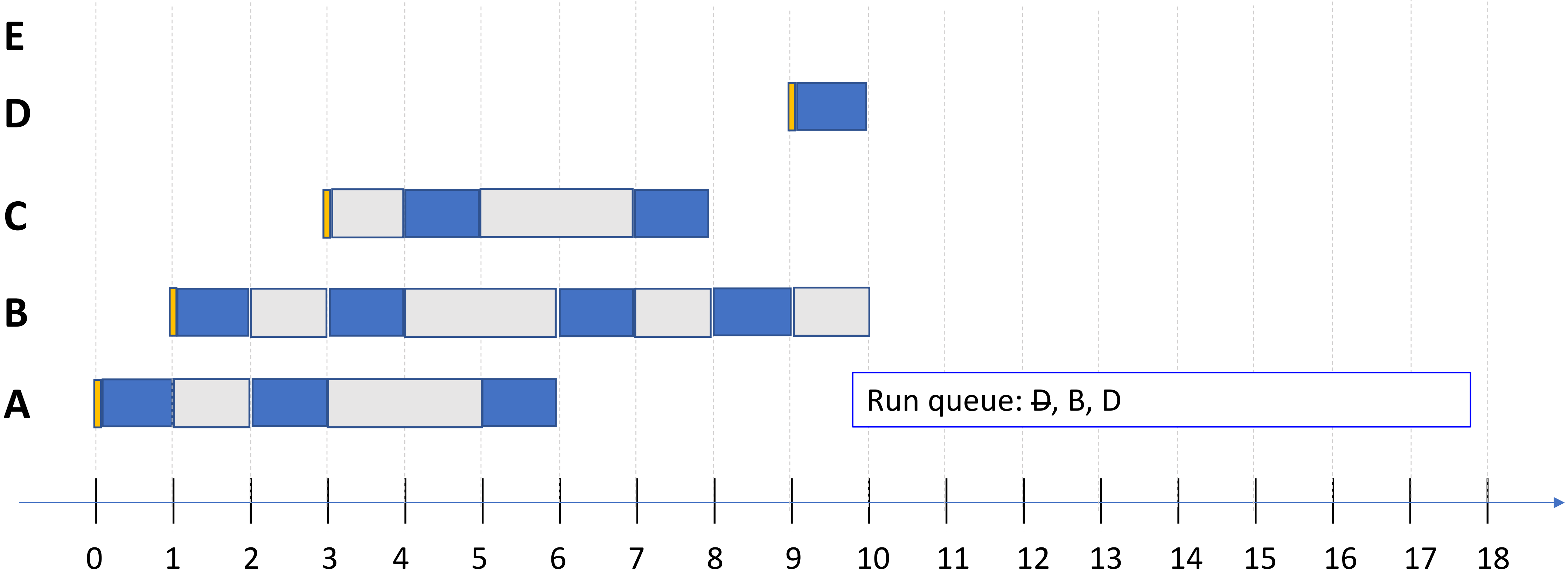
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



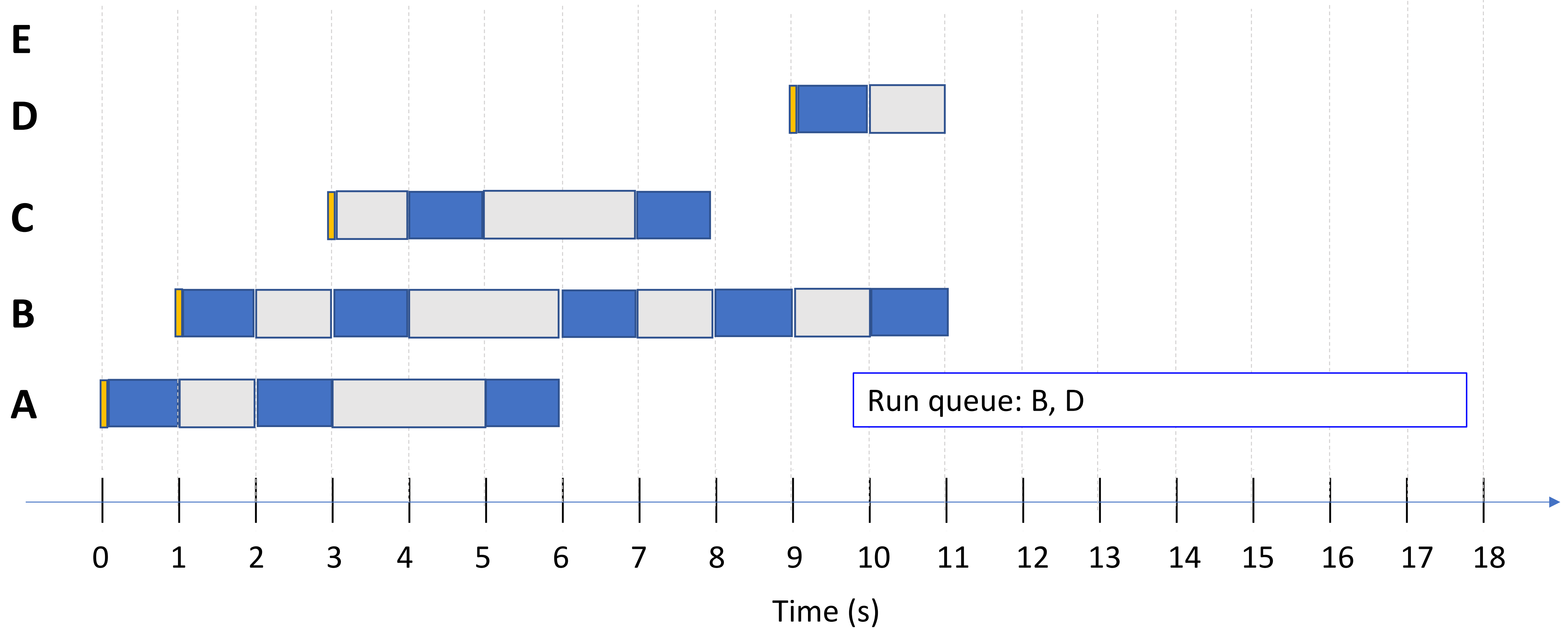
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



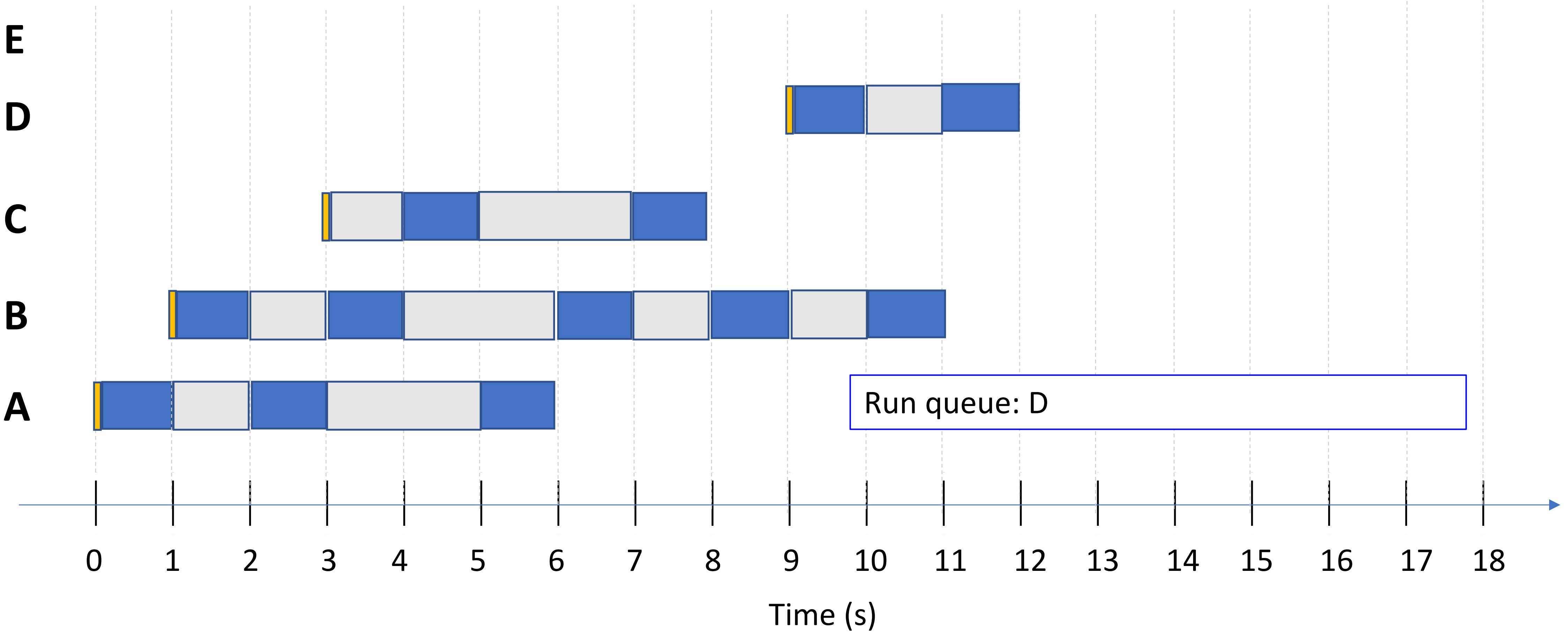
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



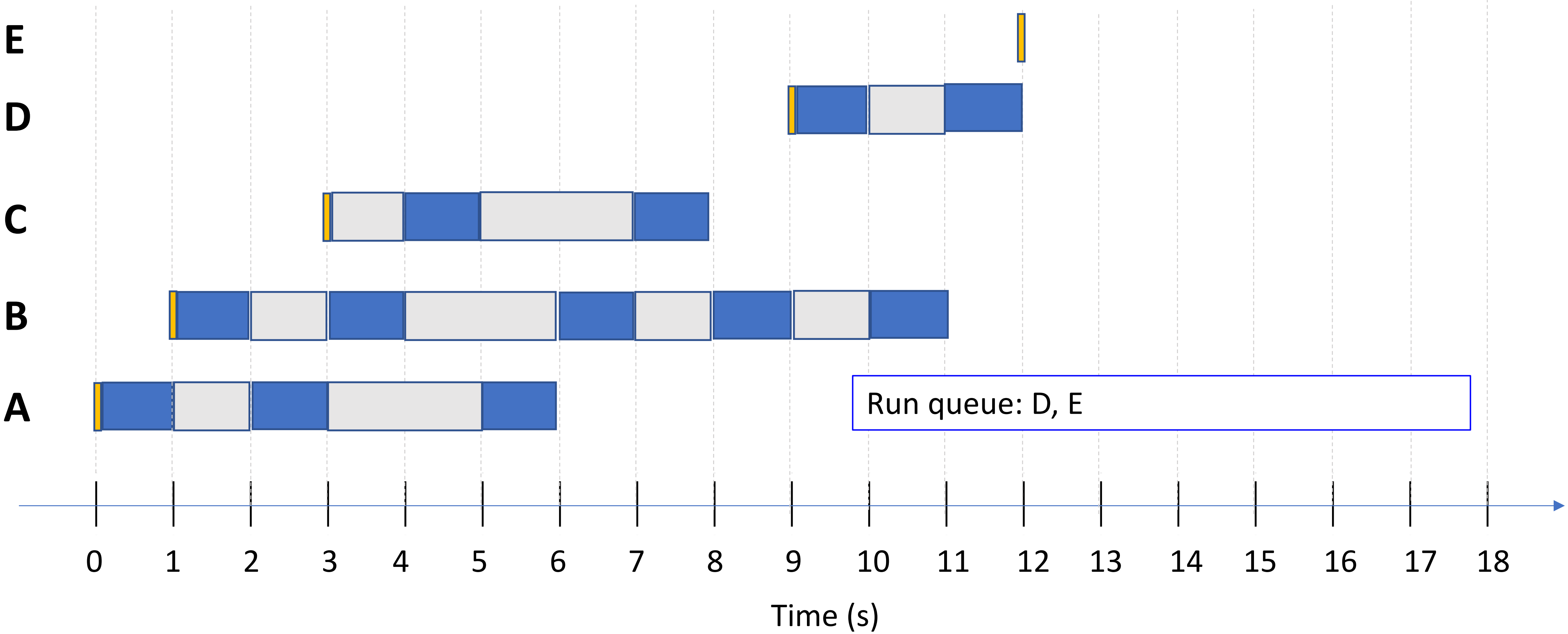
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



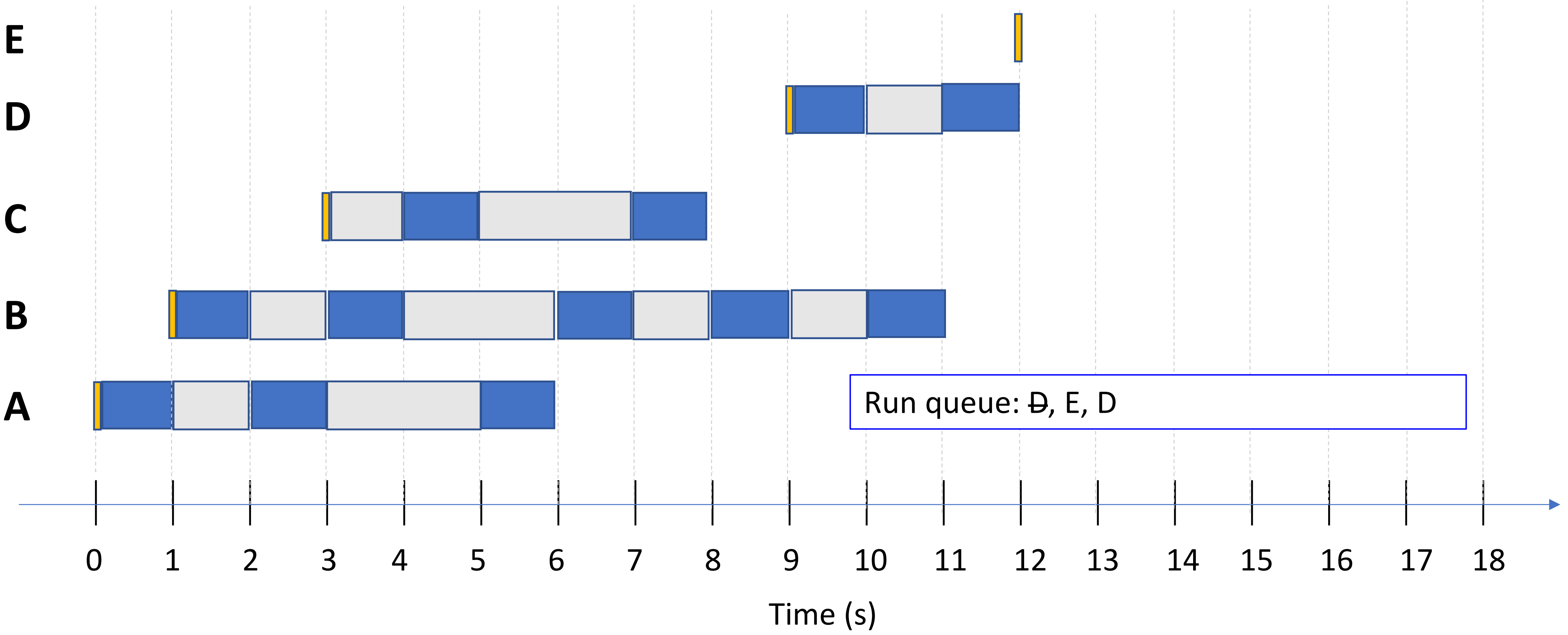
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



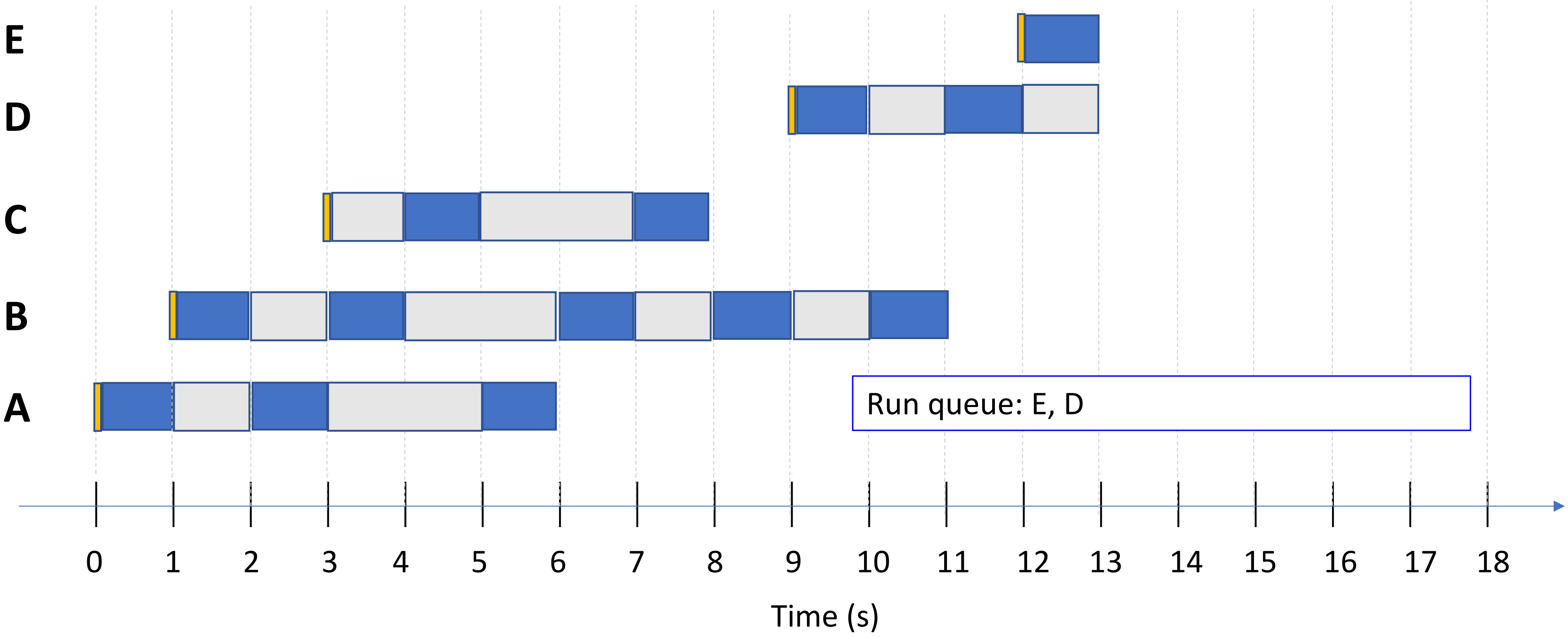
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



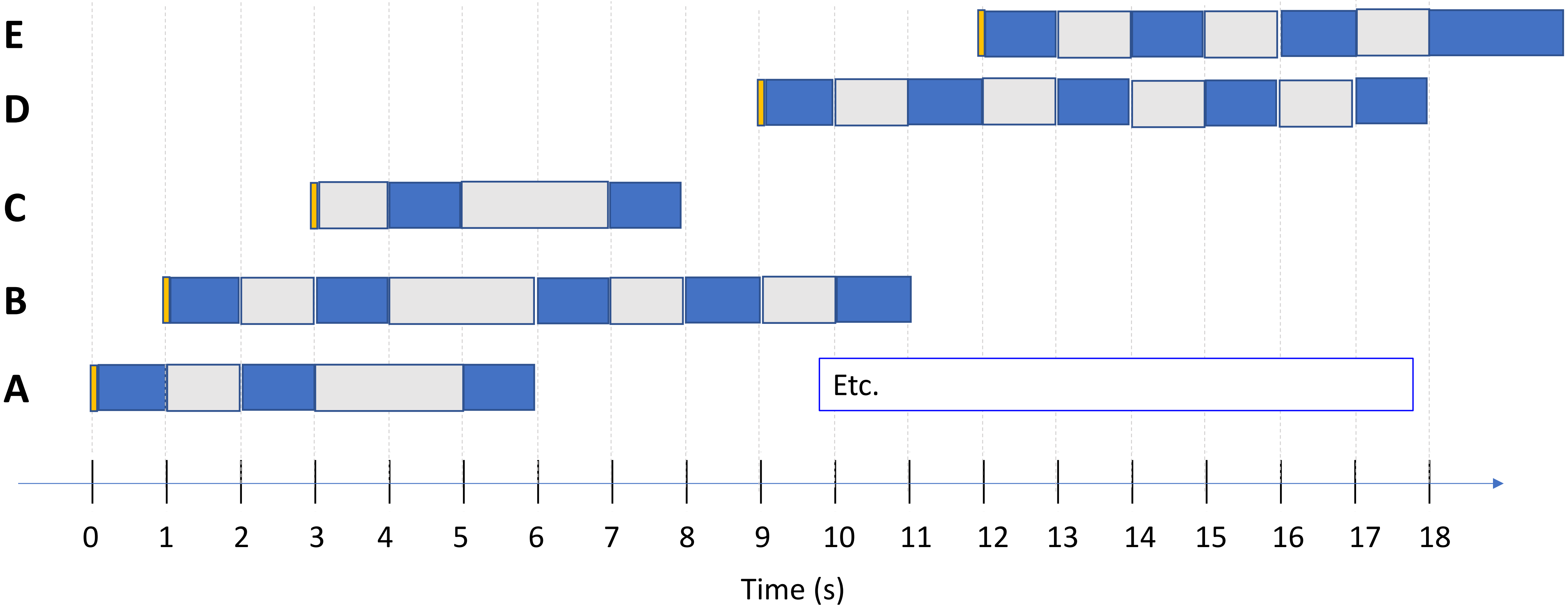
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



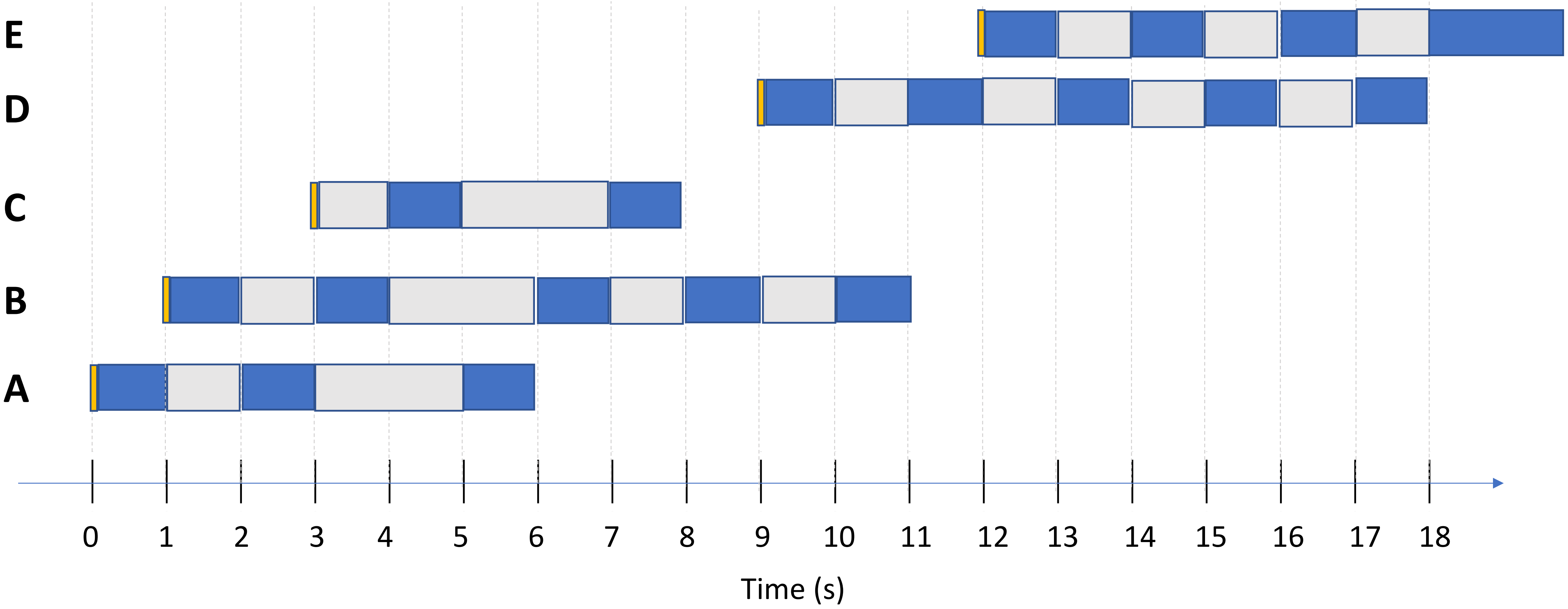
RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3		
B	1	5		
C	3	2		
D	9	5		
E	12	5		



RR

Proc	Arrival time	Execution time	Turnaround time	Response time
A	0	3	6	0
B	1	5	10	0
C	3	2	5	1
D	9	5	9	0
E	12	5	8	0



Summary – Key Concepts

- Process
  - Program in execution
- Linux process tree
  - Created by fork() / exec() / wait() / exit()
- Process switch
  - Change of process using the CPU
  - Save and restore registers and other info
- Process scheduler
  - Decides which process to run next

Further Optional Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 3 – 7 (inclusive) https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University), Prof. Natacha Crooks (UC Berkeley).