Announcements

- Room change ELA1
- Parallelism and Concurrent final exam overlap
- Students without pre-requisite courses

Recap of Week 1

Pamela Delgado

February 20, 2019

(slides Willy Zwaenepoel)

What does the OS do?

- Abstraction: makes hardware easier to use
- Resource management: allocate hardware resources between users

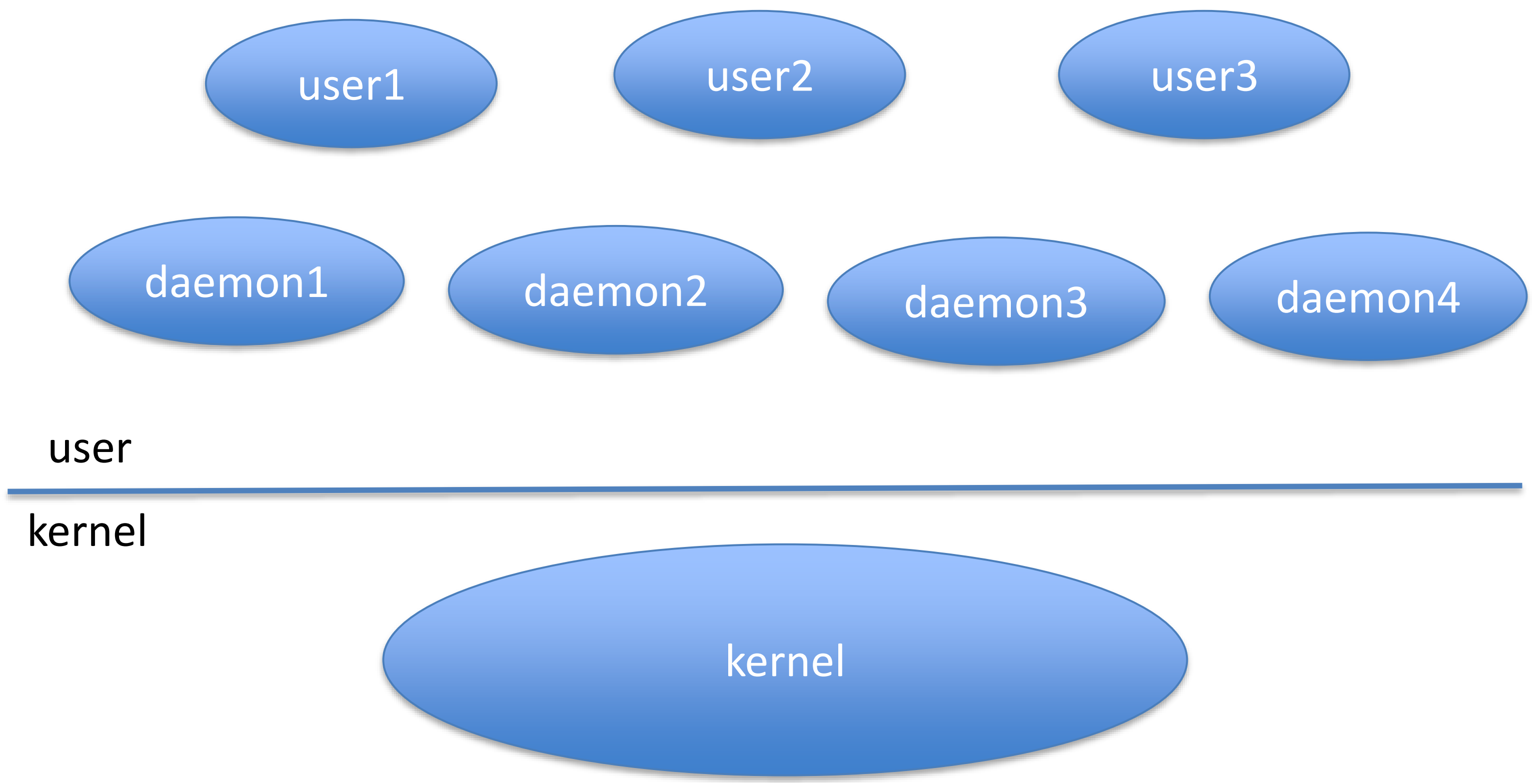
Key Components

- Process management
  - CPU ➔ processes
- Memory management
  - Memory ➔ address spaces
- File systems
  - Disk, SSD ➔ files

User/Kernel Mode

- User mode:
  - Applications
  - System programs
- Kernel mode:
  - OS kernel

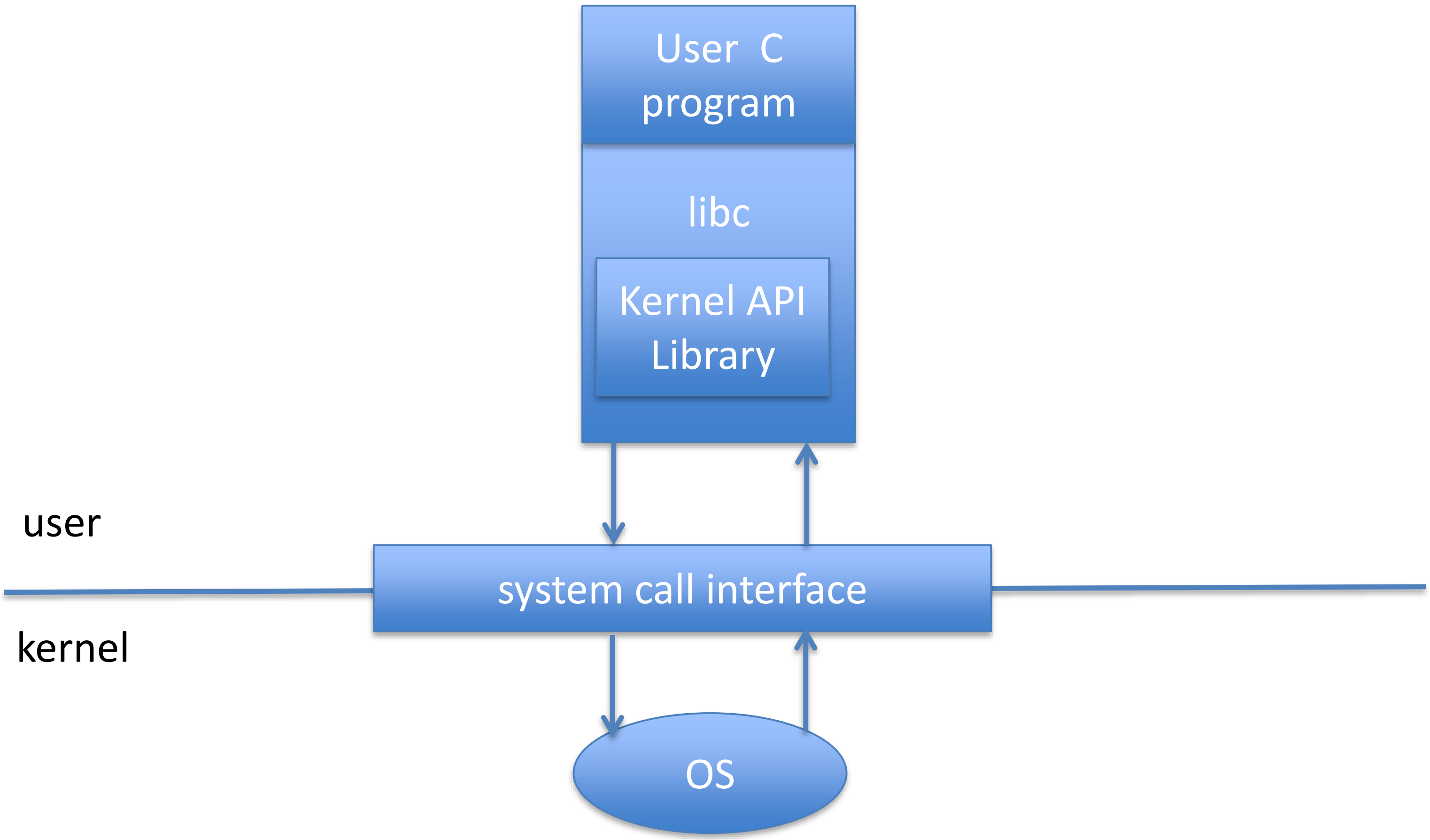
User/Kernel Mode



System Calls

- Only way for application to call kernel
- Special instruction
- Often wrapped by kernel API, libc

Interaction Application/Kernel



Kernel is Event-Driven Program

- Nothing to do

} Do nothing

Kernel is Event-Driven Program

- Nothing to do

} Do nothing

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

} Start running

Lecture 2
Process Management

Pamela Delgado

February 27, 2019

(slides Willy Zwaenepoel)

Key Concepts

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

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

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()

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() returns
}
else {
    exec( filename )
}
```

Question about fork-exec

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

does it make sense
to write code here?

Outline of Linux Shell

```
forever {
    read from input
    if( logout) exit()
    if ( pid = fork() ) {
        wait()
    }
    else {
        exec( filename )
    }
}
```

Operation

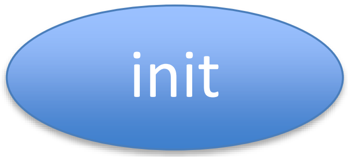
- New command line (!= logout)
  - Shell forks a new process and waits
  - Child executes program on command line

The Linux Process Tree

Boot

- First process after boot is the init process
- Happens by black magic

User logs in

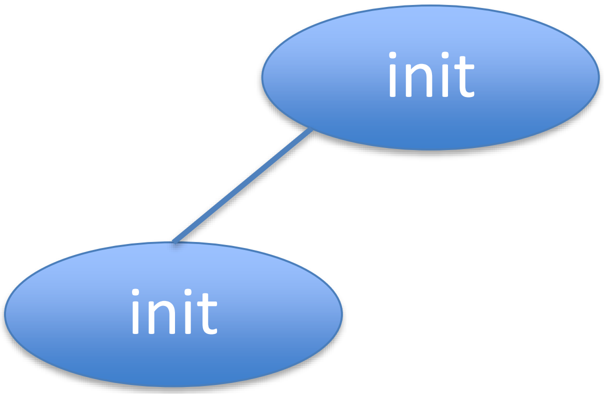


User logs in

- Init forks and waits
- Child execs shell

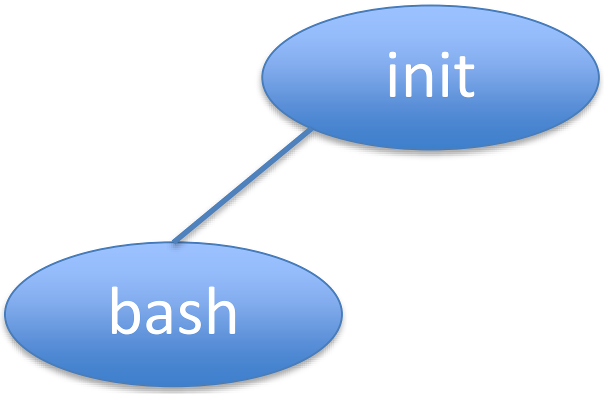
User logs in

- Init forks and waits
- Child execs shell

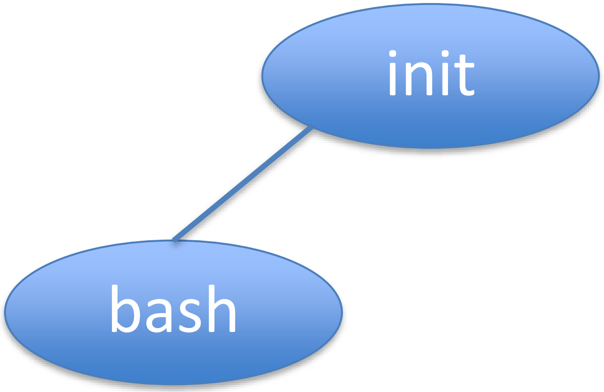


User logs in

- Init forks and waits
- Child execs shell

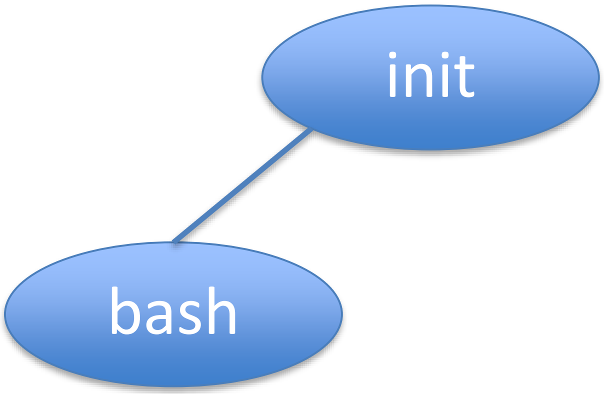


User runs make



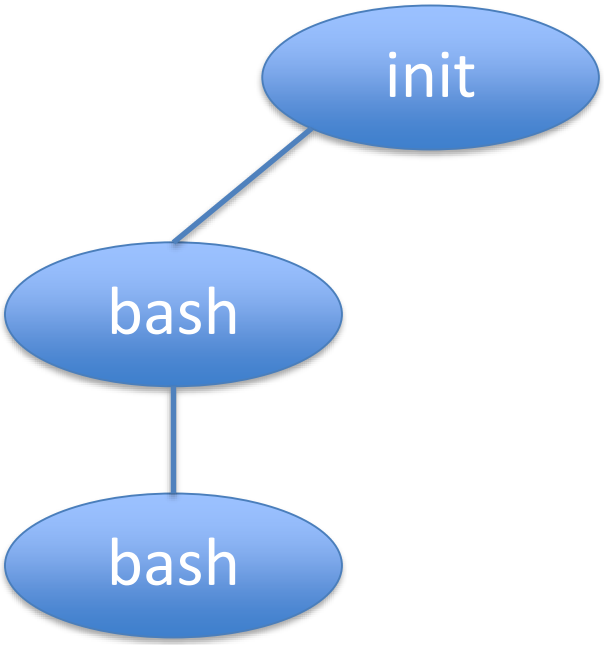
User runs make

- Shell forks and waits
- Child execs make



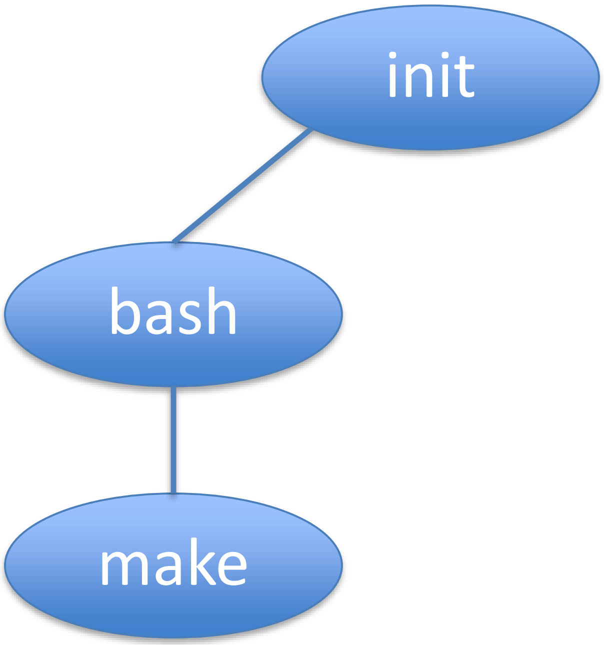
User runs make

- Shell forks and waits
- Child execs make

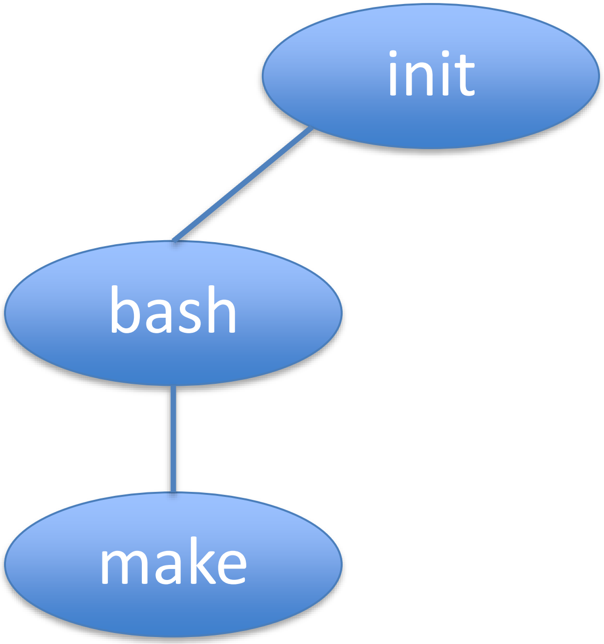


User runs make

- Shell forks and waits
- Child execs make

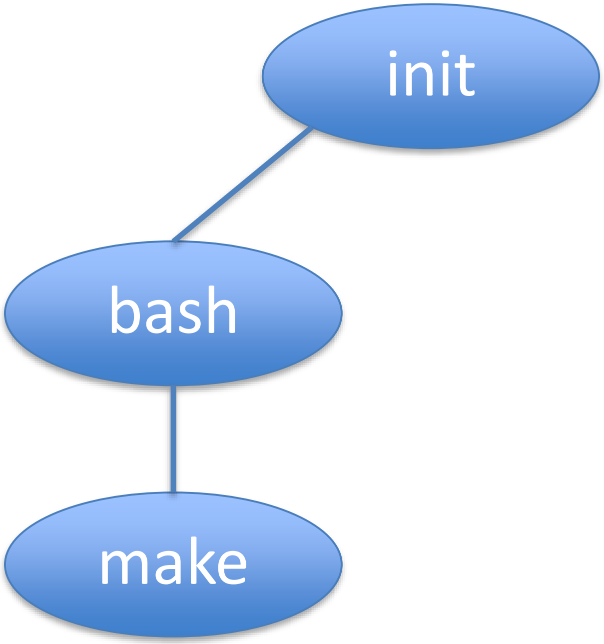


Another user logs in



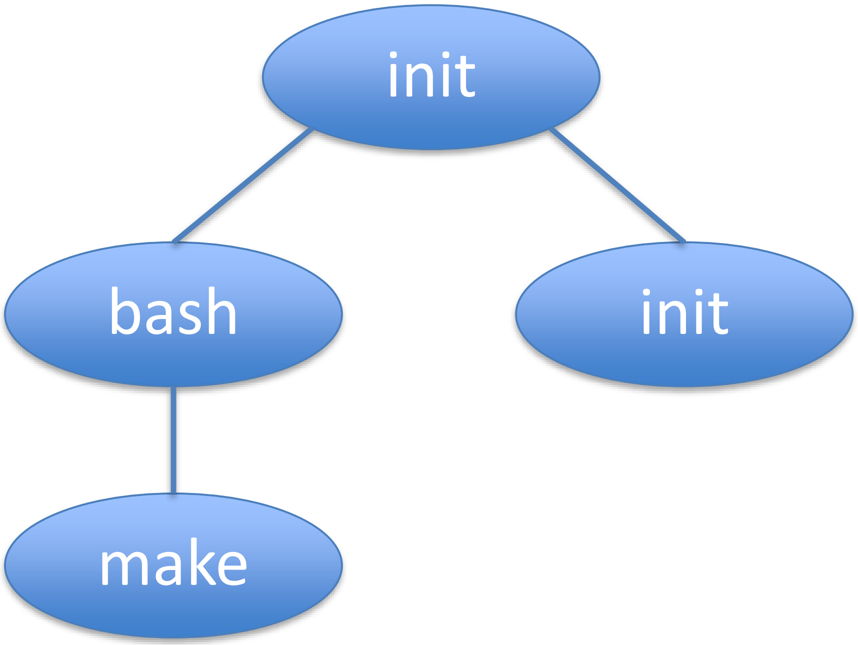
Another user logs in

- Init forks and waits
- Child execs shell



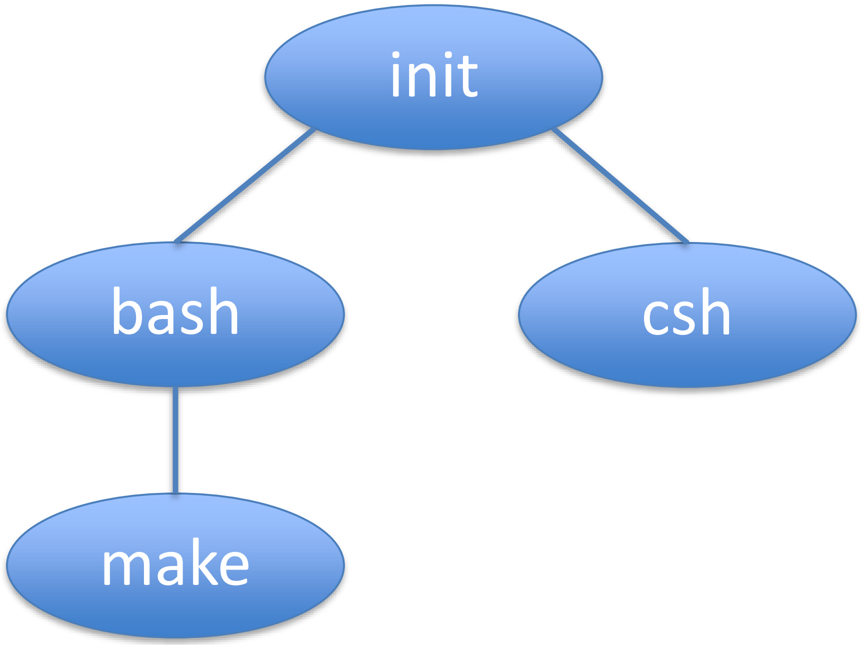
Another user logs in

- Init forks and waits
- Child execs shell

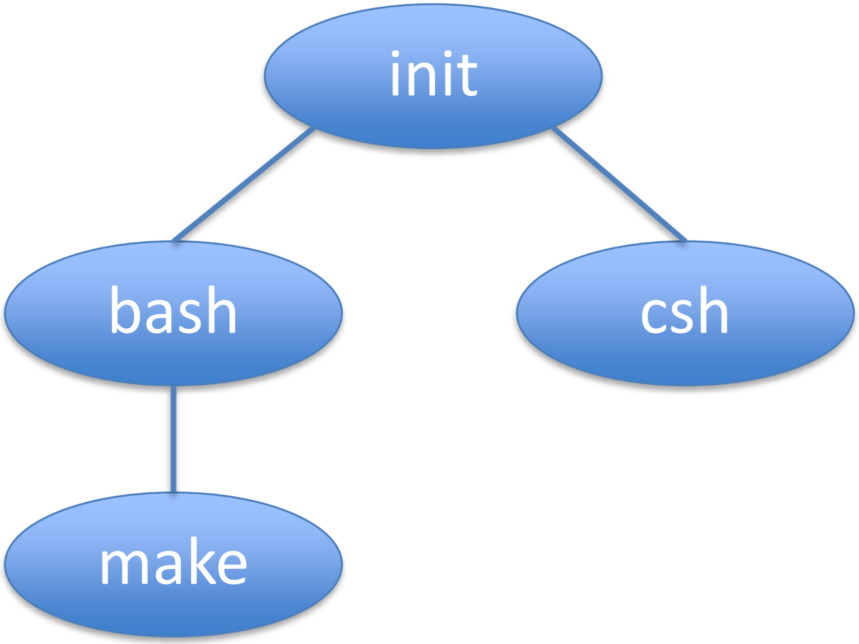


Another user logs in

- Init forks and waits
- Child execs shell

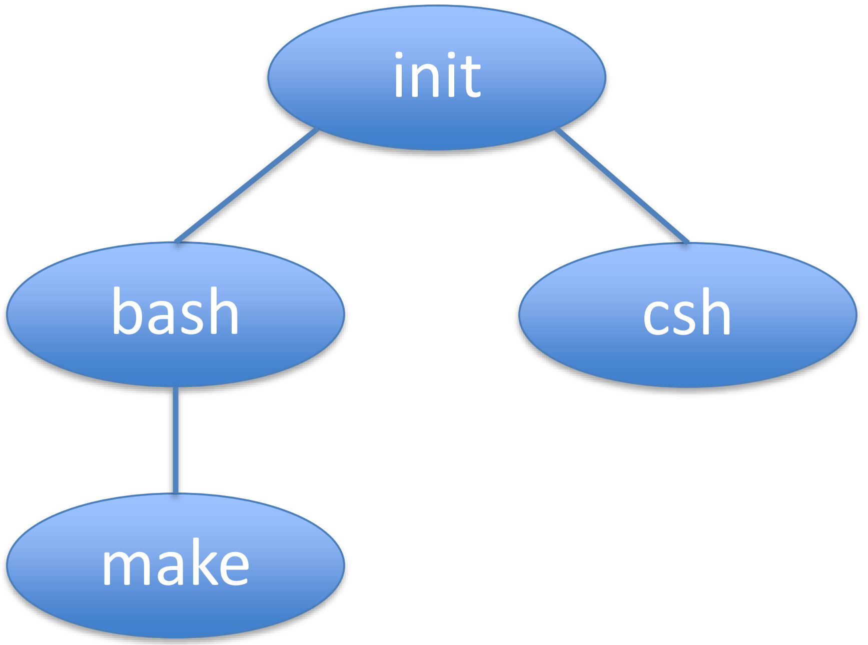


Make runs gcc



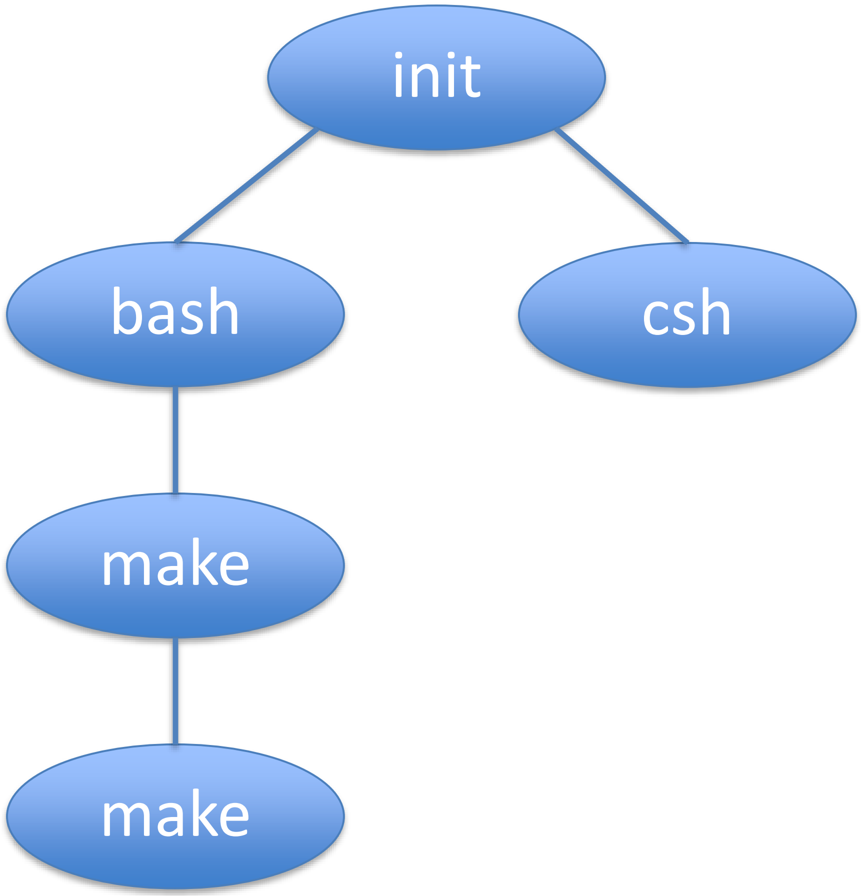
Make runs gcc

- Make forks and waits
- Child execs gcc



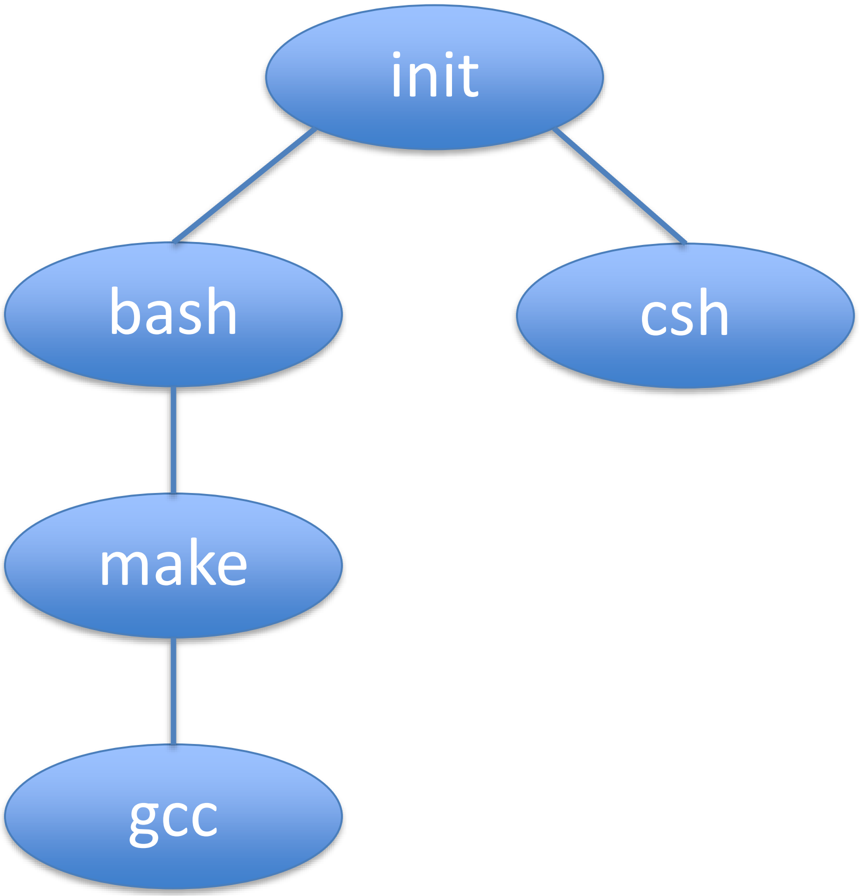
Make runs gcc

- Make forks and waits
- Child execs gcc



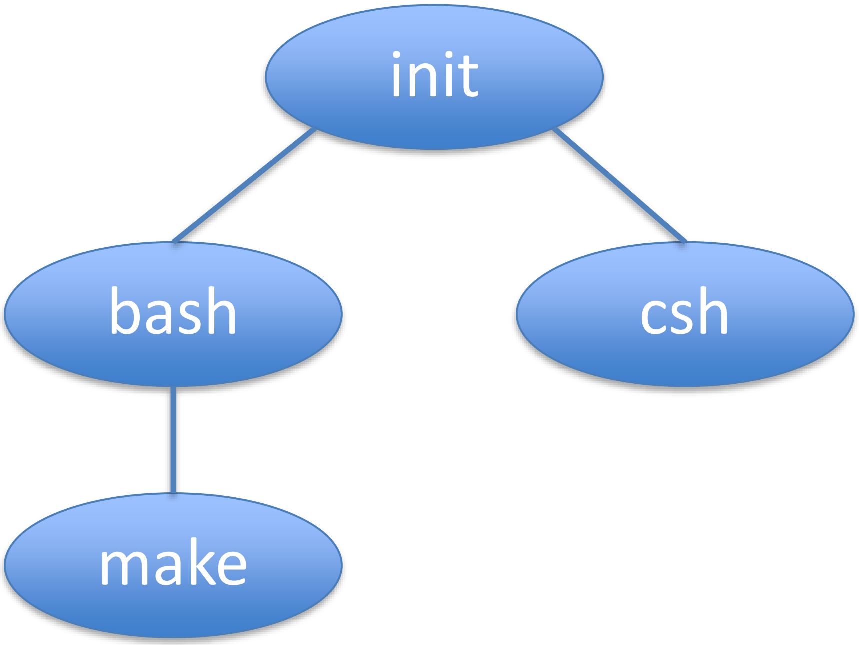
Make runs gcc

- Make forks and waits
- Child execs gcc



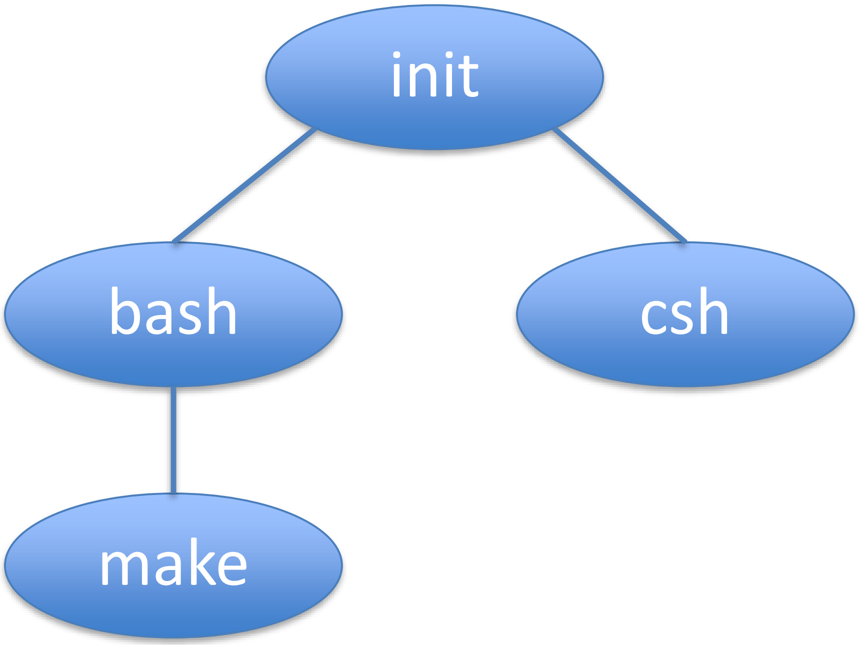
Gcc finishes

- Gcc exits
- Make returns from wait



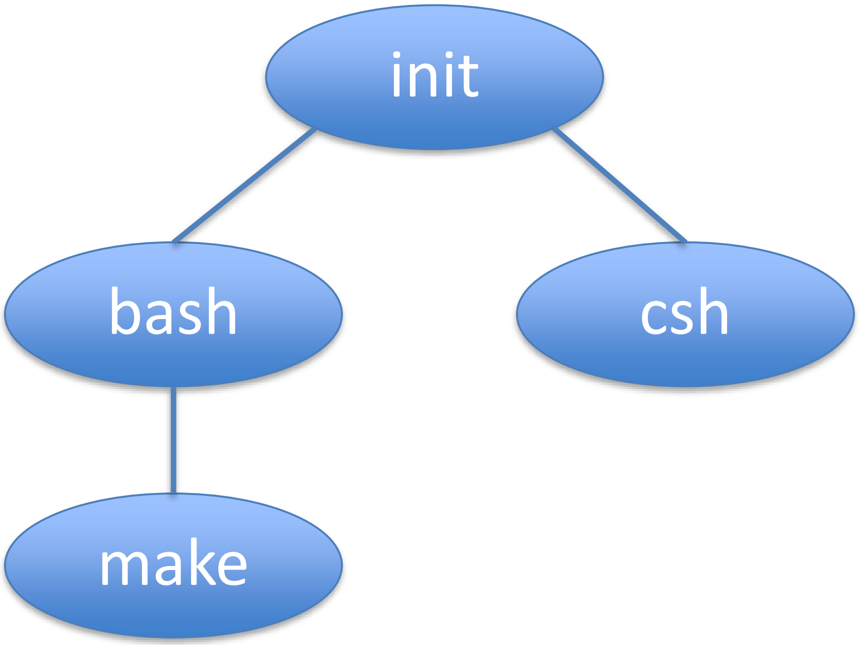
Gcc finishes

- Gcc exits
- Make returns from wait

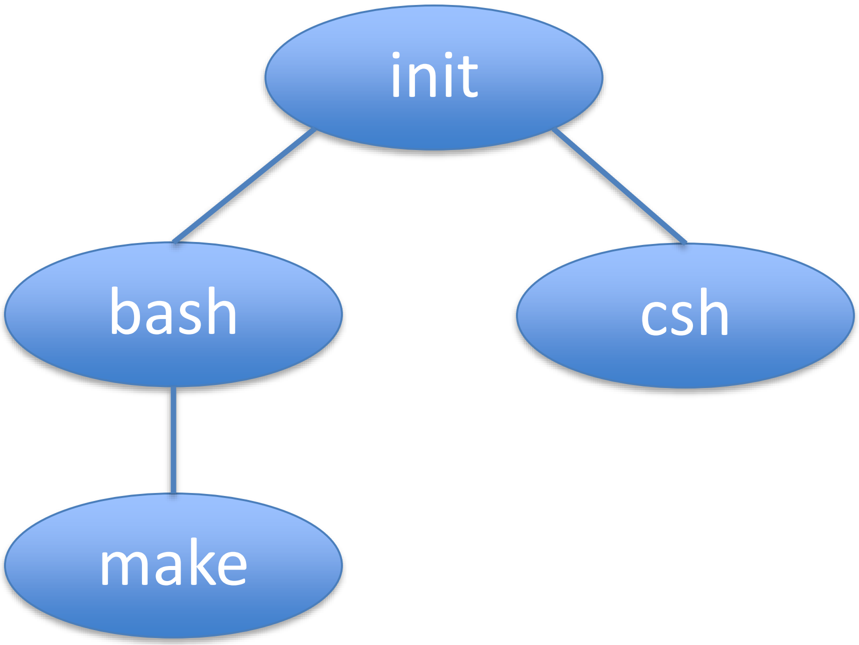


Gcc finishes

- Gcc exits
- Make returns from wait

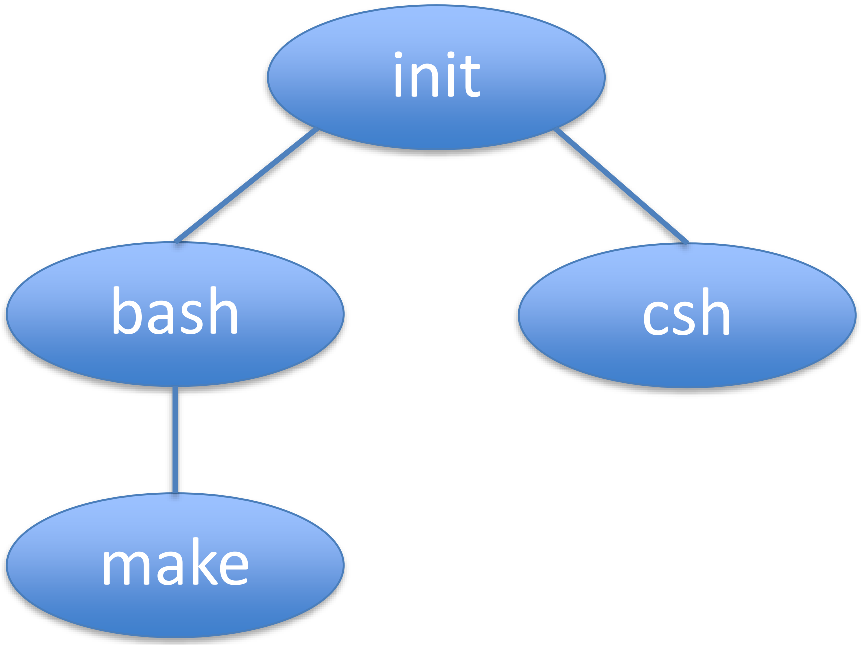


Second user logs out



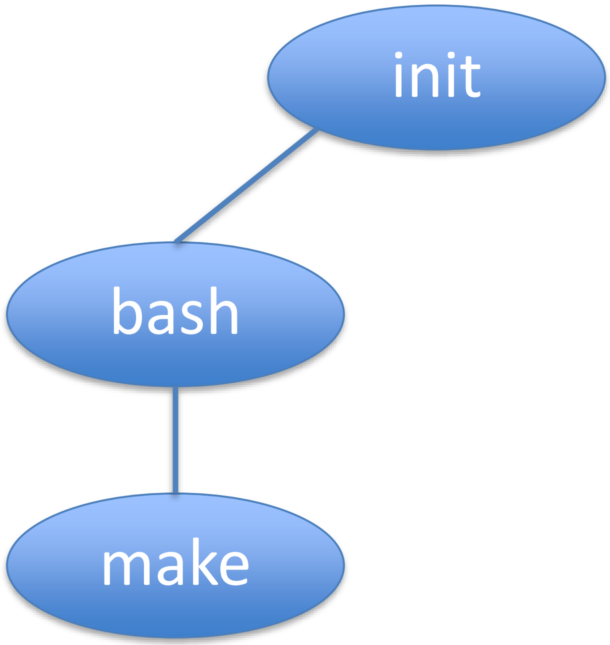
Second user logs out

- Csh exits
- Init returns from wait



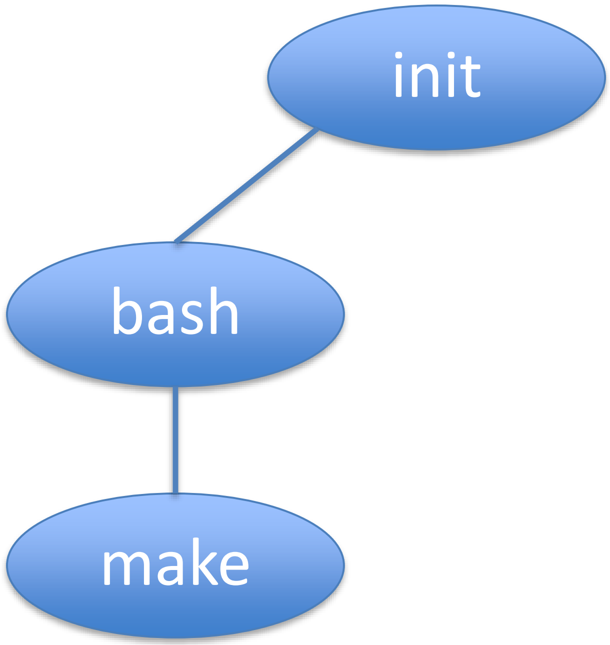
Second user logs out

- Csh exits
- Init returns from wait



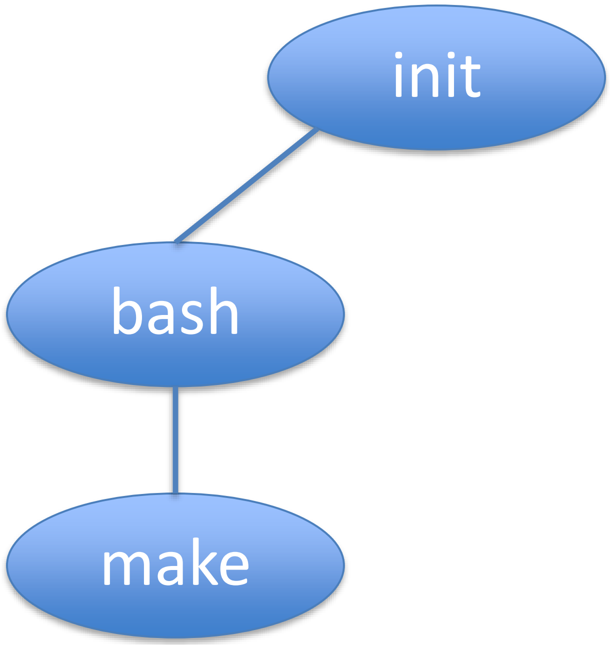
Second user logs out

- Csh exits
- Init returns from wait



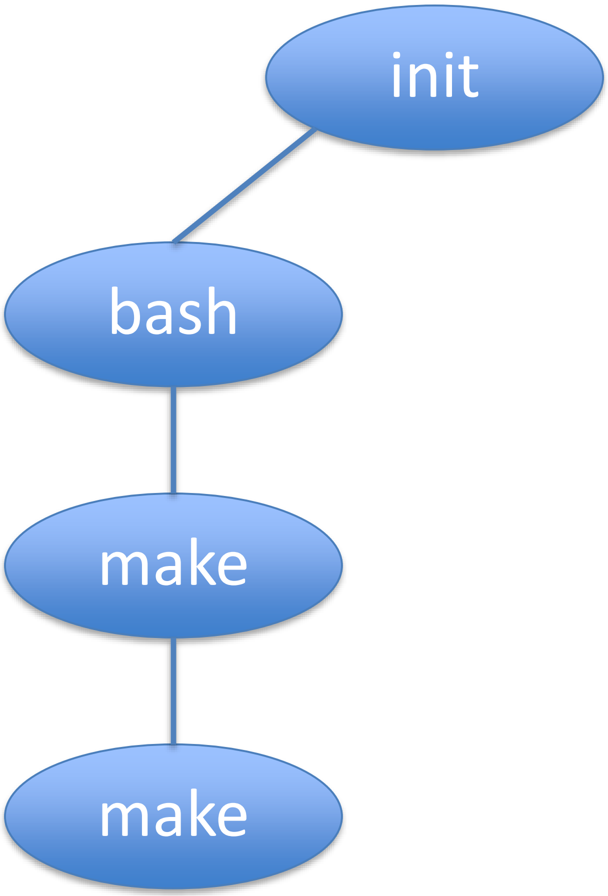
Make runs cp

- Make forks and waits
- Child execs cp



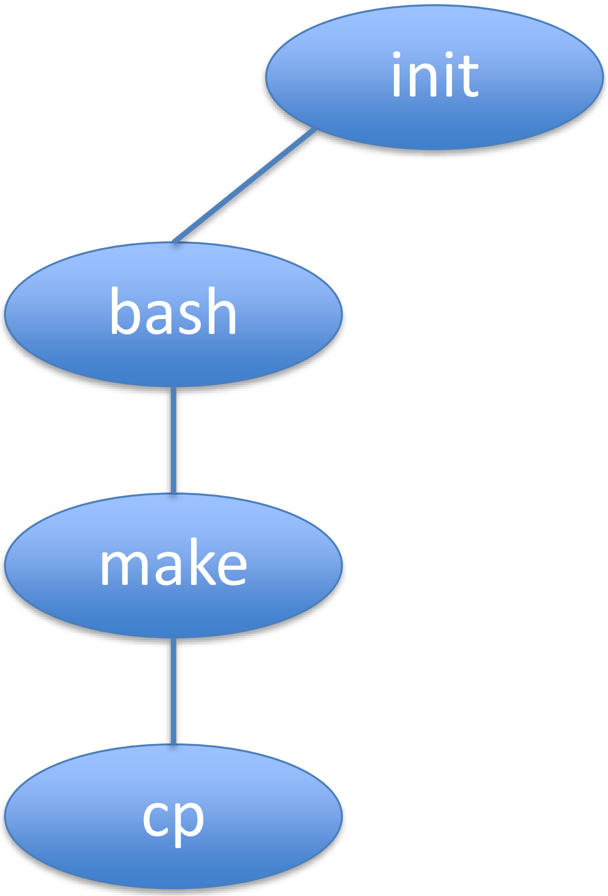
Make runs cp

- Make works and waits
- Child execs cp



Make runs cp

- Make forks and waits
- Child execs cp



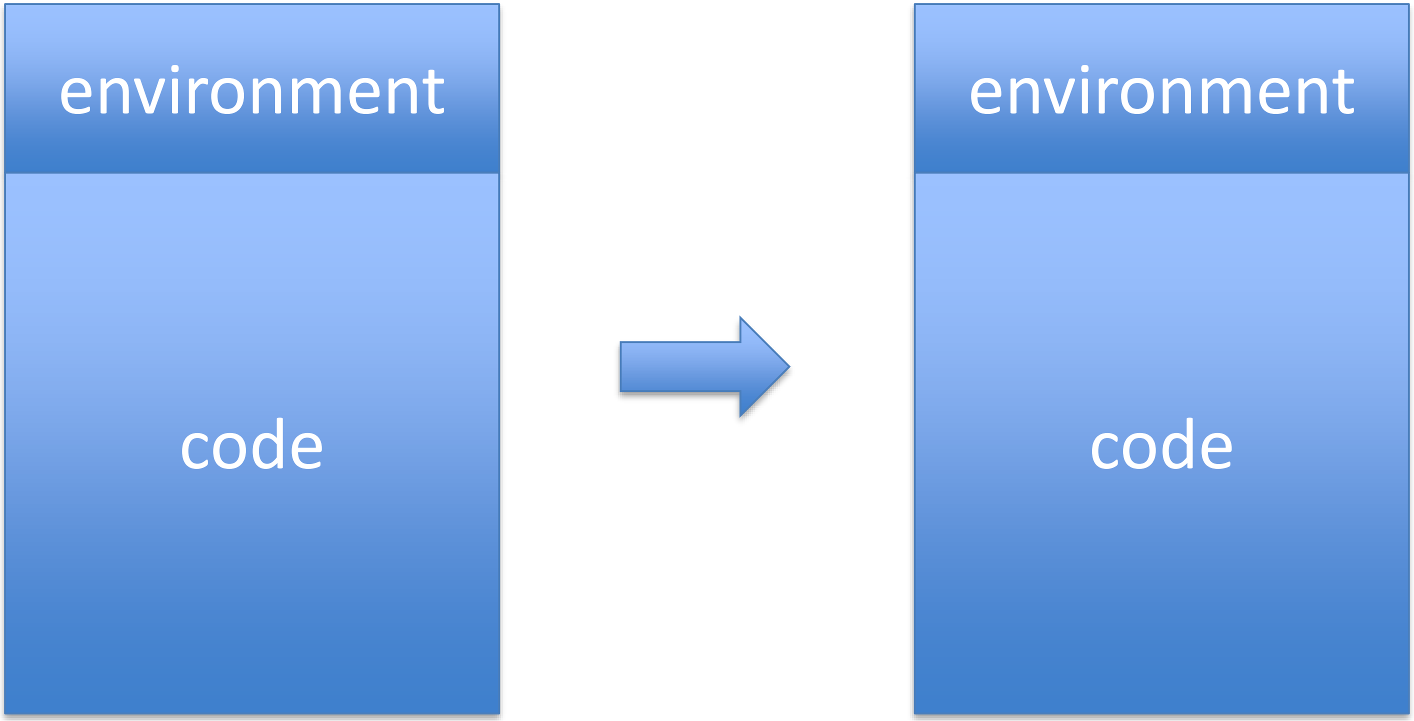
Why fork+exec vs. create?

Process = Environment + Code

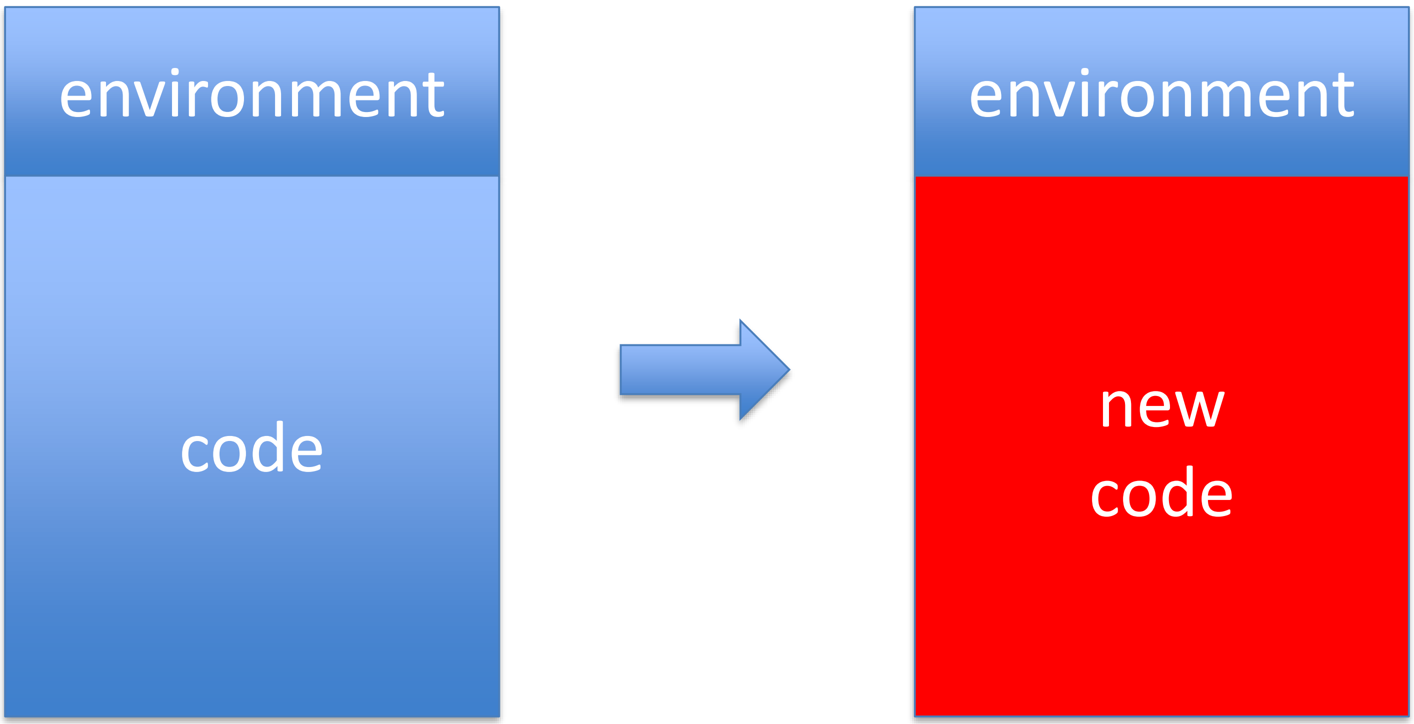
- Environment includes:
  - Ownership
  - Open files
  - Values of environment variables

Process = Environment + Code

After a fork()



After an exec() in the Child



Advantage

- Child automatically inherits environment

Question about fork-exec

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

does it make sense
to write code here?

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
- See exercises for details

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)

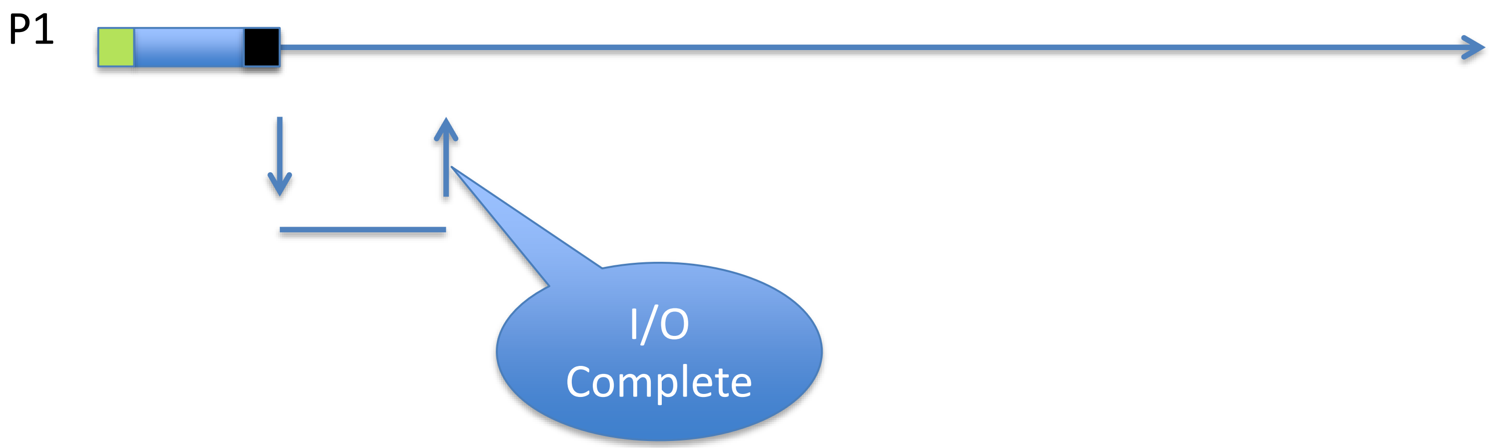
- Either it computes (uses the CPU)
- Or it does I/O (uses a device)

Single Process System

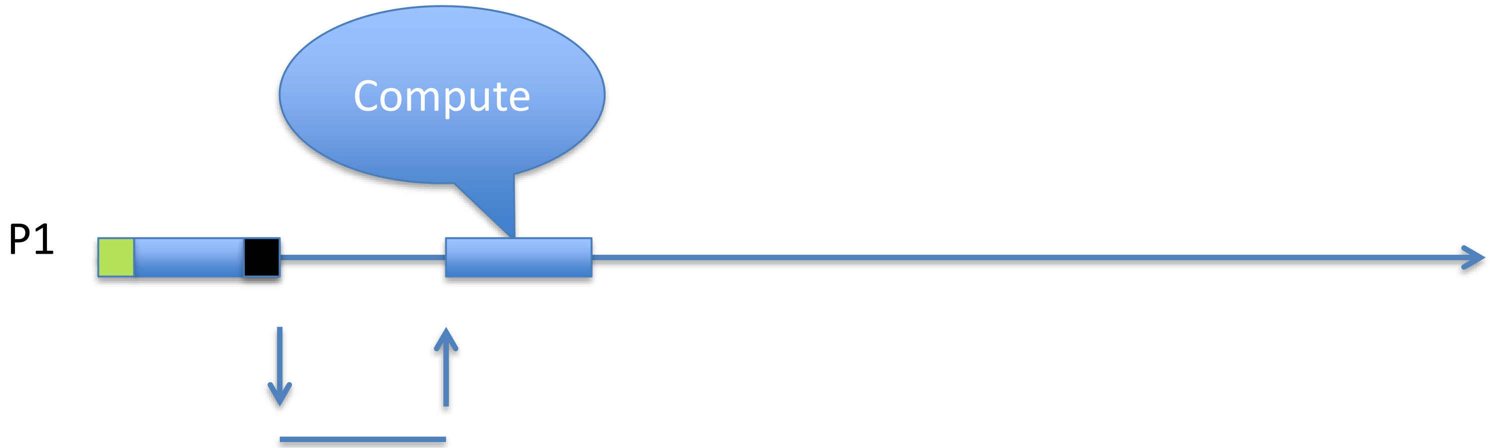
Single Process System

Single Process System

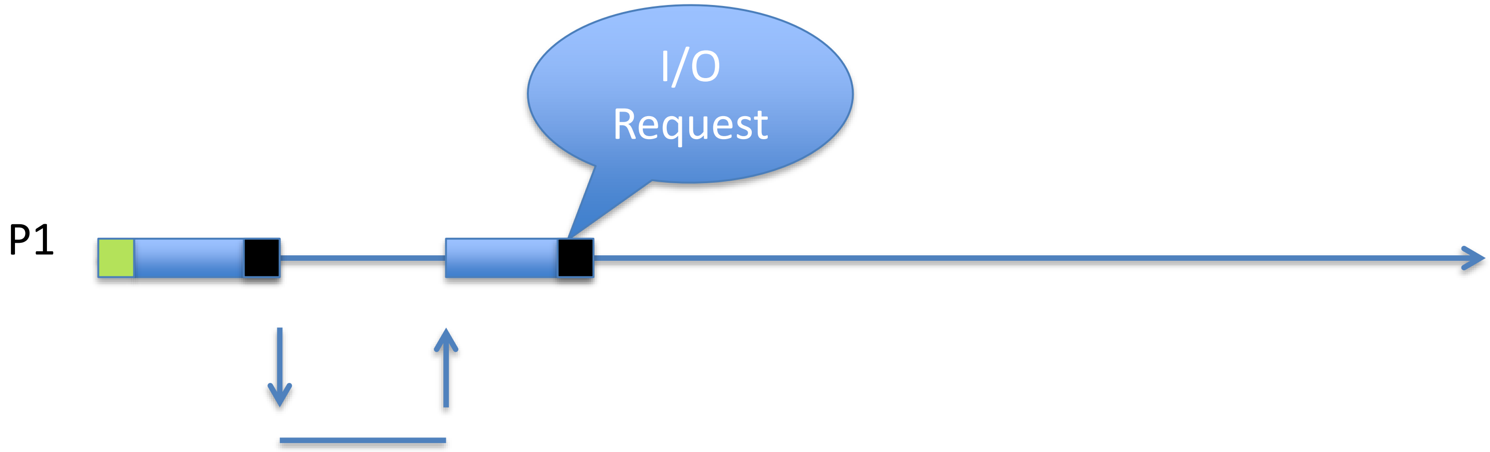
Single Process System



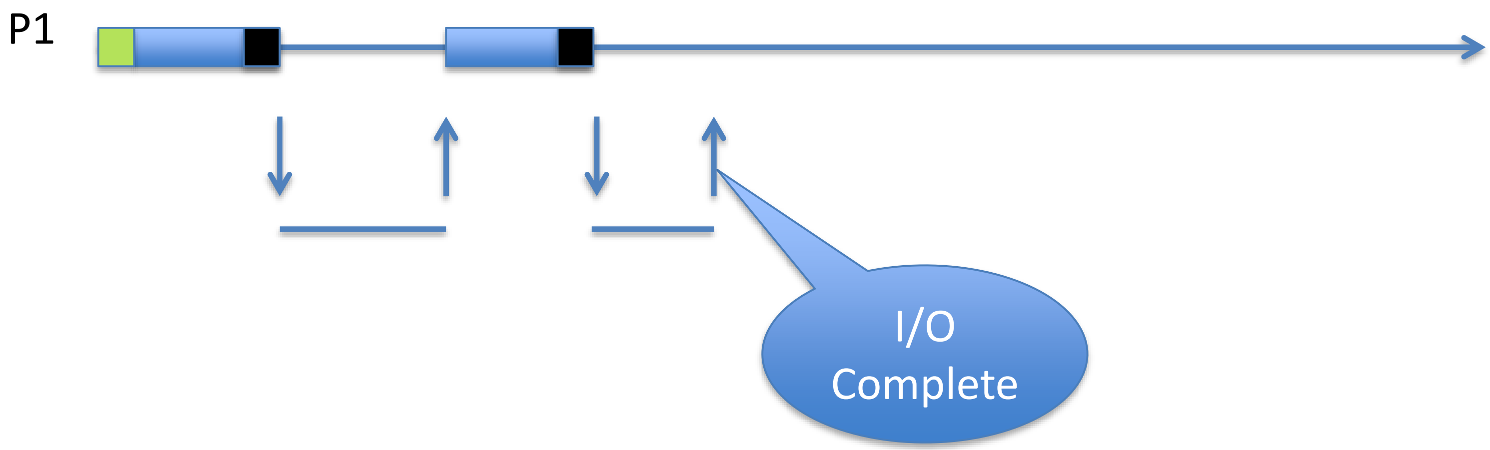
Single Process System



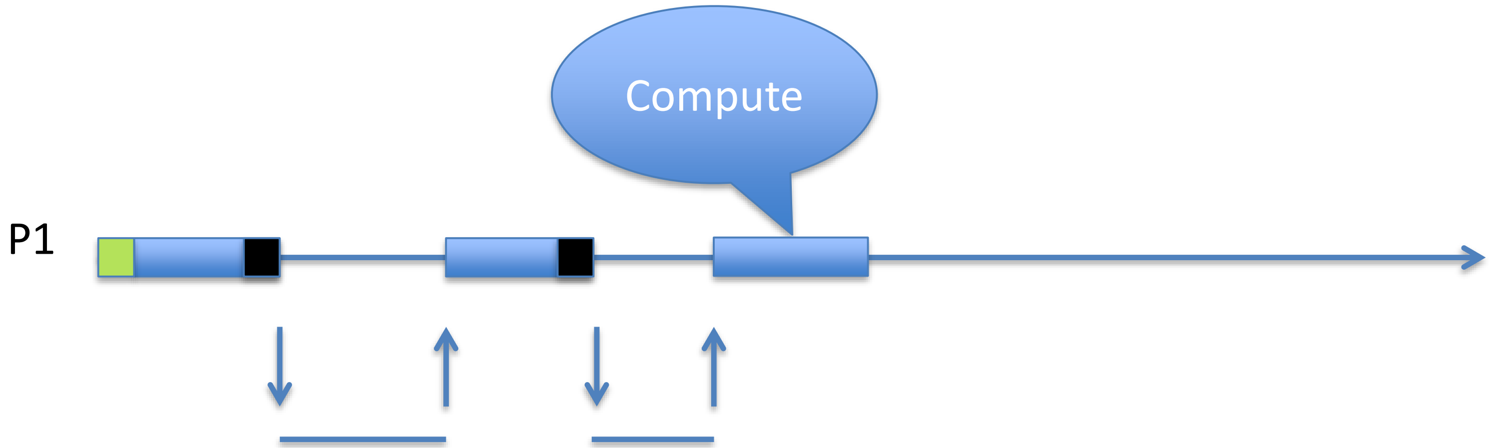
Single Process System



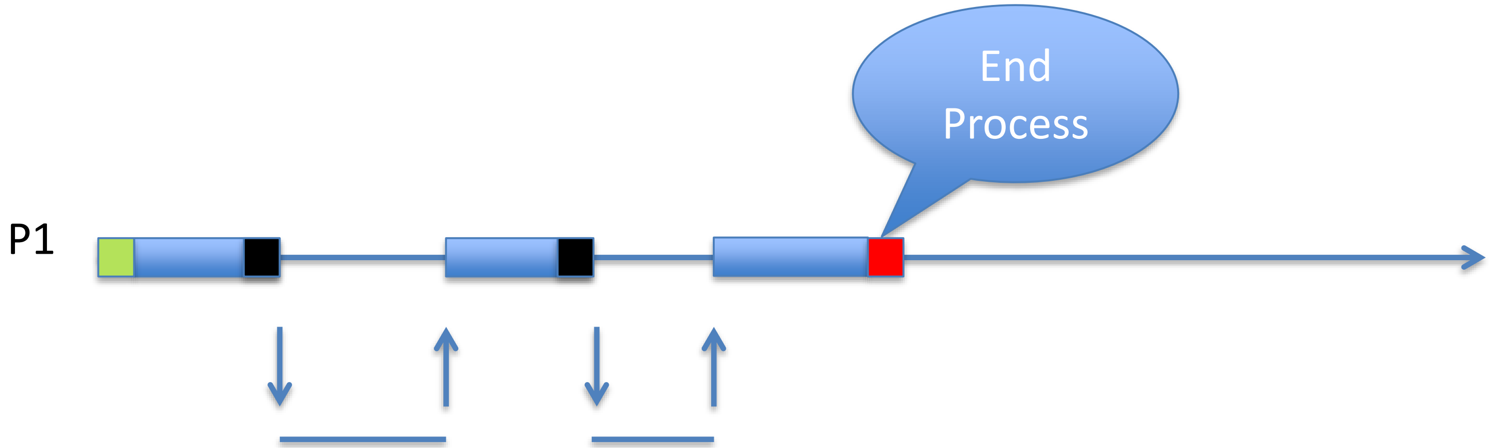
Single Process System



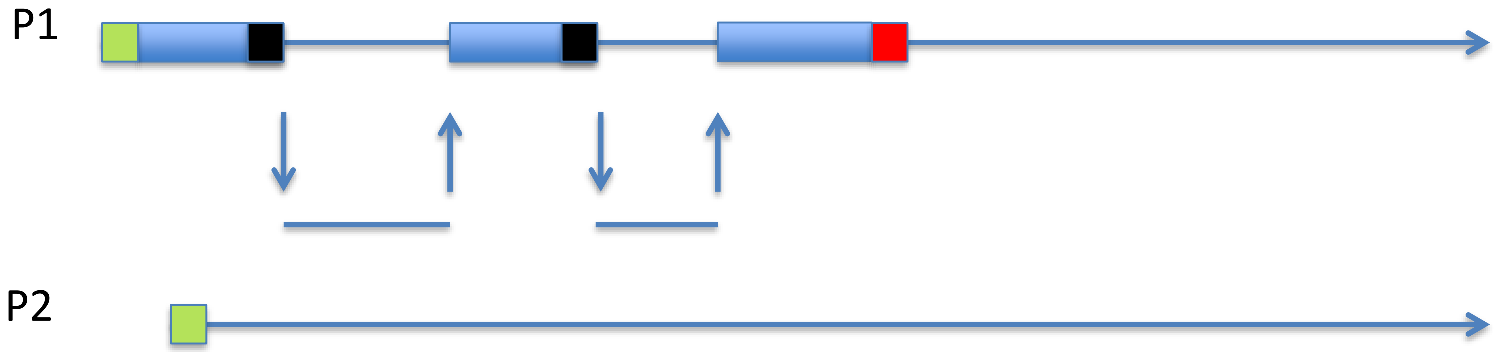
Single Process System



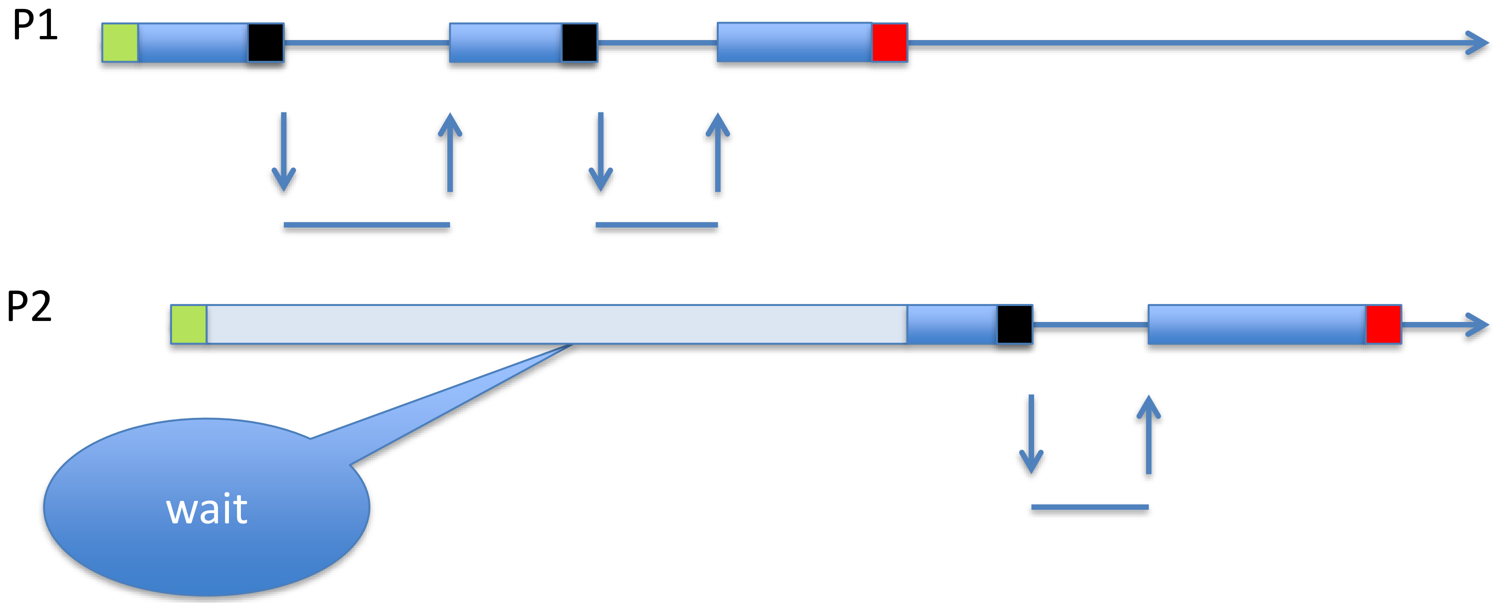
Single Process System



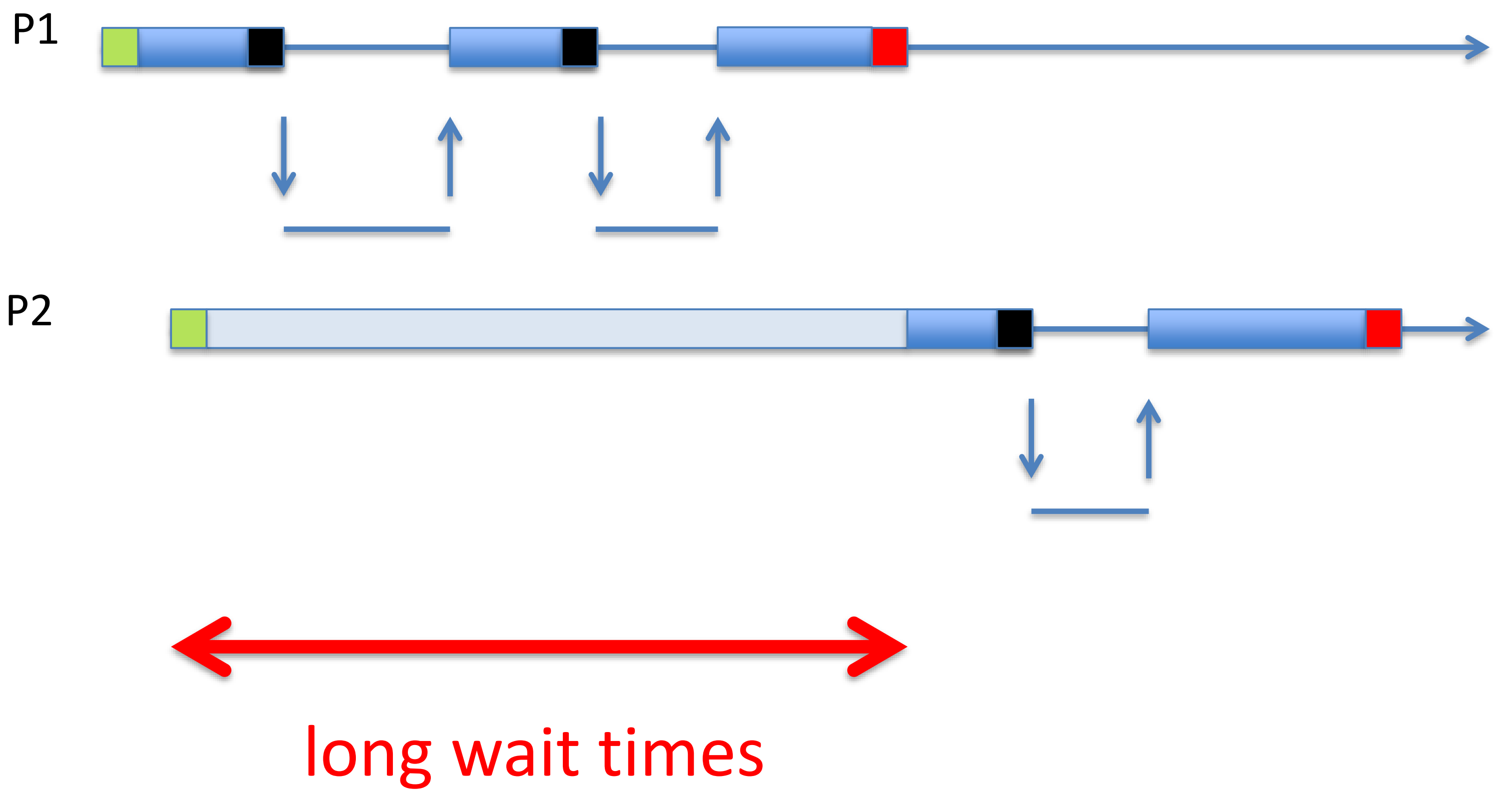
A Second Process



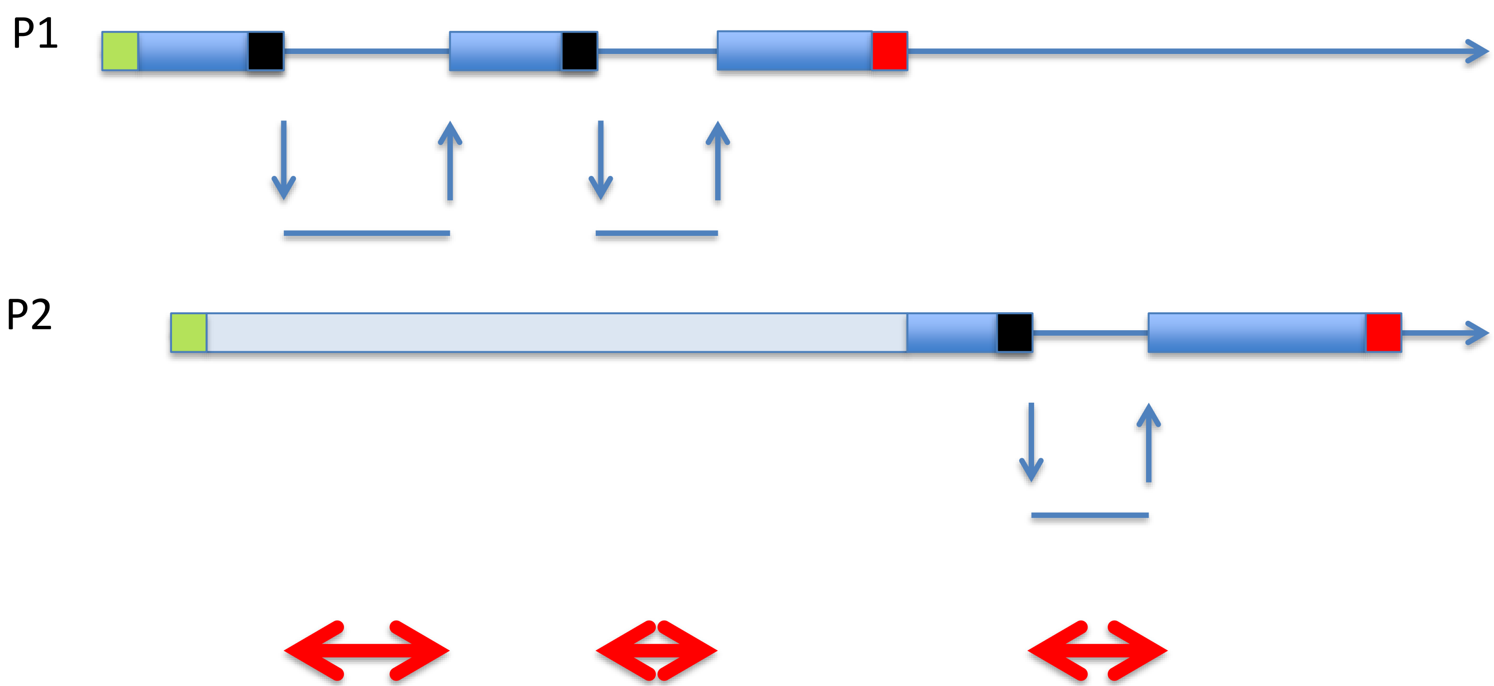
A Second Process



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

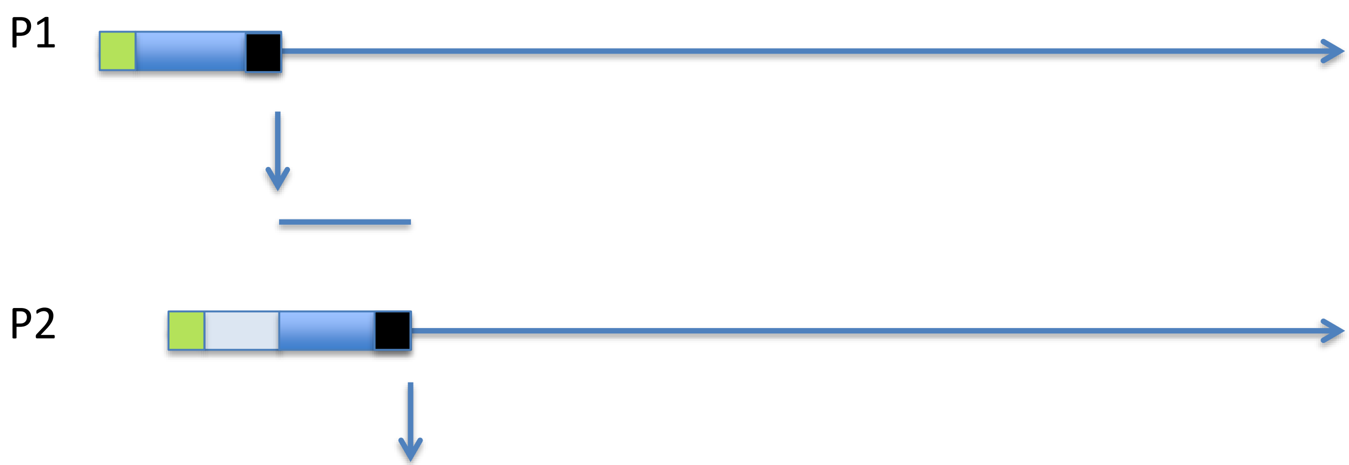
Multiprocess System

Multiprocess System

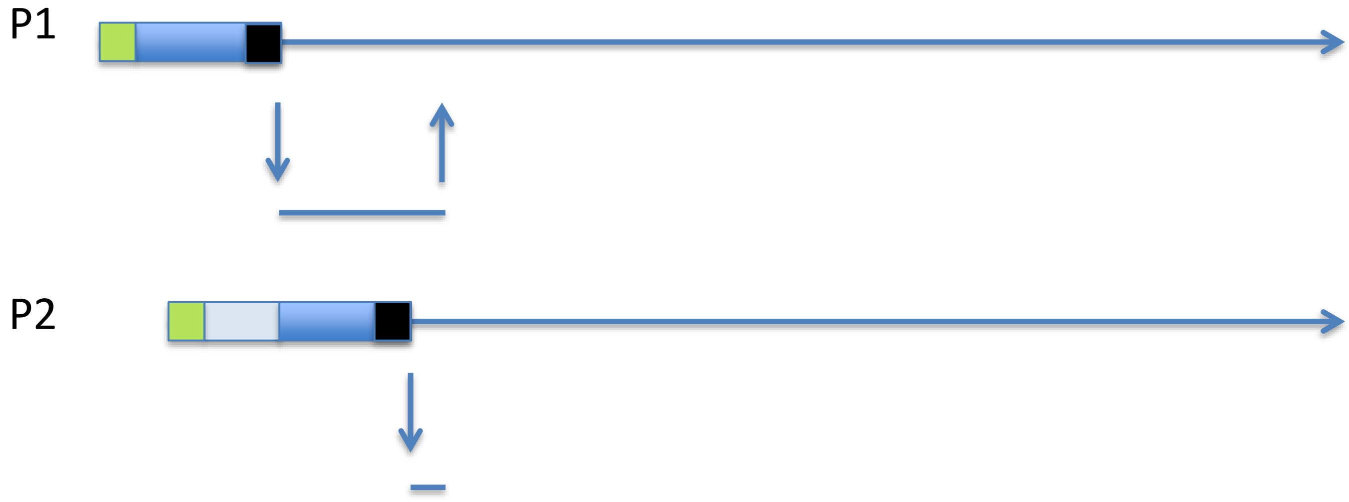
Multiprocess System

Multiprocess System

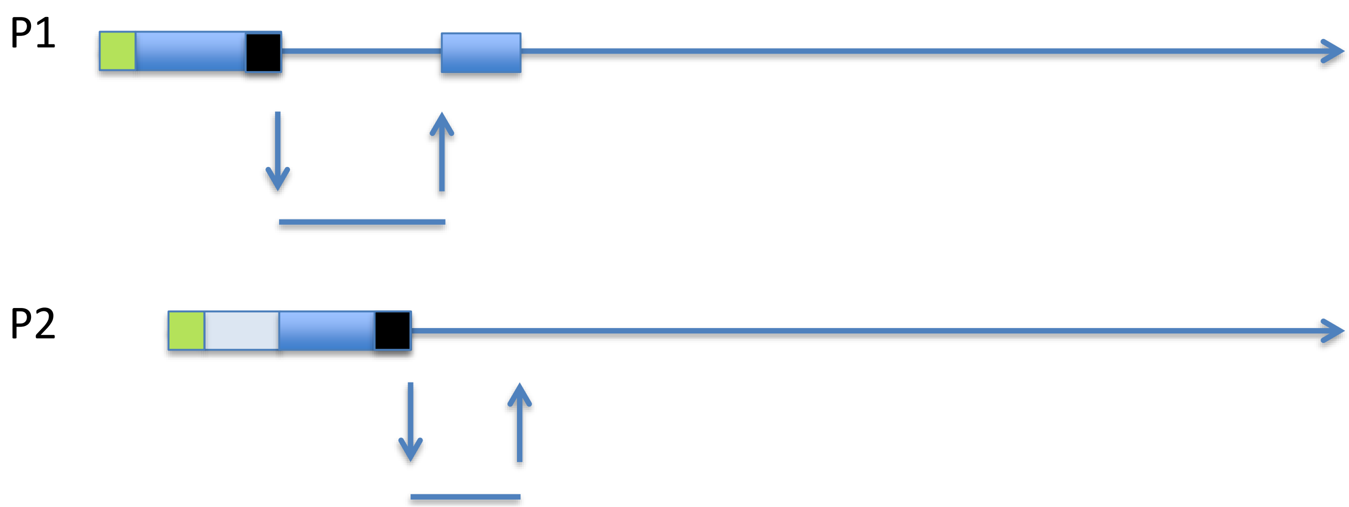
Multiprocess System



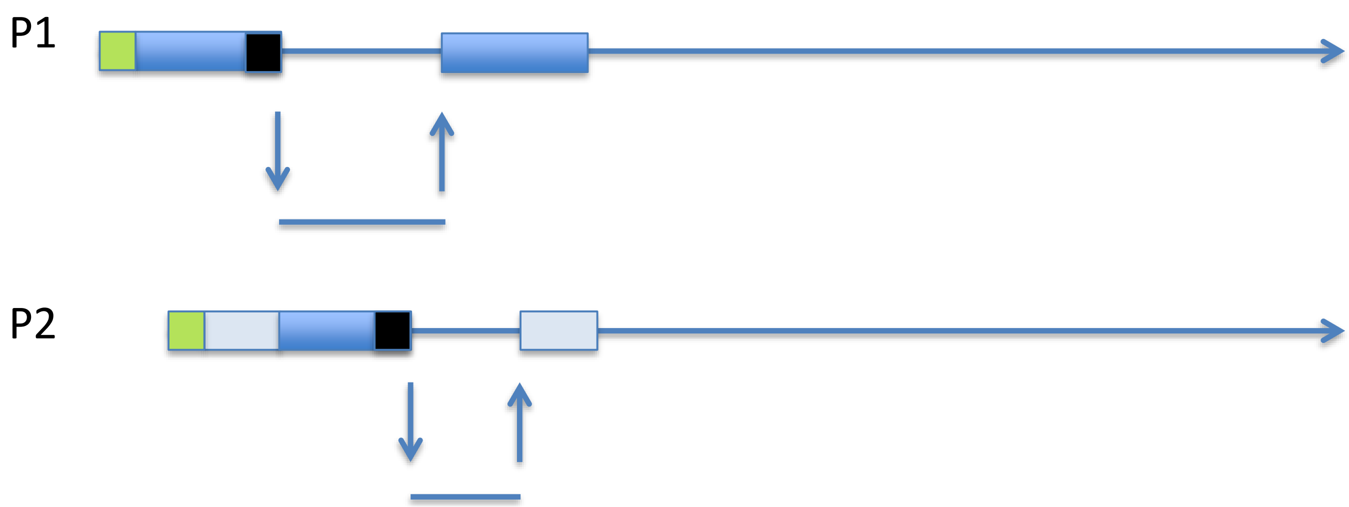
Multiprocess System



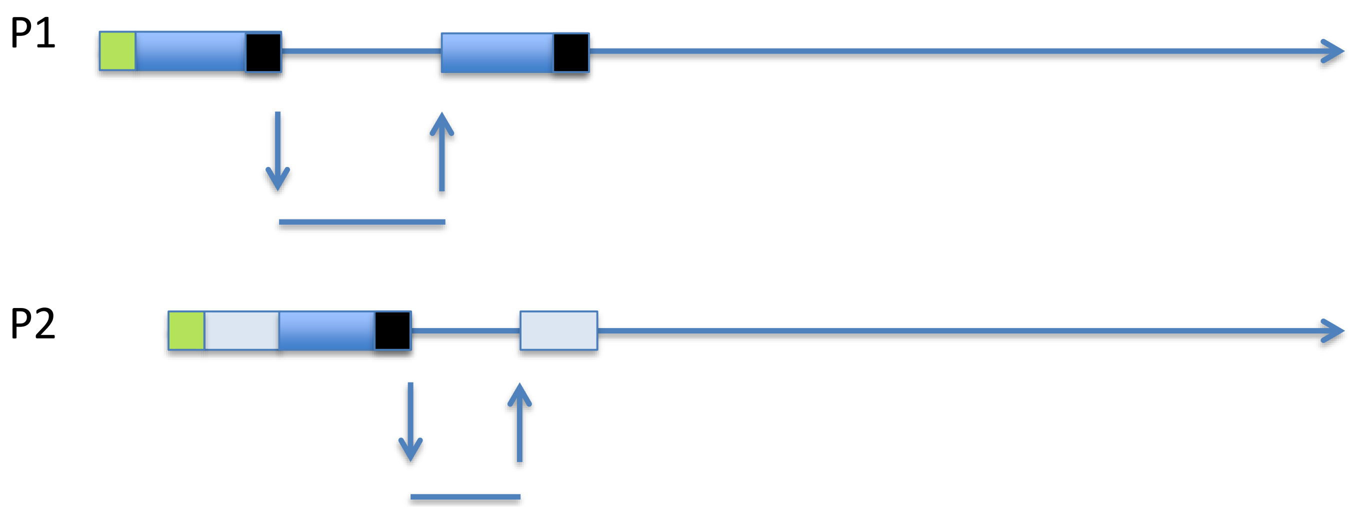
Multiprocess System



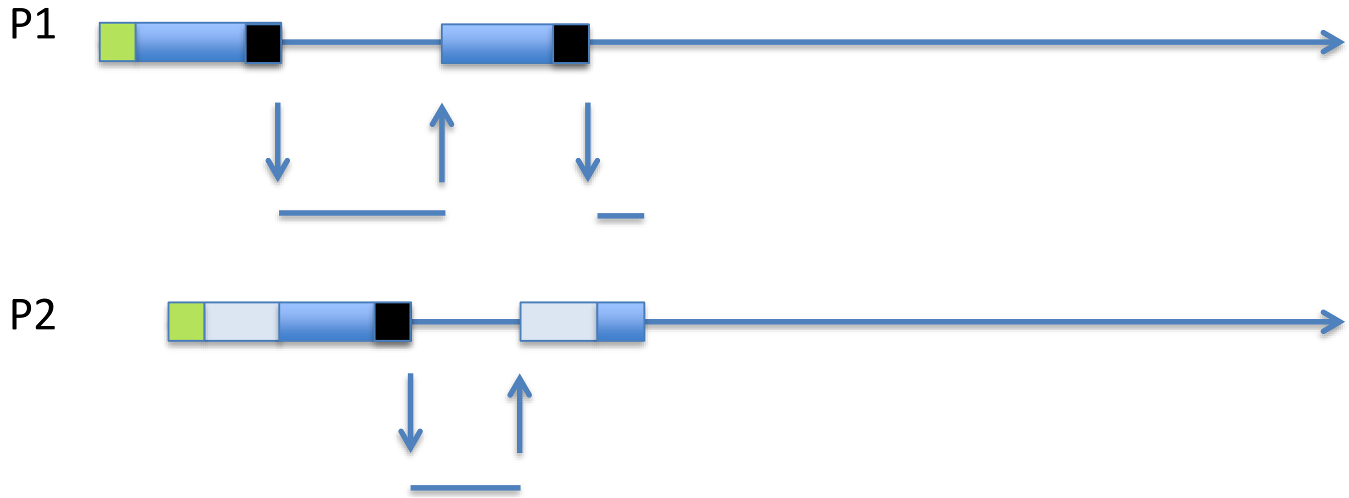
Multiprocess System



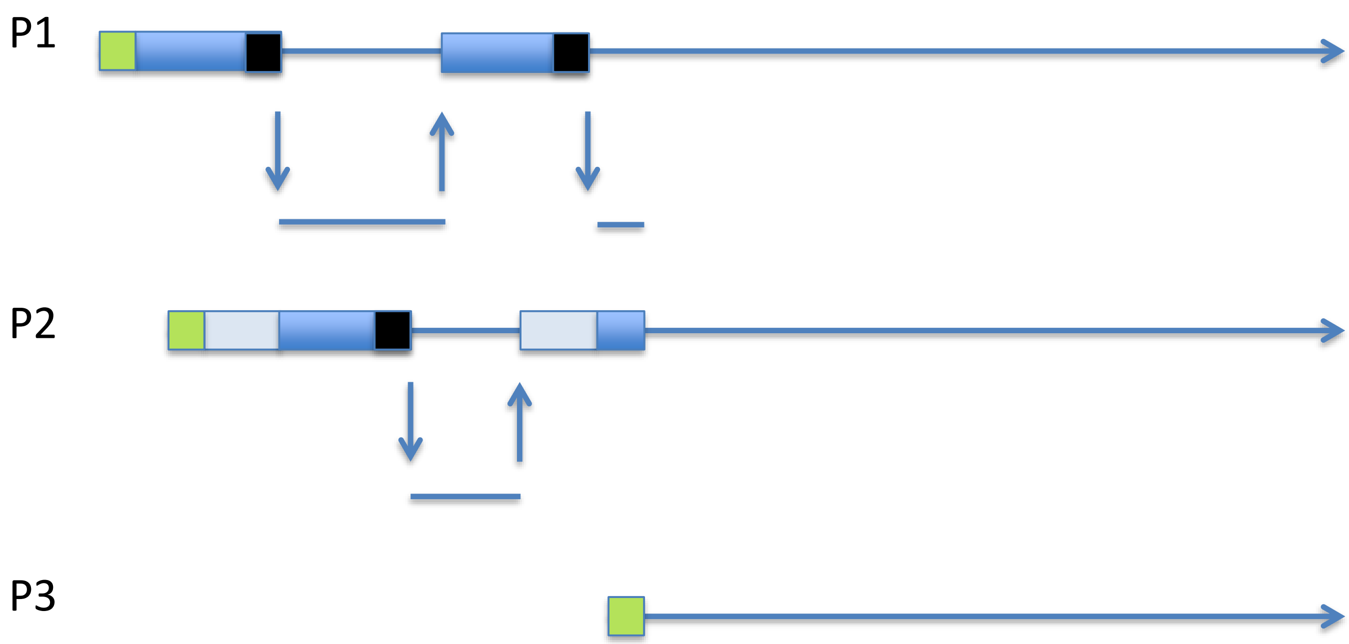
Multiprocess System



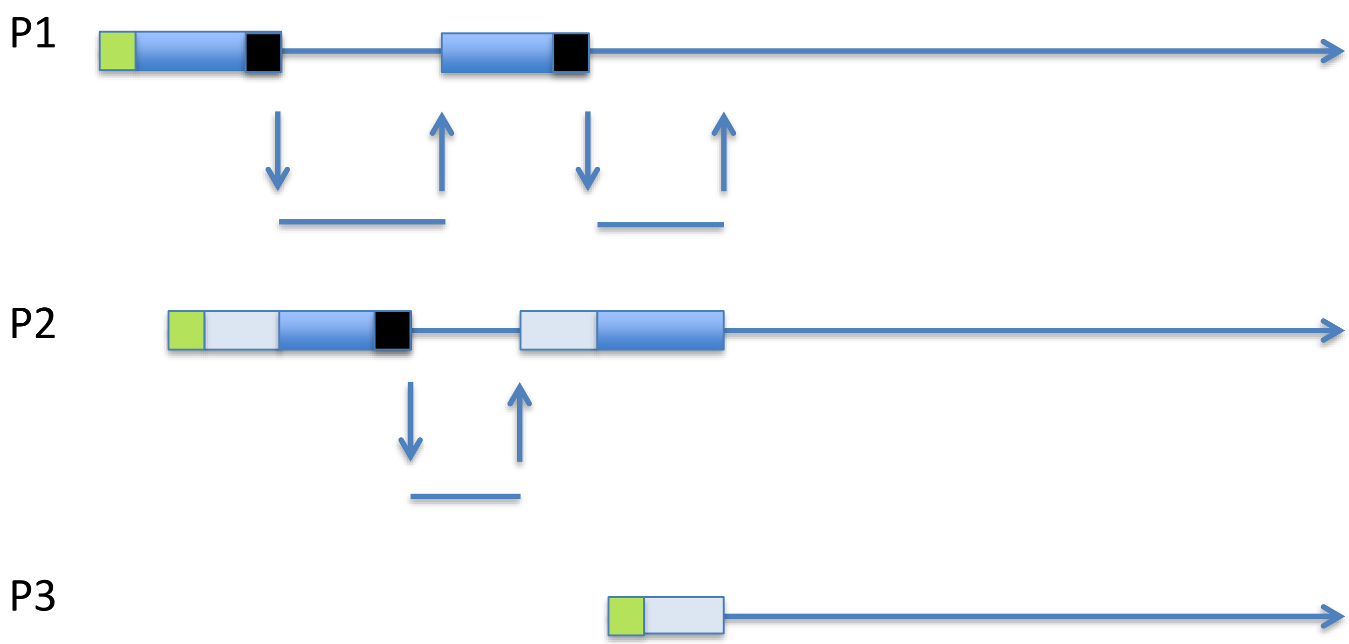
Multiprocess System



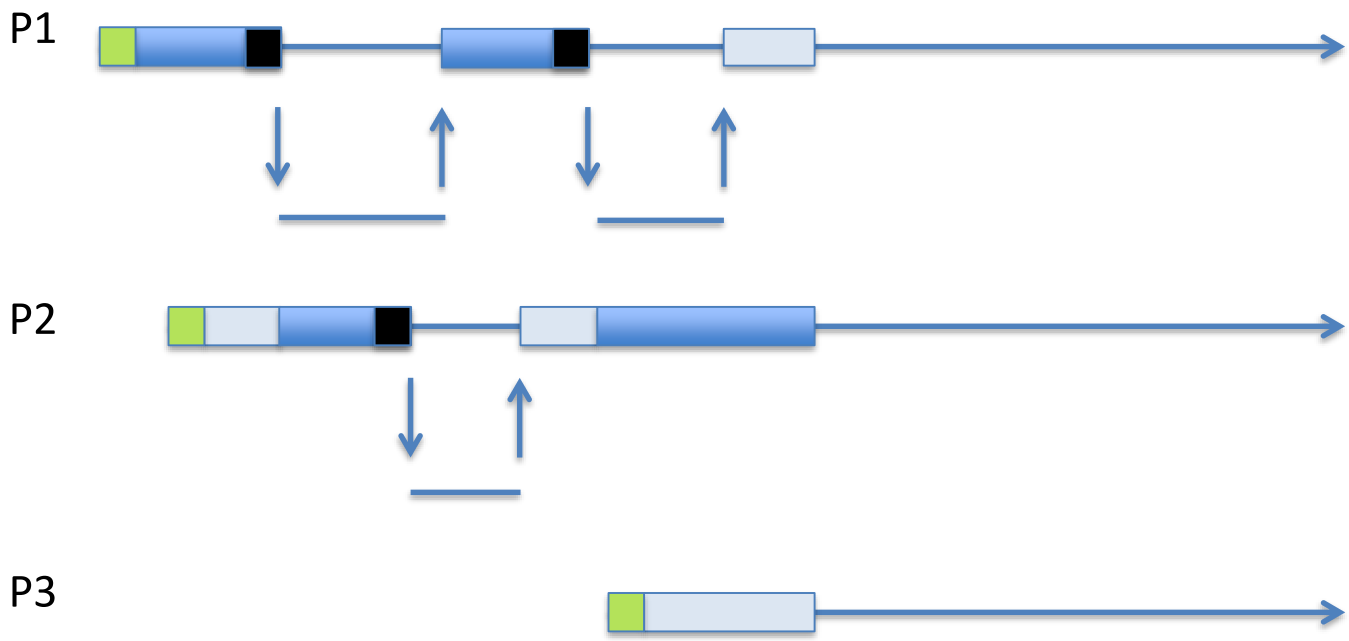
Multiprocess System



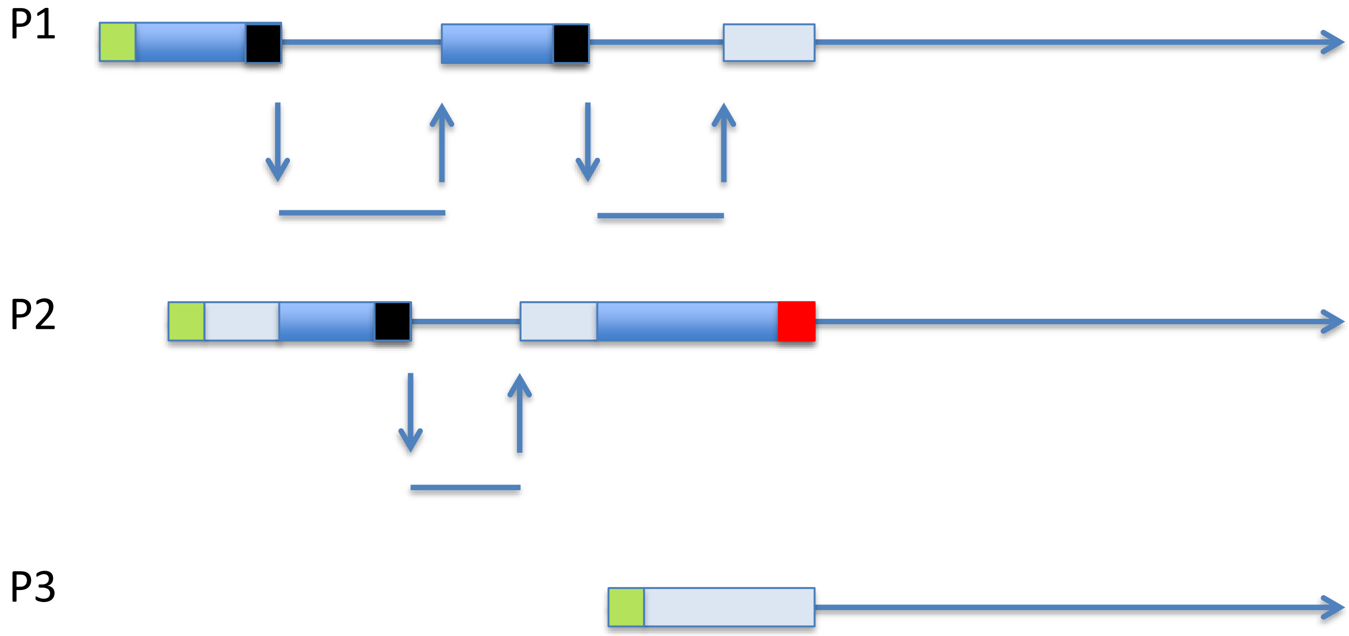
Multiprocess System



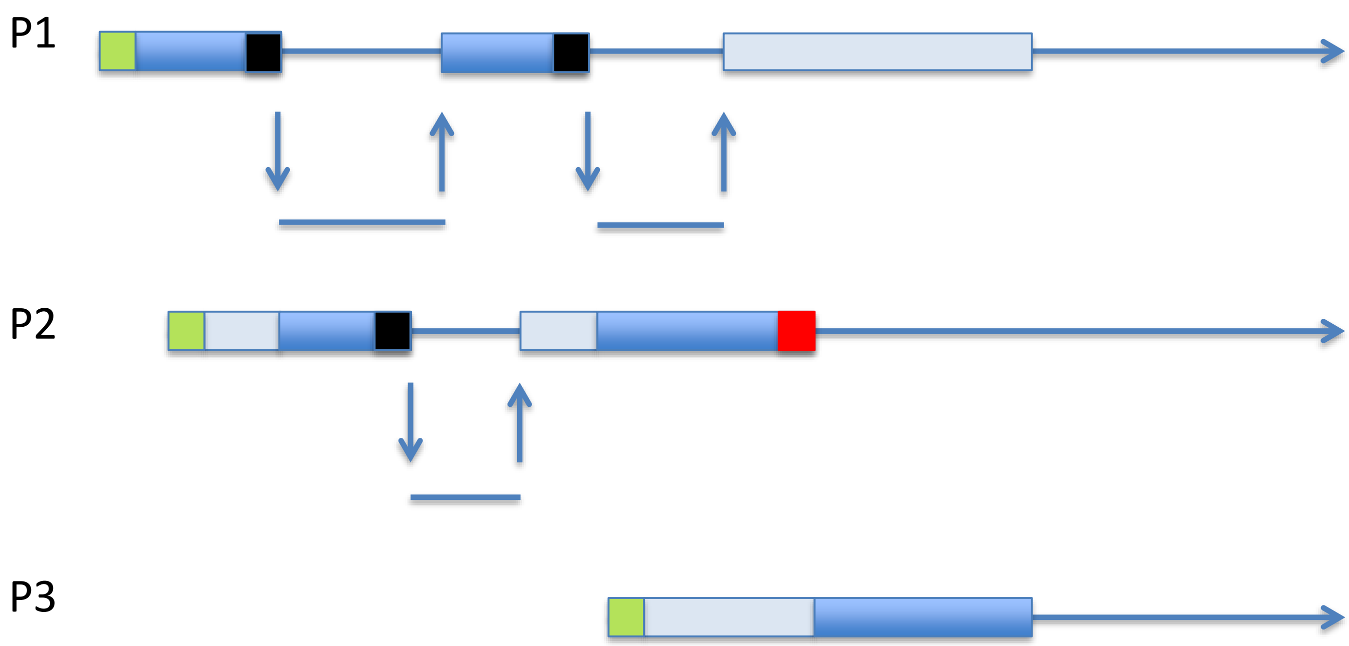
Multiprocess System



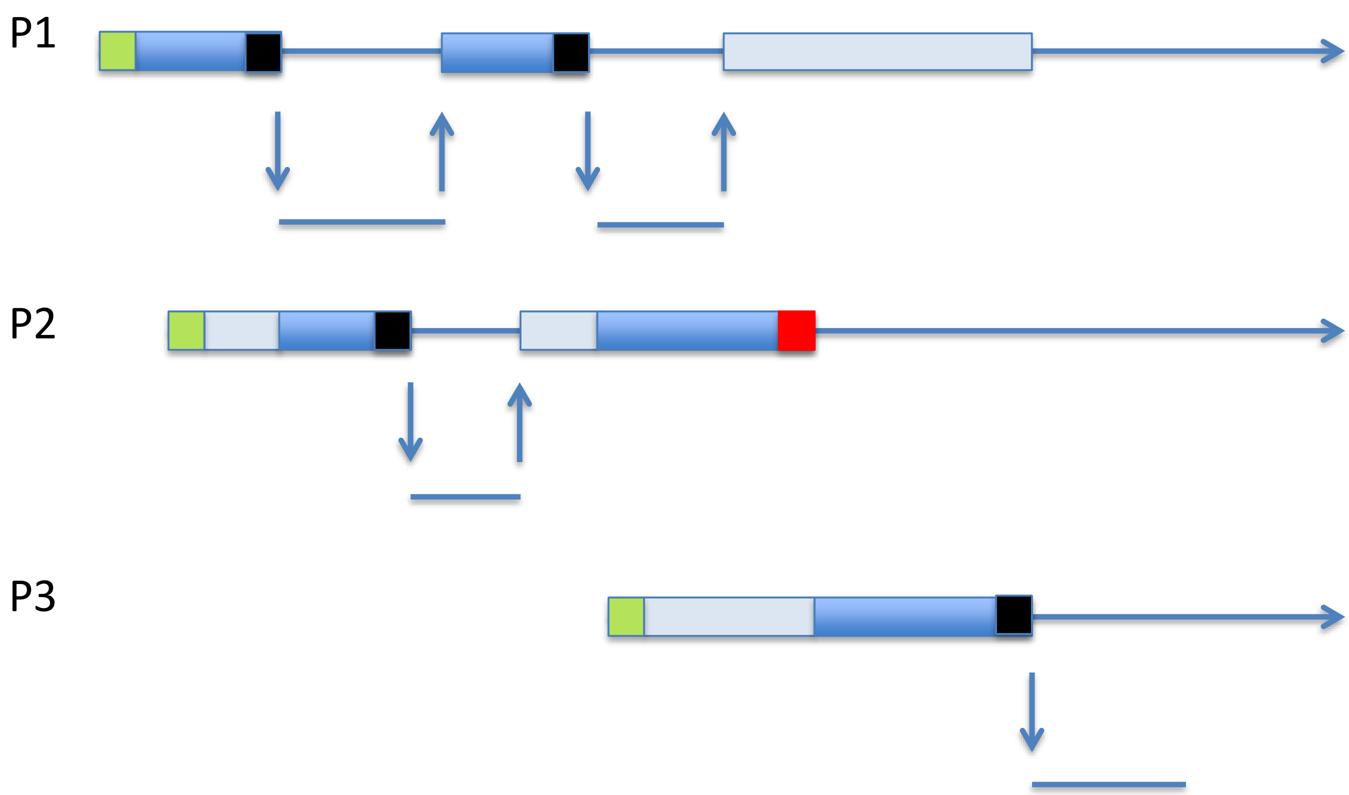
Multiprocess System



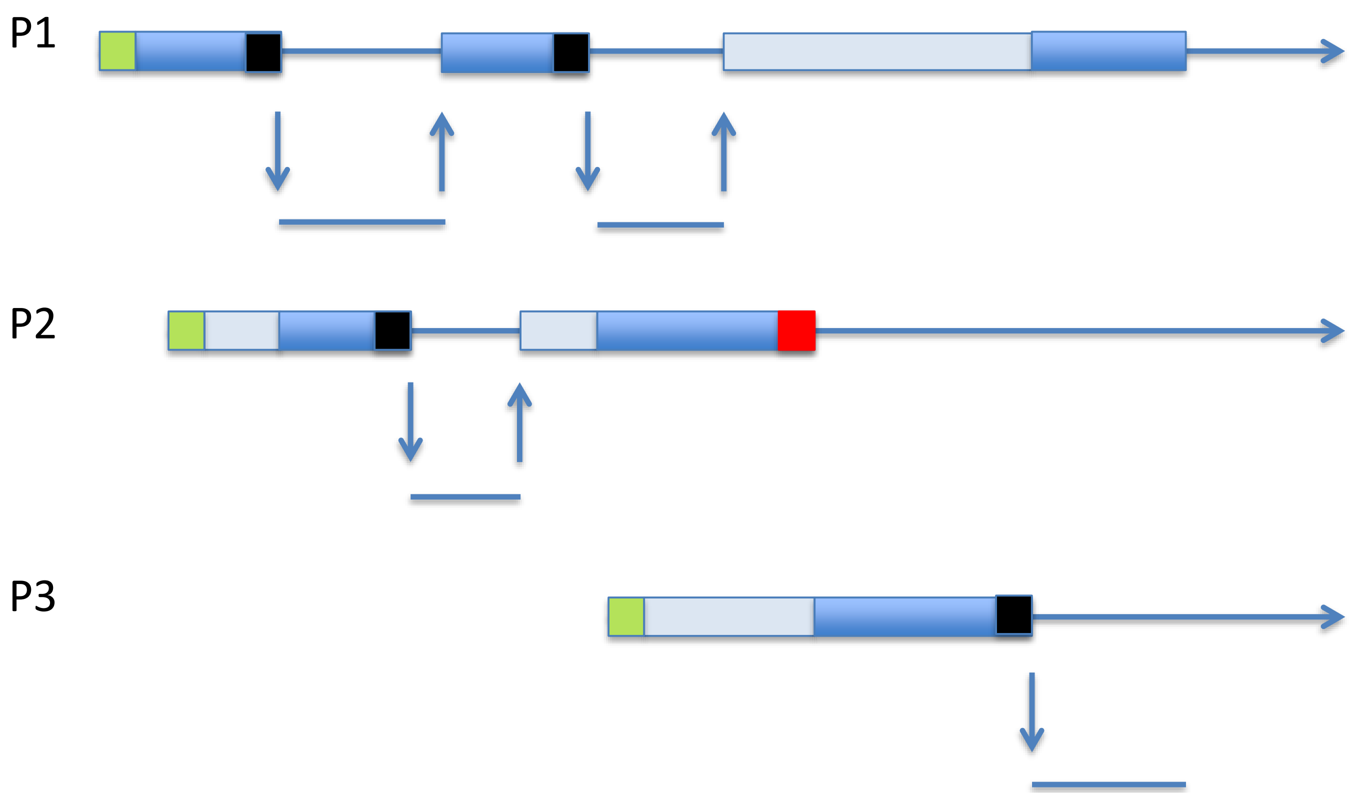
Multiprocess System



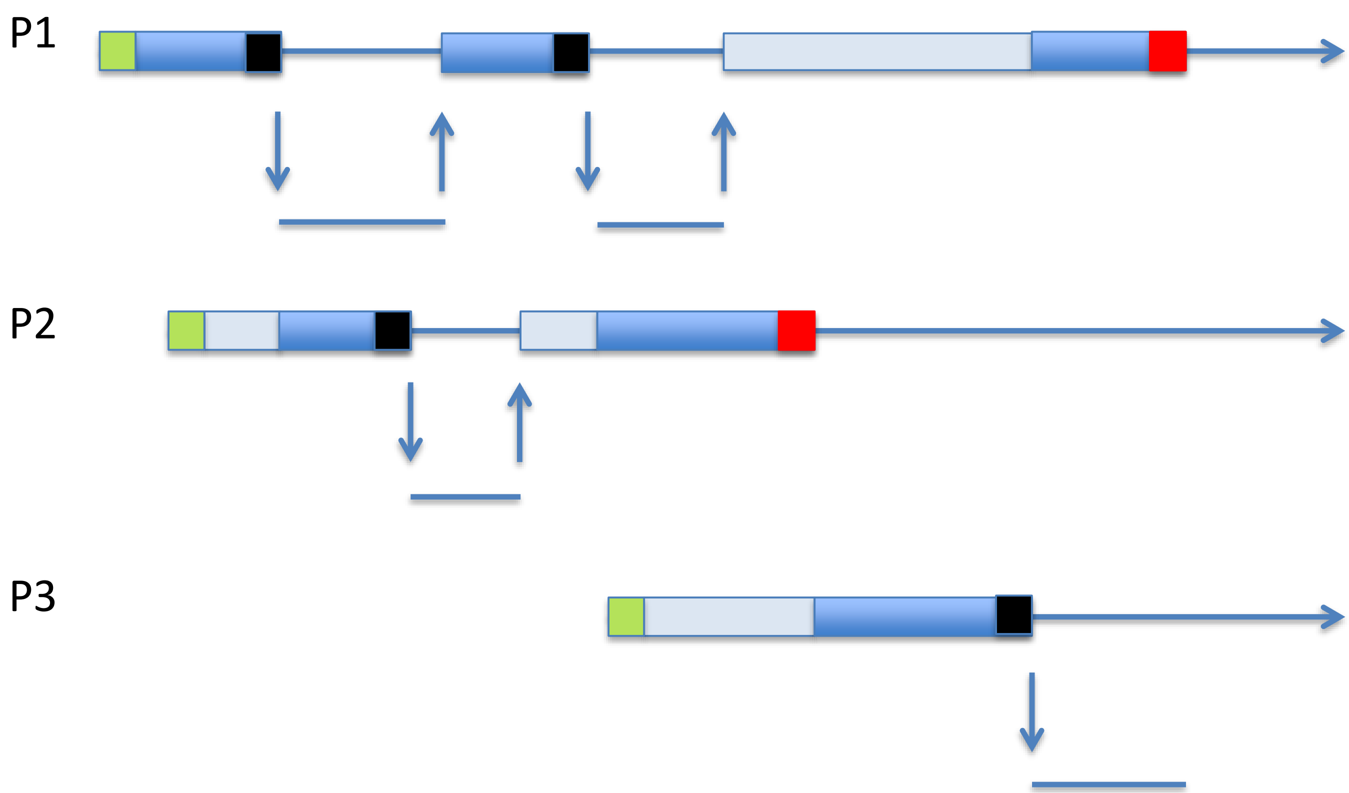
Multiprocess System



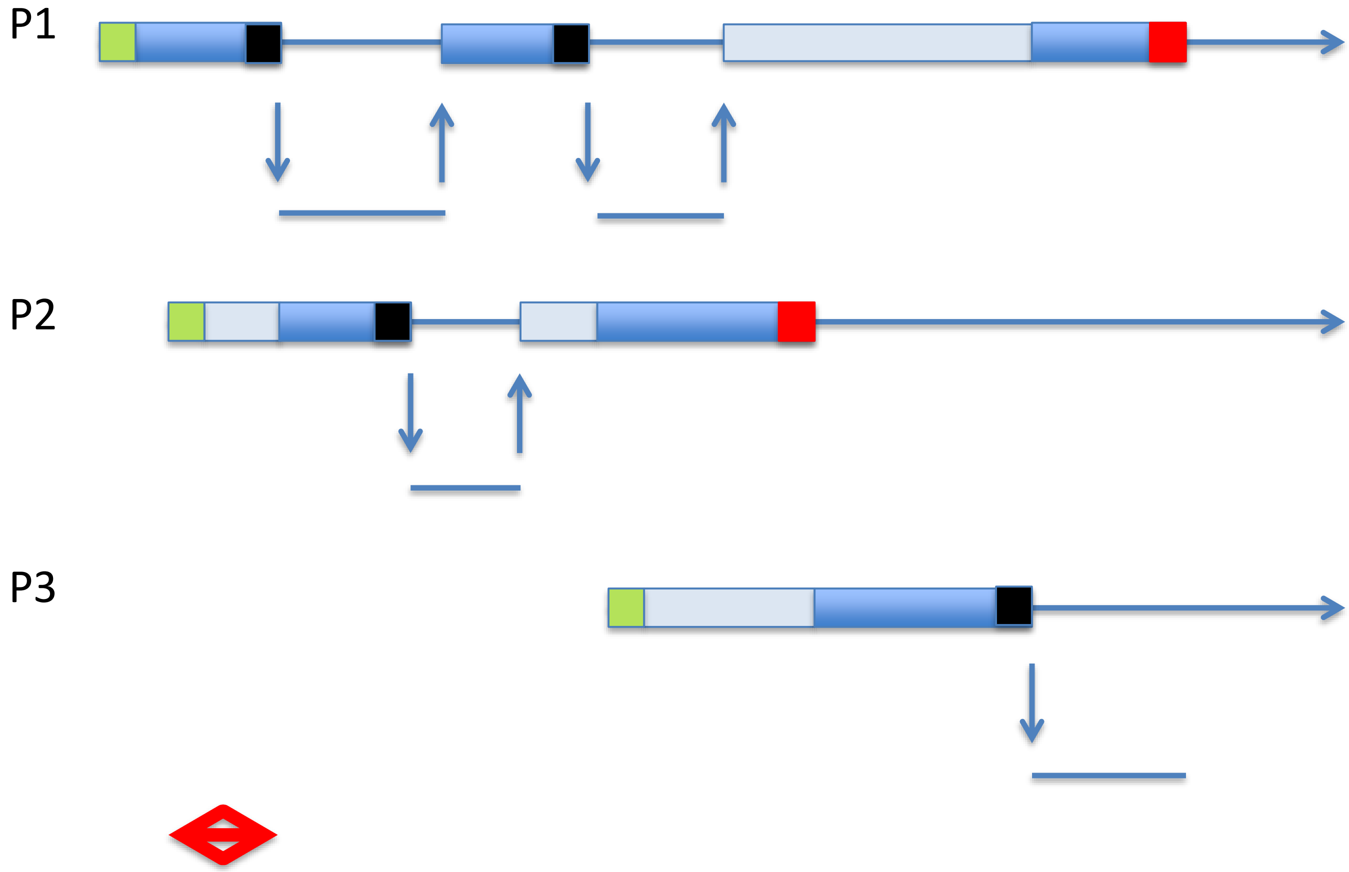
Multiprocess System



Multiprocess System

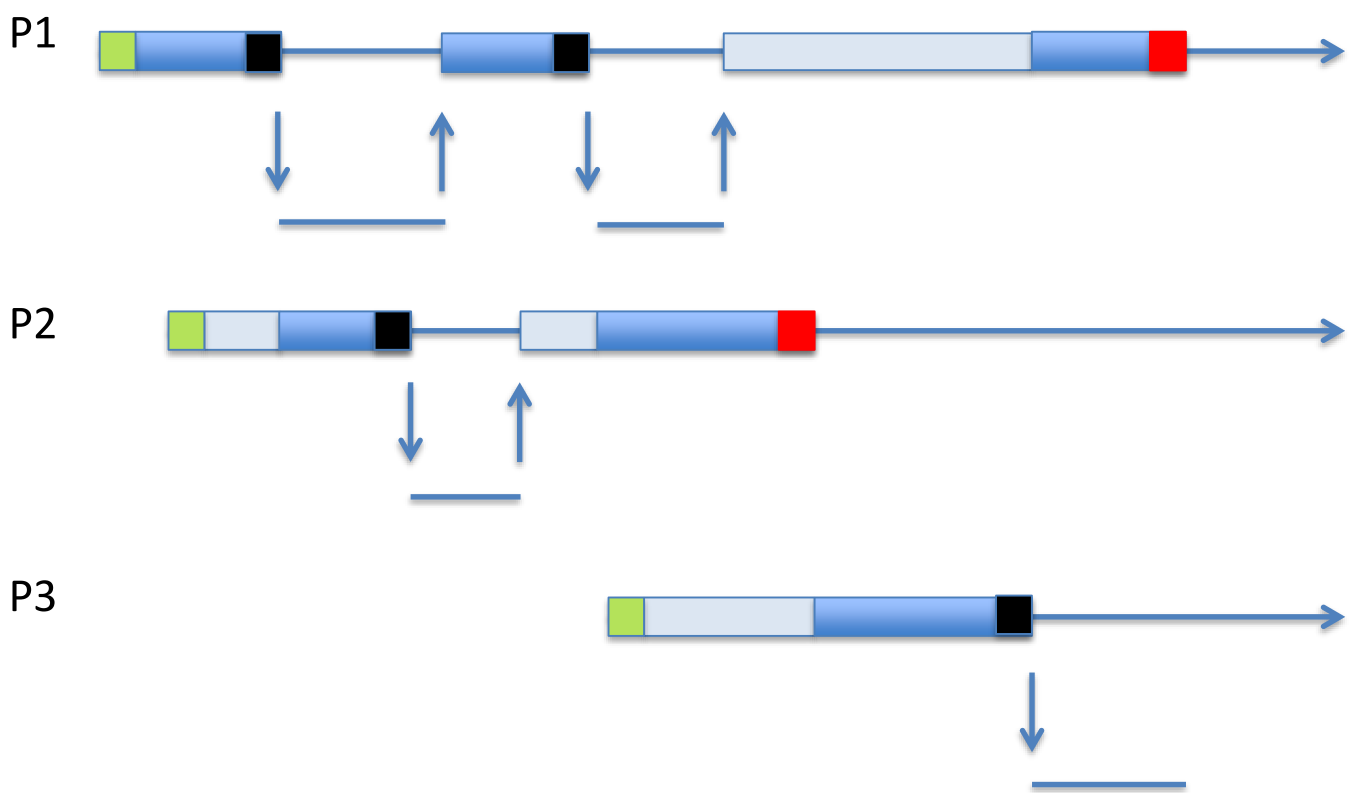


Multiprocess System



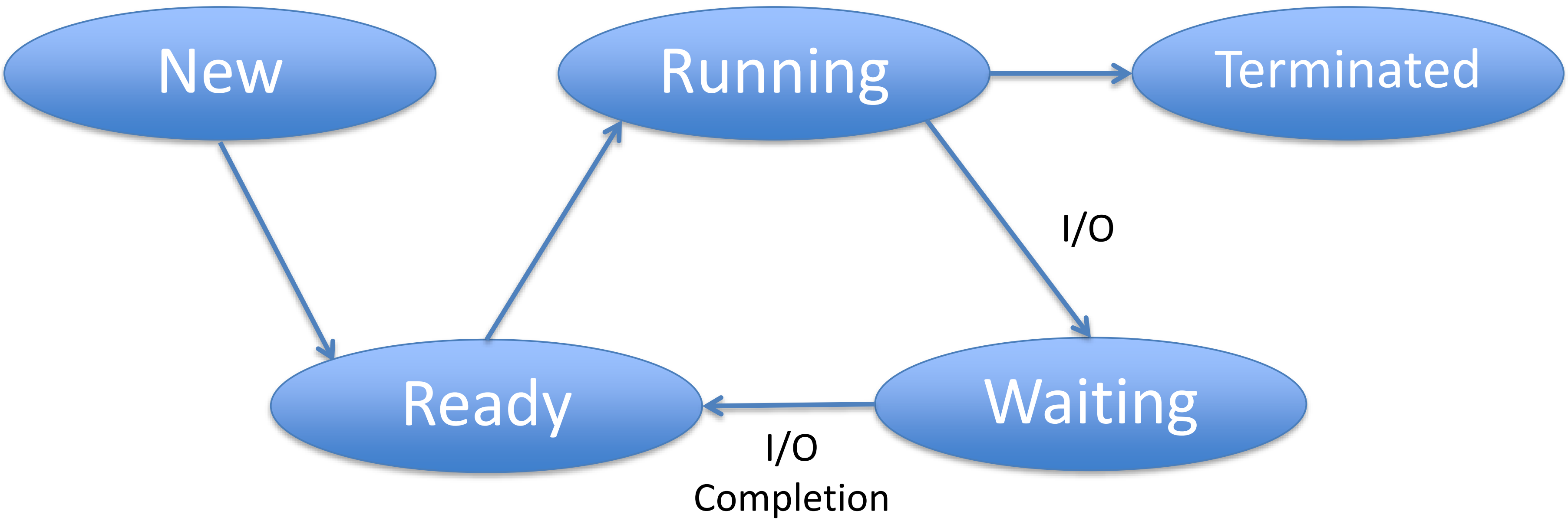
short wait time

Multiprocess System



high utilization (short CPU idle times)

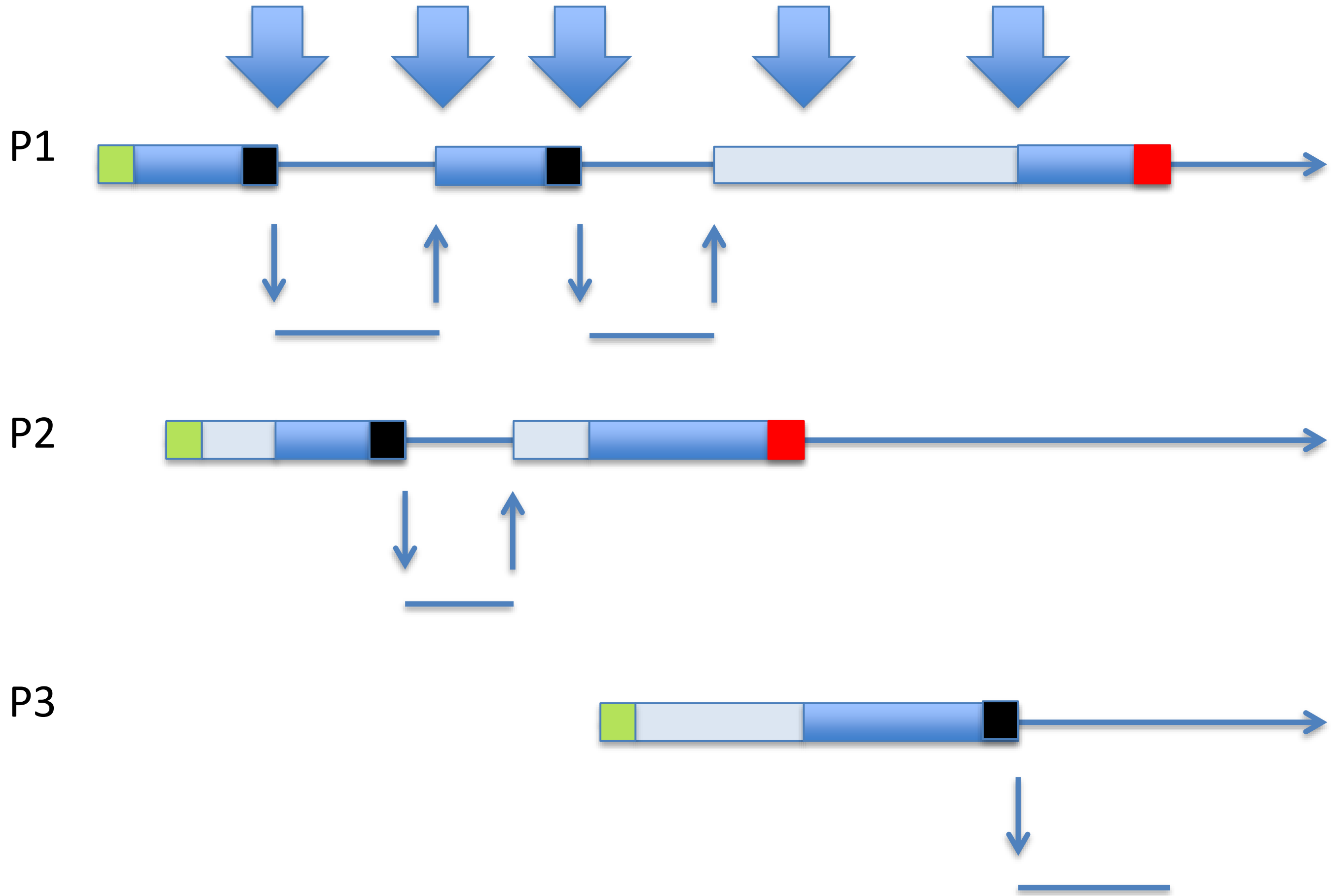
Process State Diagram for Multiprocessing System



Two Important Concepts

- Process switch
- Process scheduling

Process Switch



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 Implementation

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

Process

Process Switch Implementation

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

Code, Stack, Heap: Resides in process-private locations

Registers, MMU info: Resides in shared locations

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

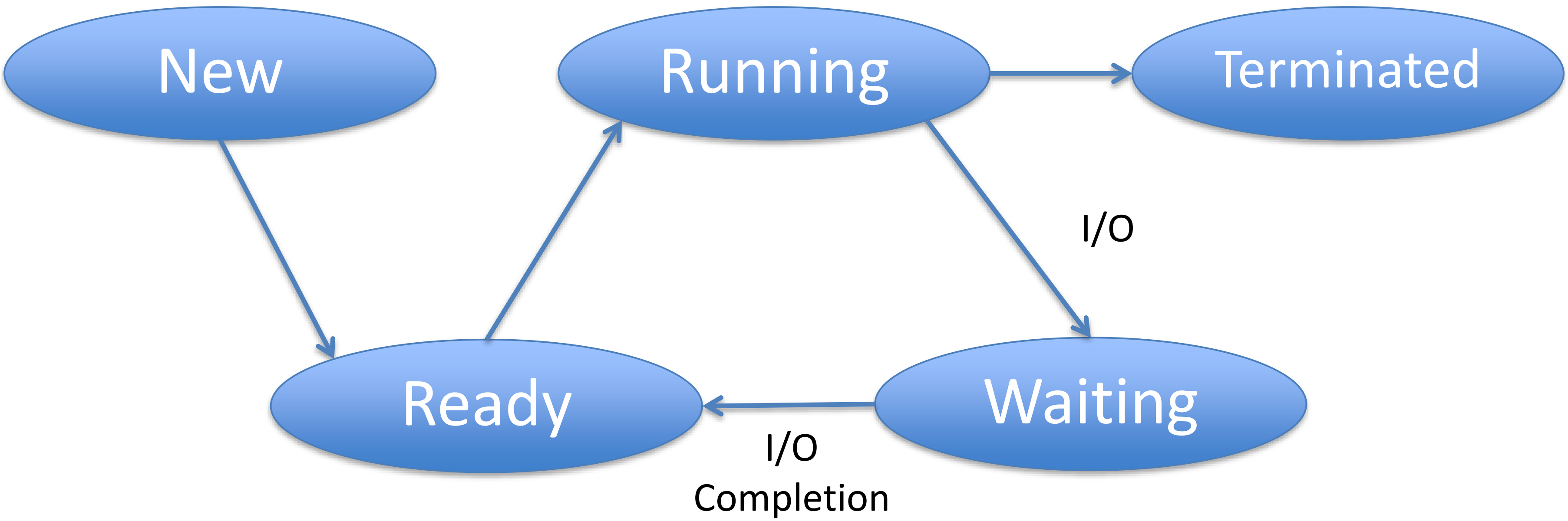
Process Switch - Caveat

- 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

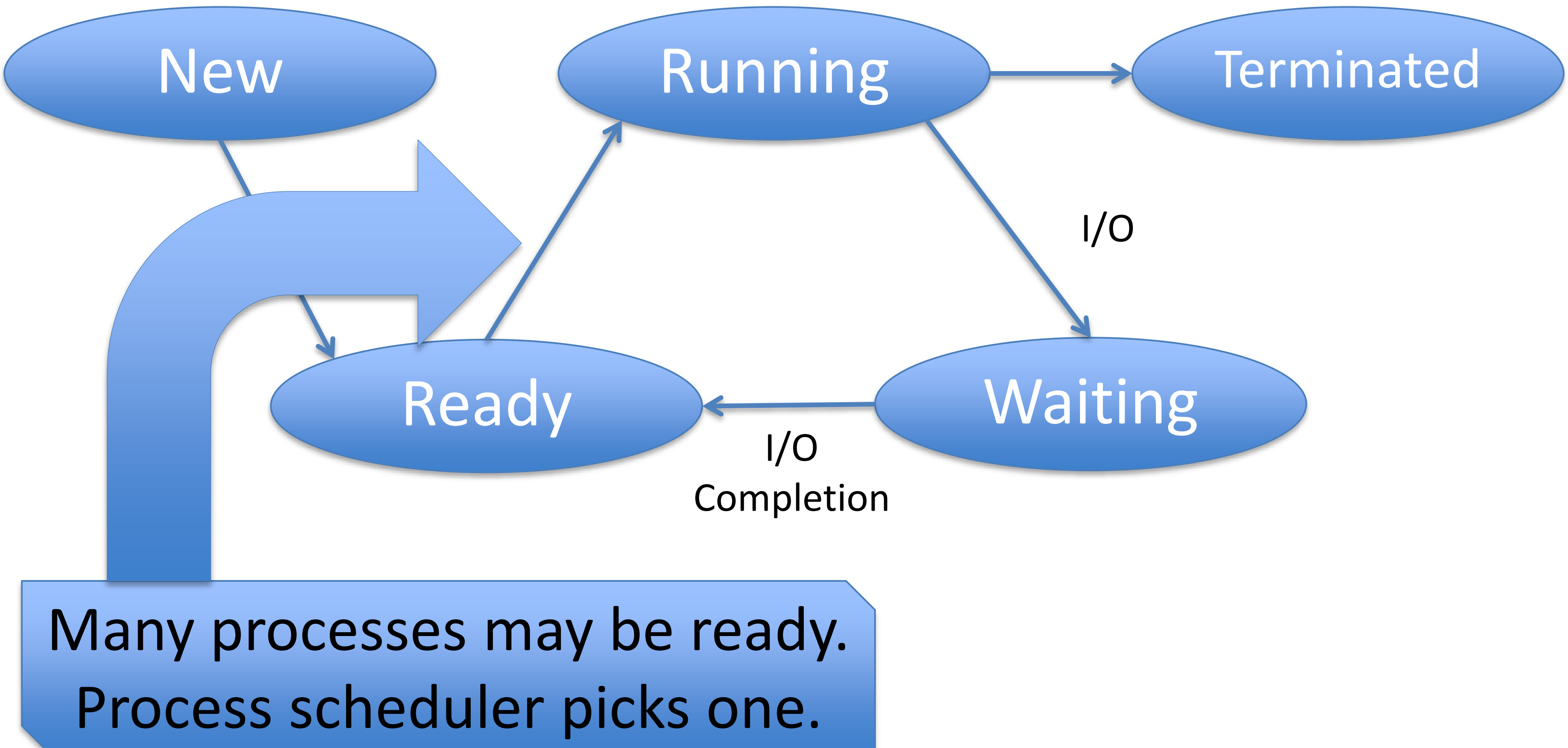
Two Important Concepts

- Process switch
- *Process scheduling*

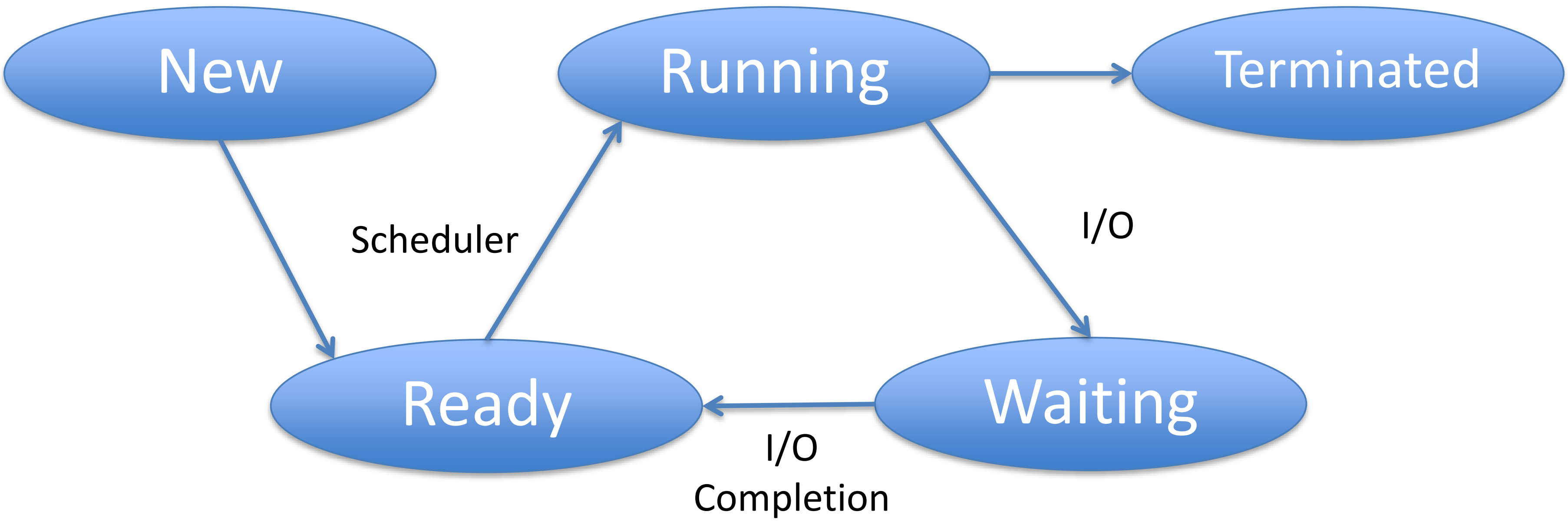
Process Scheduling



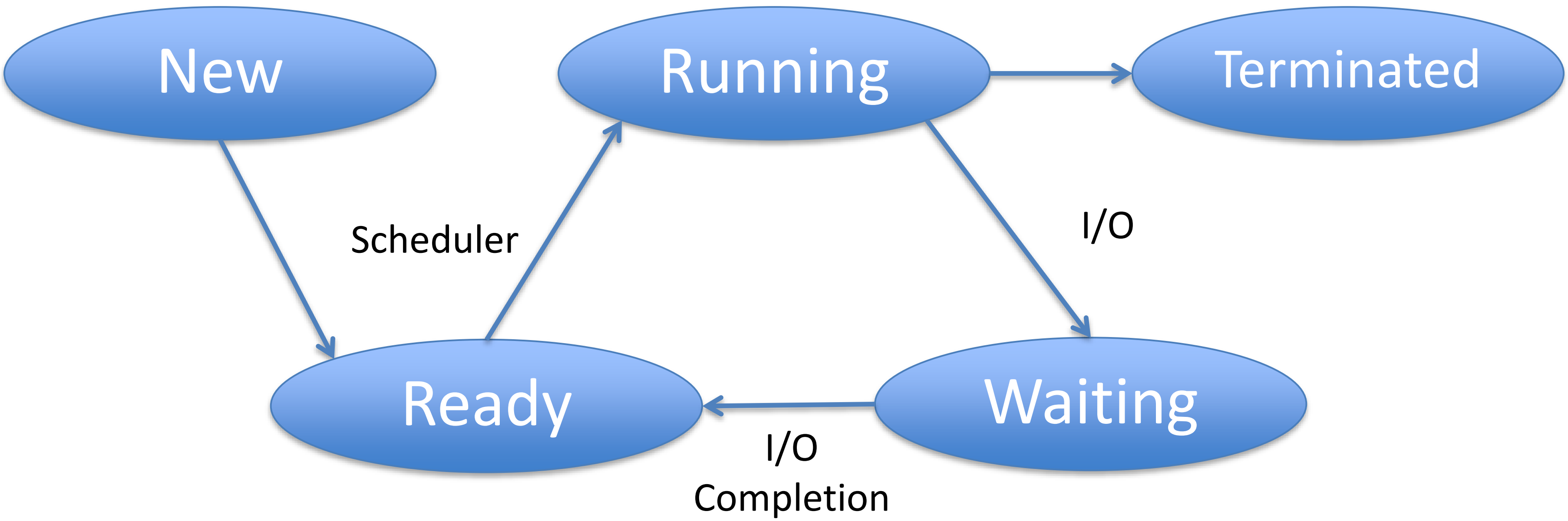
Process Scheduling



Process Scheduling

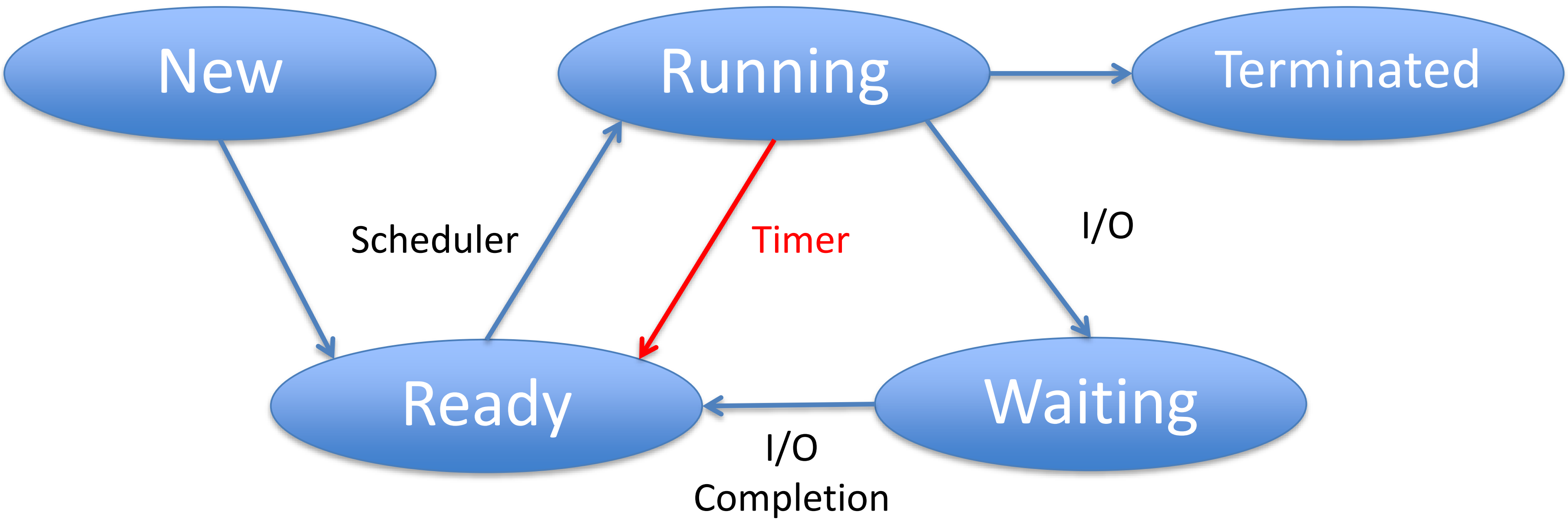


Problem



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
  - 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

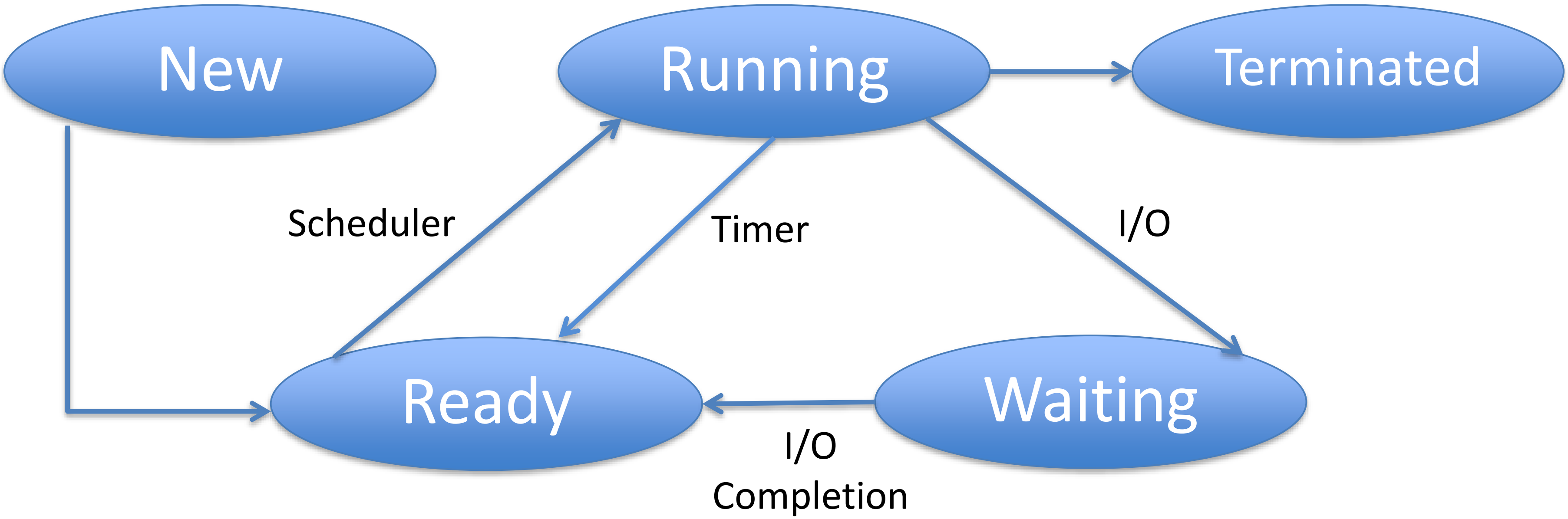
Process Scheduling Implementation

- 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?

- It depends …

Interactive vs. 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

What makes a good scheduler for interactive?

- Short response time
- Response time = wait from ready to running

What makes a good scheduler for batch?

- High throughput
- Throughput = number of jobs completed

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

Trouble is …

- Often, scheduler does not know a priori if a process is interactive or batch

What makes a good scheduler?

- It depends …
- Possibilities:
  - Fast response time for interactive processes
  - High throughput for batch process

What makes a good scheduler?

- It depends …
- Possibilities:
  - Fast response time for interactive processes
  - High throughput for compute-bound process
  - “Important” jobs get done quickly
  - “Fairness”
  - …

Example Scheduling Algorithms

- First come, first served (FCFS)
- Shortest job first (SJF)
- Round robin (RR)
- Priority (PR)
- Combination schedulers

A Note about Terminology

- 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!!

First come, first served (FCFS)

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

First come, first served (FCFS)

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

- 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

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

- 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

RR: 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

Round Robin (RR)

- 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

Priority (PR)

- Assign each process a priority Pr(P)
- Process ready:
  - Insert in queue according to Pr(P)
- Head of queue: highest-priority process

Priority (PR)

- Assign each process a priority Pr(P)
- Process ready:
  - Insert in queue according to Pr(P)
- Head of queue: highest-priority process

- Differentiation according to job importance
- Prone to starvation of low-priority jobs

A Variation: Priority + Aging (PR + A)

- Assign each process a priority Pr(P)
- Process ready:
  - Insert in q according to Pr(P)
- Reduce priority over time

A Variation: Priority + Aging (PR + A)

- Assign each process a priority Pr(P)
- Process ready:
  - Insert in q according to Pr(P)
- Reduce priority over time

- Lessens problem of starving low-priority jobs

Combination Approaches

- Almost all real schedulers are combinations
- Examples:
  - PR + RR
  - RR + FCFS
- Typical implementation:
  - Multiple queues

PR + RR

- As with priority
- But RR between processes with equal priority
- Typical implementation:
  - Multiple queues, one for each priority
  - Process ready:
    - Insert at tail of queue with its priority
  - Schedule:
    - Head of non-empty queue with highest priority for Δ

RR + FCFS

- Two queues: one for RR, one for FCFS
- Initially, process goes in RR queue
- After n Δ's, it goes in FCFS queue

Scheduler Implementation

- Must be very efficient
- Runs (at least) every Δ
- If Δ = 100 msec, scheduler run takes 10 msec

  10% of your machine is gone!
- Be careful with large number of processes

Summary

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

Summary

- Process
  - Program in execution
- Linux process tree
- Process switch
- Process scheduler

Summary

- Process
- Linux process tree
  - Created by fork() / exec() / wait() / exit()
- Process switch
- Process scheduler

Summary

- Process
- Linux process tree
- Process switch
  - Change of process using the CPU
  - Save and restore registers and other info
- Process scheduler

Summary

- Process
- Linux process tree
- Process switch
- Process scheduler
  - Decides which process to run next