OPERATING SYSTEMS: PROCESSES

Scheduling

Content

- **Process creation.**
- Process termination.
- Process lifecycle.
- Kinds of scheduling.
- Scheduling algorithms.

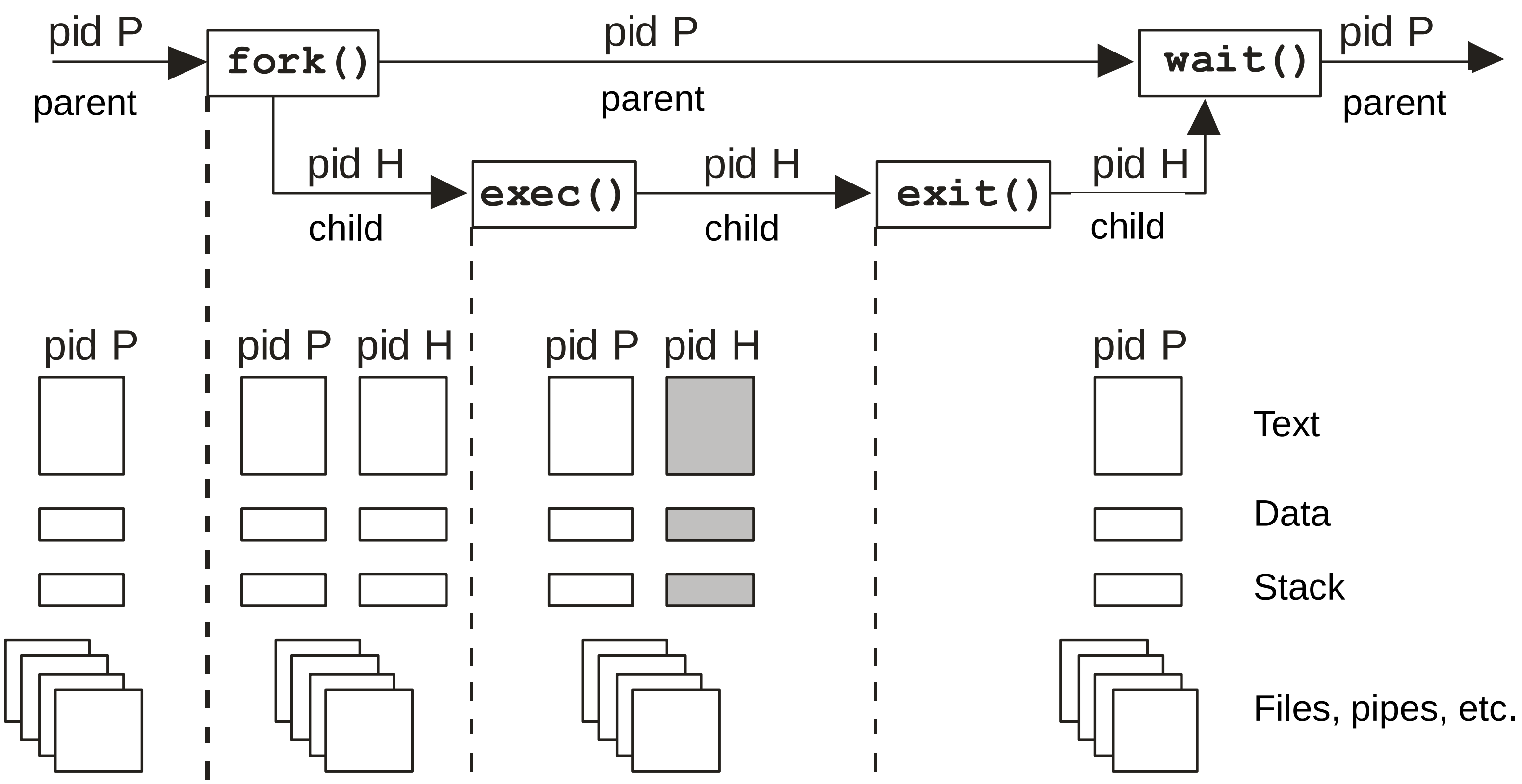
Process creation

- OS provides mechanism to allow a process to create other processes → **System call**.

- Process creation can be repeated recursively leading to a "*family structure*" → Process tree.

- Resource allocation for new process:
  - Directly obtained from the OS.
  - Parent must share its resources with child process
    - To avoid a process blocks the system by indefinitely replicating.

Use of fork, exec, wait y exit

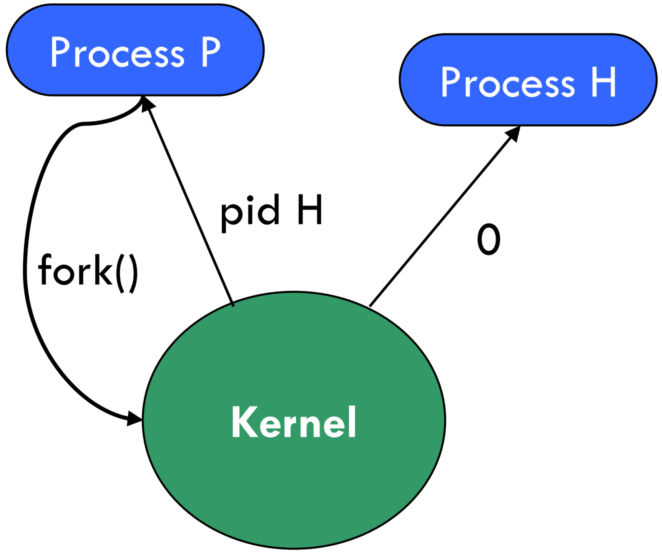


Process creation

- When a process is created:
  - In terms of **execution**:
    - Parent runs in parallel with children.
    - Parent waits until some or all of its children have terminated.
  - In terms of **memory space**:
    - Child process is a clone of parent process.
    - Child process has already another program loaded in memory.

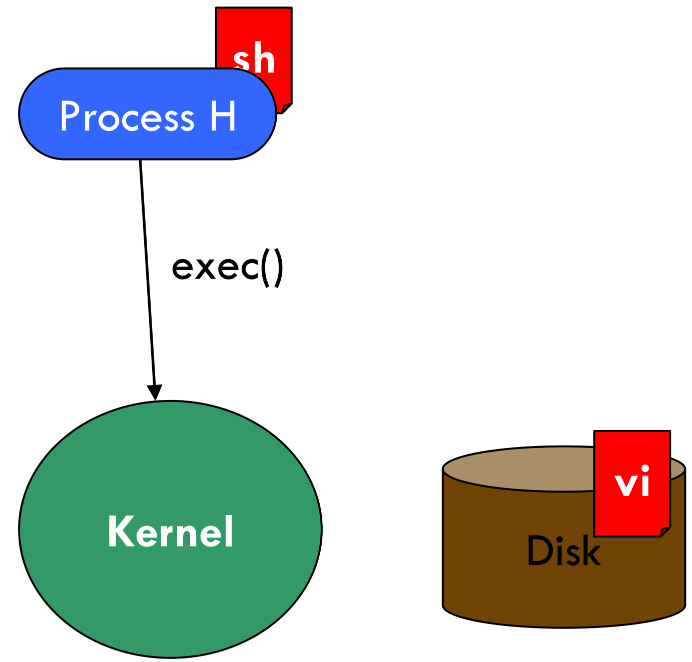
UNIX process creation

- Unix family makes distinction between process creation and new program execution.
- System call to create a new process is ***fork()***
- This system call creates a copy almost identical of the parent process.
  - Both processes, parent and child, continue execution in parallel.
  - Parent gets as a result from the *fork()* call the child PID and child gets a 0.
  - Some resources are not inherited (e.g.: pending signals).

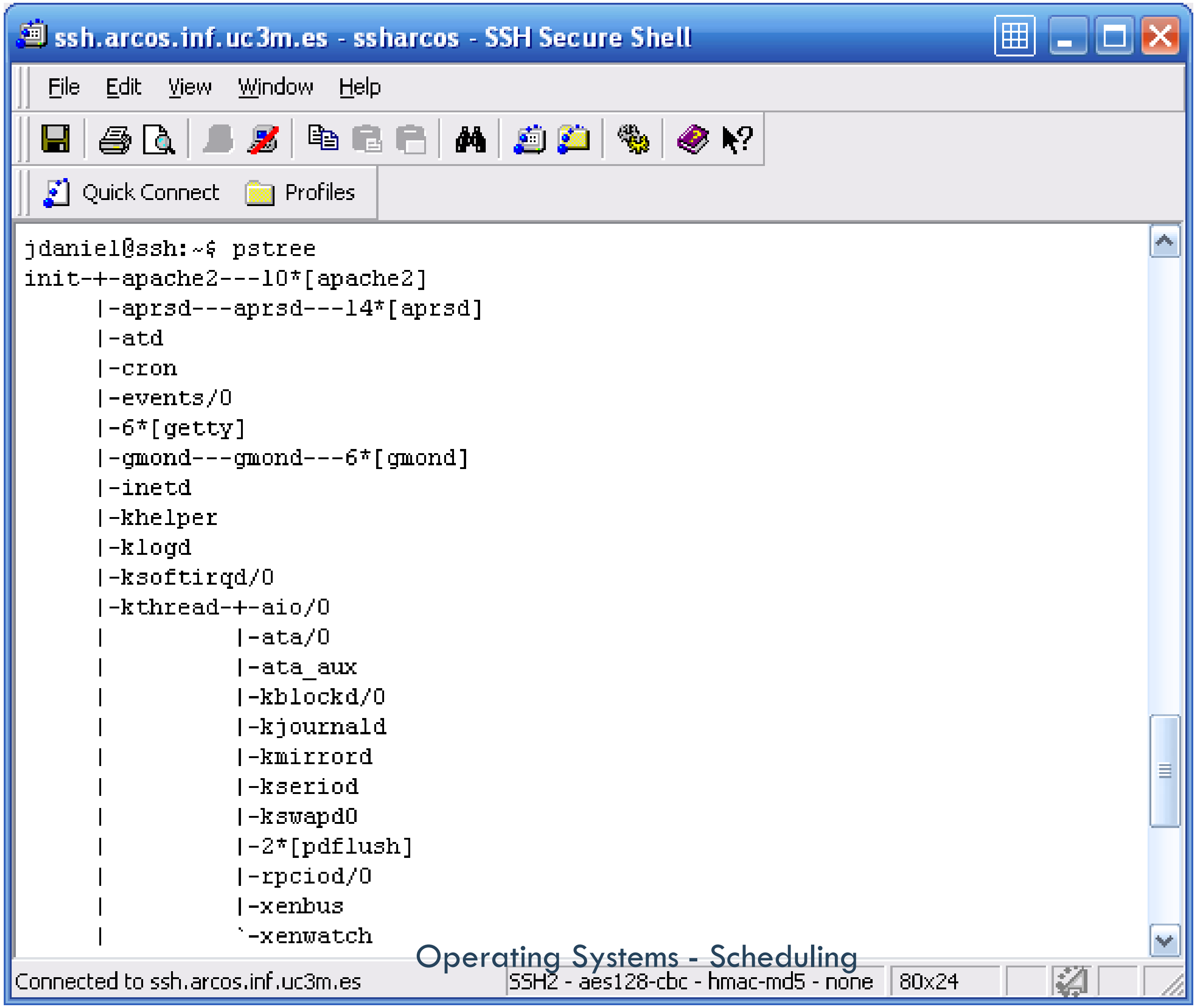


UNIX process creation

- Child process may invoke the **exec*()** system call.
  - Changes its memory image with a different program.
- Parent may go on creating more children, or waiting until the created child finishes.
  - **wait()** takes the process from the *"Ready"* queue until the child has terminated.



Process hierarchy (pstree)

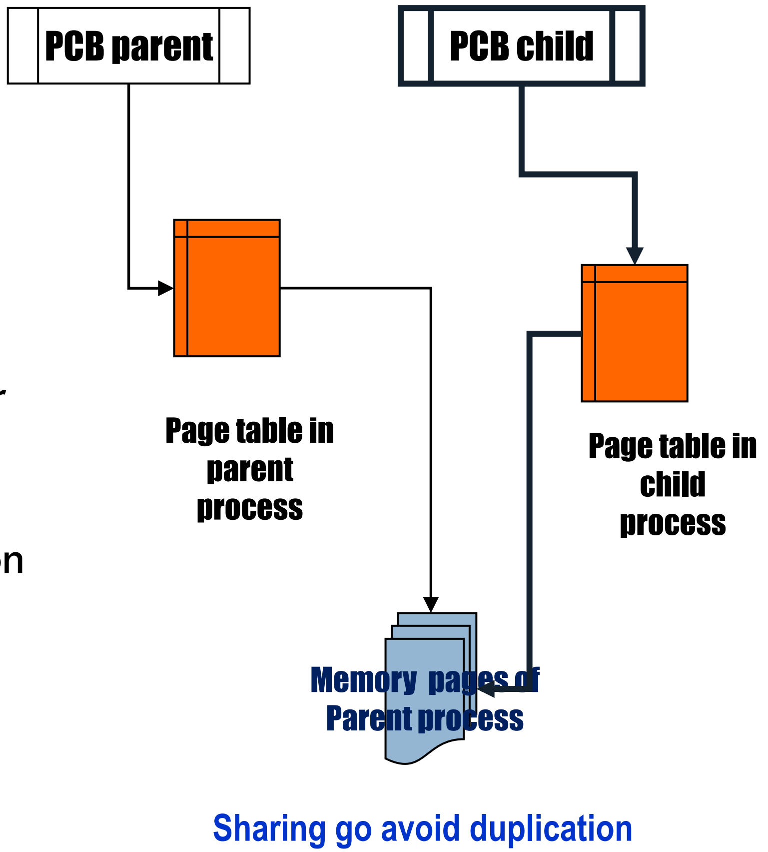



```
jdaniel@ssh:~$ pstree
init-+-apache2---10*[apache2]
     |-aprsd---aprsd---14*[aprsd]
     |-atd
     |-cron
     |-events/0
     |-6*[getty]
     |-gmond---gmond---6*[gmond]
     |-inetd
     |-khelper
     |-klogd
     |-ksoftirqd/0
     |-kthread-+-aio/0
     |         |-ata/0
     |         |-ata_aux
     |         |-kblockd/0
     |         |-kjournald
     |         |-kmirrord
     |         |-kseriod
     |         |-kswapd0
     |         |-2*[pdflush]
     |         |-rpciod/0
     |         |-xenbus
     |         `-xenwatch
```

Copy on Write (COW)

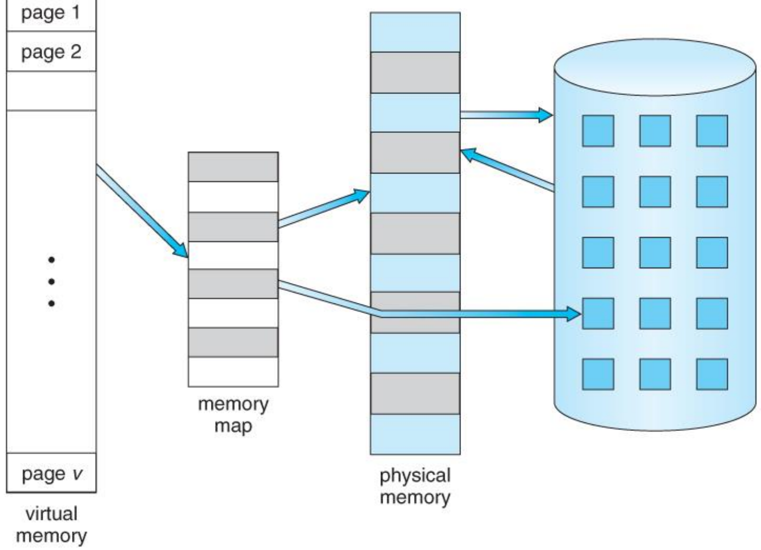
- Inefficiencies of the **fork()** model:
  - A lot of data is copied, but could be shared instead.
  - If another memory image is loaded, it is even worse as all copies are discared.

- Many UNIX systems use COW:
  - *Copy-on-Write* is a technique to delay or avoid copying data when performing a fork.
  - Data are marked so that if a modification is tried, then a copy is performed for each process (parent and child).
  - Now fork() only copies the page table from parent (but not the pages) and creates a new PCB for child.

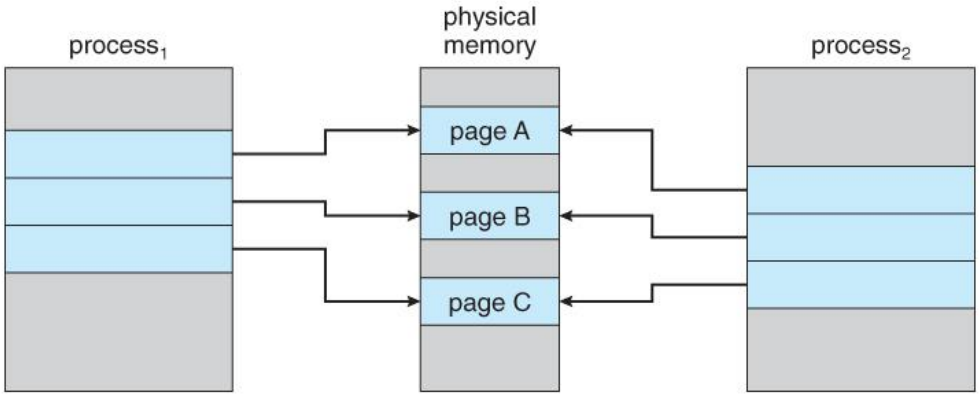


Copy on Write (COW)

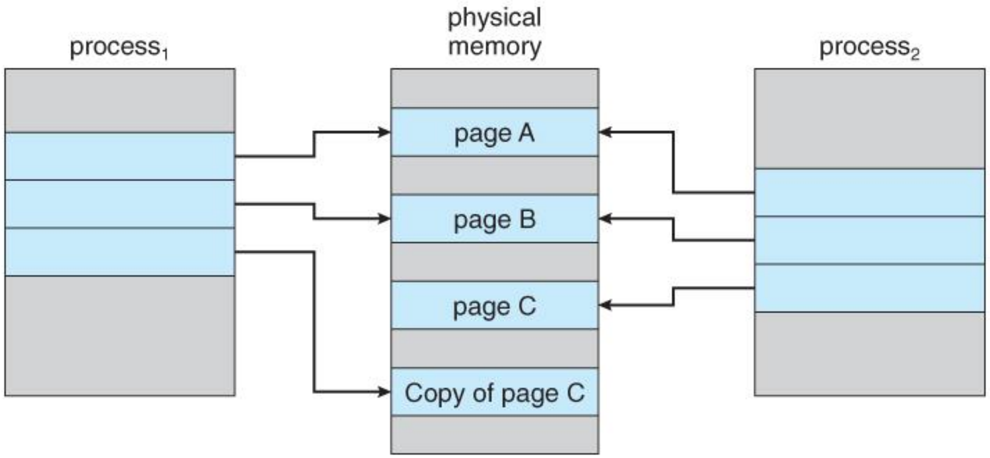
Virtual memory overview



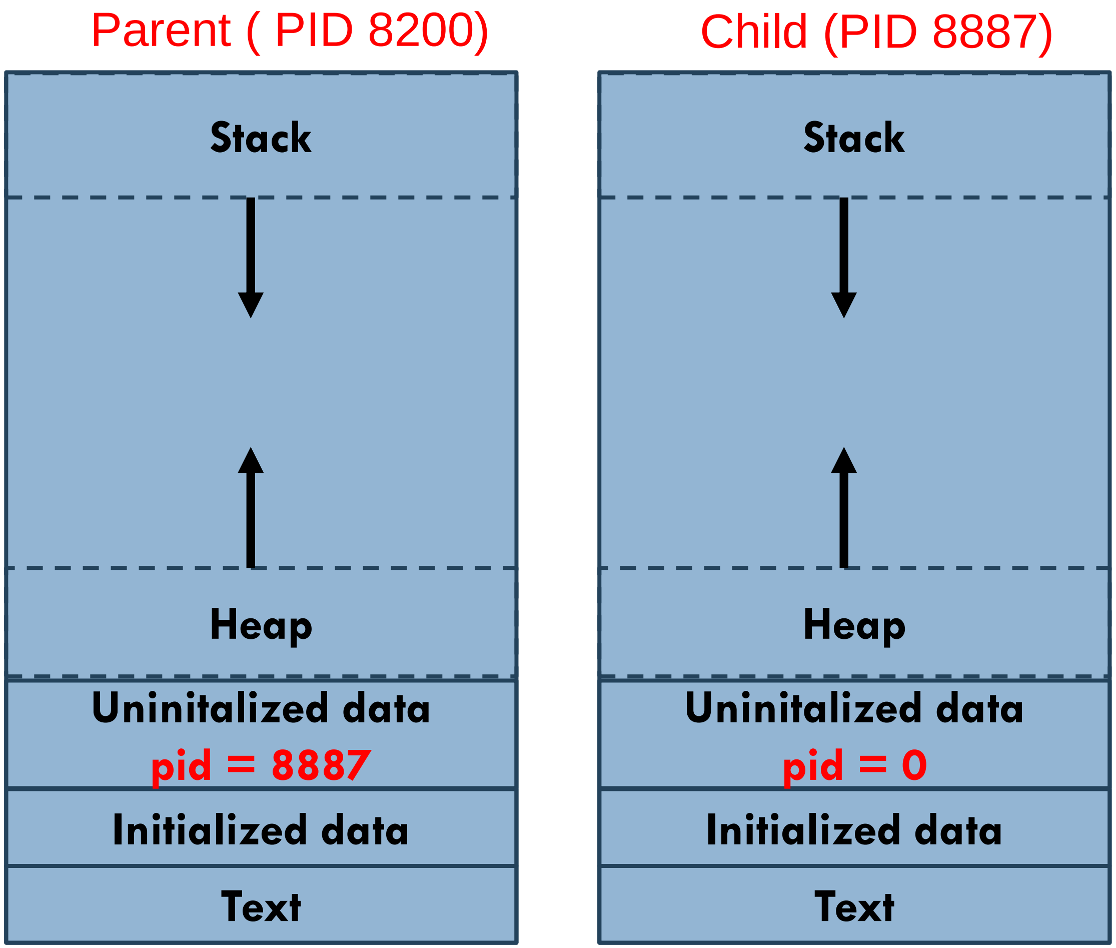
P1 executes fork()



P1 writes on page c



Fork service




```
pid_t pid;
pid = fork();
switch (pid) {
  case -1: /* error */
    exit(-1);
  case 0: /* child process */
        printf("Child process");
      break;
  default:
    printf("Parent process");
}
```

Linux process creation

`fork:`

"Copies parent process and gives new identity to child"

Get free entry in Process Table

↓

Copy PCB from parent

↓

Duplicate Memory Map from parent (including stacks)

↓

State ← *ready*

PCB in ready queue

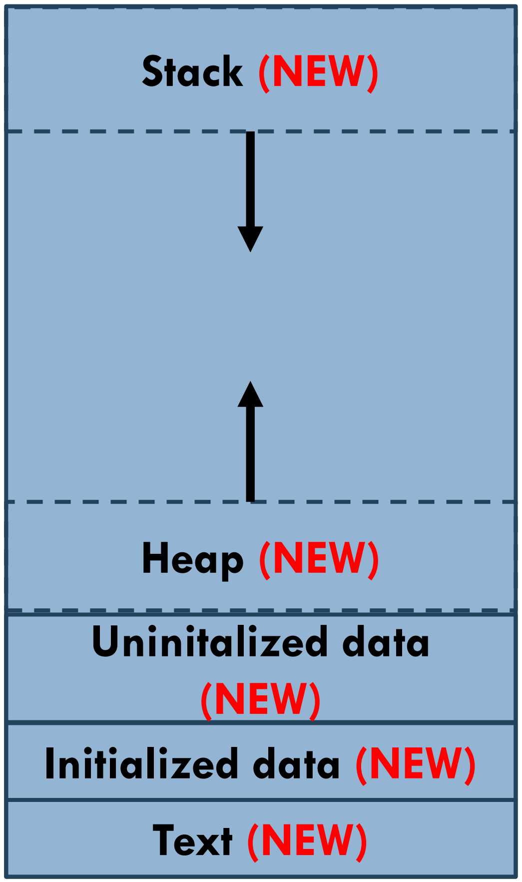
Also clean up signals, events,…

↓

Return **PID** to parent

Return **0** to child

Exec service

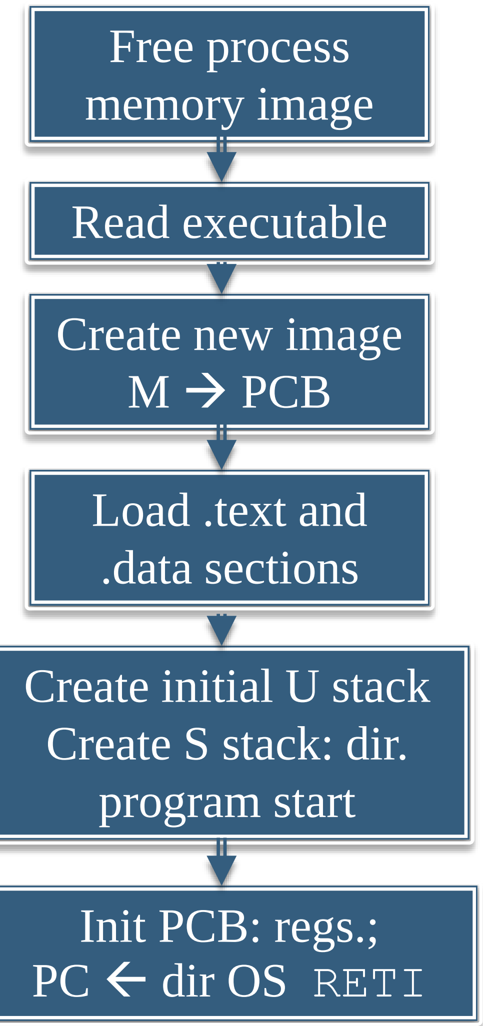



```
if (execvp(argv[1], &argv[1])<0) {
      perror("error"); }
```

Process creation in Linux

`exec:`

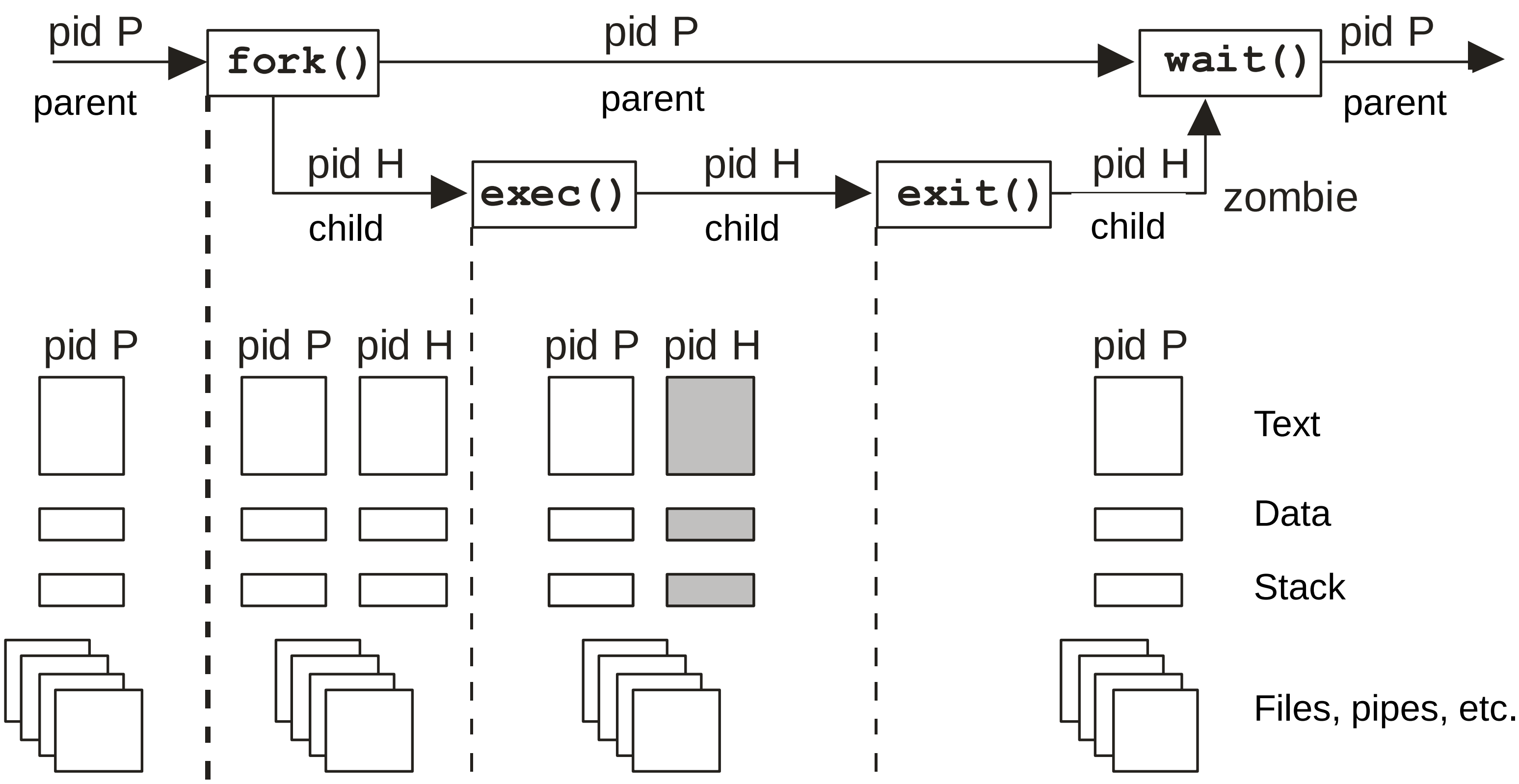
"Change memory image from a process using as a new container a new one"



Content

- Process creation.
- **Process termination.**
- Process lifecycle.
- Kinds of scheduling.
- Scheduling algorithms.

Use of fork, exec, wait y exit



Process termination

- When a process finishes all its allocated resources are freed up.
  - Memory, open files, entries in tables, …
- and kernel notifies about events to parent process.

- A process may terminate in 2 ways:
  - Voluntarily: **exit()** system call.
  - Involuntarily:
    - Exceptions: divide by zero, segmentation fault, …
    - Aborted by user (ctrl-c) or other process (kill)
      - i.e.: signals that cannot be handled or ignored.

Process termination

- When a process terminates two outcomes are possible:
  - Its children are not affected.
  - All children also terminate → **cascade termination (e.g. VMS)**

- In Unix,
  - When the parent is terminated, its children now depending from **init** process.
  - When the child is terminated, the process changes to ***zombie*** state until parent process gets its termination code.

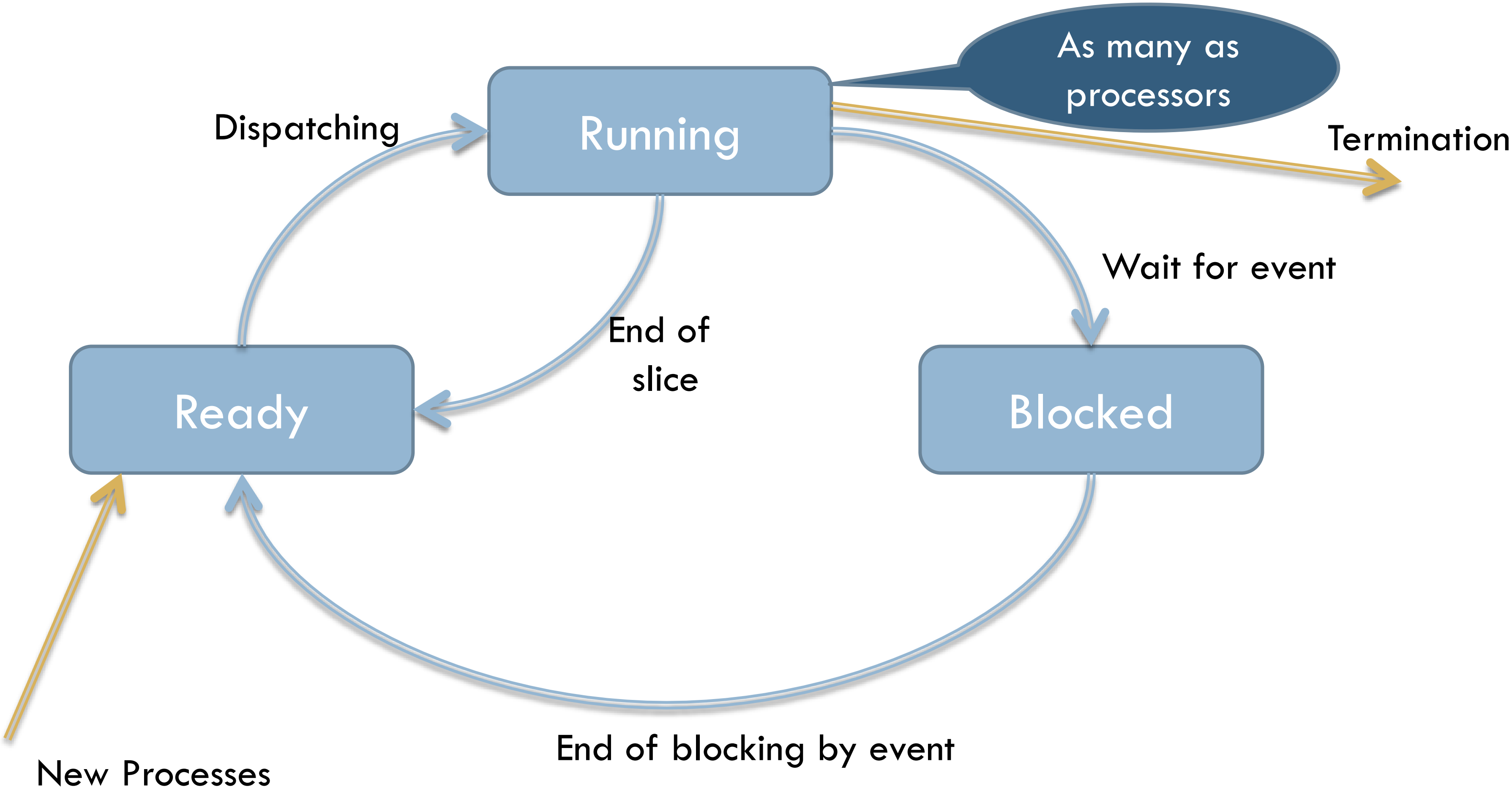
When is PCB eliminated?

- Process termination and PCB elimination are two different tasks:
  - When parent gets information from child, data structures can be removed.
  - **wait()** system call:
    - Blocks process until a child terminates.
    - Returns **PID** of the terminated child.

Content

- Process creation.
- Process termination.
- **Process lifecycle.**
- Kinds of scheduling.
- Scheduling algorithms.

Process basic lifecycle

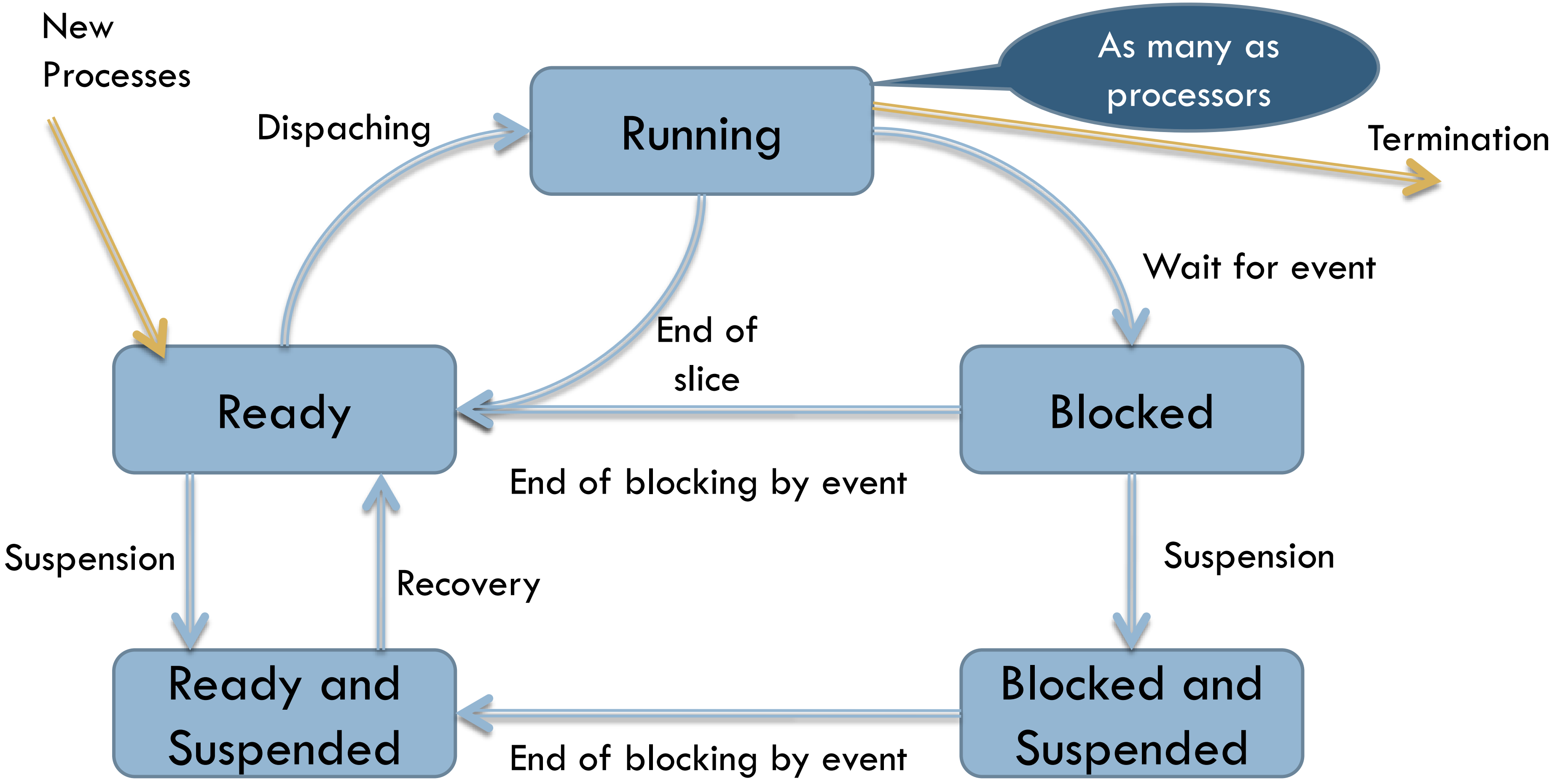


Spawning to disk (swap)

- When there are many processes in execution, performance may degrade due to excessive paging (trashing).
  - Solution: suspended state where Operating System sends a process to the swap area in disk.

- Introduce new process states.
  - Blocked and suspended.
  - Ready and suspended.

Process lifecycle



Content

- Process creation.
- Process termination.
- Process lifecycle.
- **Kinds of scheduling.**
- Scheduling algorithms.

Scheduling levels

- Short-term scheduling:
  - Selects next process to execute.
- Medium-term scheduling:
  - Select process to be suspended or recovered from swap to main memory.
- Long term scheduling:
  - Perform admission control of processes.
  - Used in batch systems.

Kinds of scheduling

- Non-preemptive.
  - Process in execution keeps using CPU all the time it needs.

- Preemptive:
  - Operating System may evict process from CPU and execute another process.

Scheduling decision points

- Points in time when OS may decide process scheduling:
  1. When a process blocks waiting for an event.
     - Perform a system call.
  2. When an interrupt happens.
     - Clock interrupt.
     - I/O interrupt.
  3. End of process.
- Non preemptive scheduling: 1 y 3.
  - Windows95, MacOS before 8.
- Preemptive scheduling: 1, 2 y 3.

Process queues

- Ready processes are kept in a queue.

- Alternatives:
  - Single queue.
  - Queues for each class of process.
  - Priority queues.

Content

- Process creation.
- Process termination.
- Process lifecycle.
- Kinds of scheduling.
- **Scheduling algorithms.**

Scheduling: metrics

- CPU utilization:
  - Percentage of time CPU is used.
  - Goal: Maximize.
- Throughput:
  - Number of jobs finished by unit of time.
  - Goal: Maximize.
- Return time (T_q)
  - Overall time a process is in system (running, blocked or waiting)
  - $T_q = T_f - T_i$
    - T_f: Finalization time.
    - T_i: Initiation time.
  - Goal: Minimize.

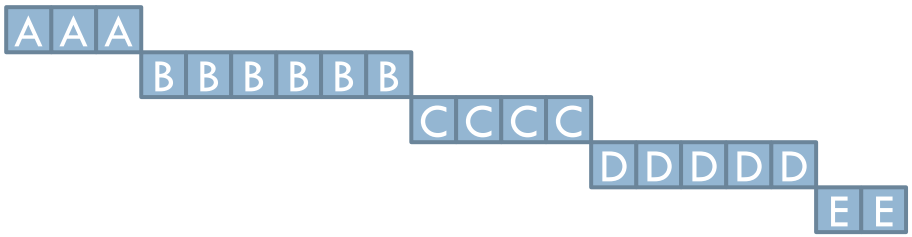
Scheduling: Metrics

- Service time(T_s):
  - Time devoted to productive tasks (cpu, I/O).
  - $T_s = T_{cpu} + T_{I/O}$
- Waiting time (T_w):
  - Time a process spends in waiting queues.
  - $T_w = T_q - T_s$
- Normalized return time (T_n):
  - Ratio between return time and service time.
  - $T_n = T_q / T_s$
  - Indication of experienced delay.

FCFS

- ***First to Come First to Serve.***
  - Non-preemptive.
  - Penalizes short processes

Process	Arrival	Service
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



FCFS: Normalized Return Time

Process	Arrival	Service	Init	End	Return	Wait	Normalized Return
A	0	3	0	3	3	0	3/3=1
B	2	6	3	9	7	1	7/6=1.16
C	4	4	9	13	9	5	9/4=1.25
D	6	5	13	18	12	7	12/5=2.4
E	8	2	18	20	12	10	12/2=6

- Average return time: **4.6**
- Average normalized return time : **2.5**

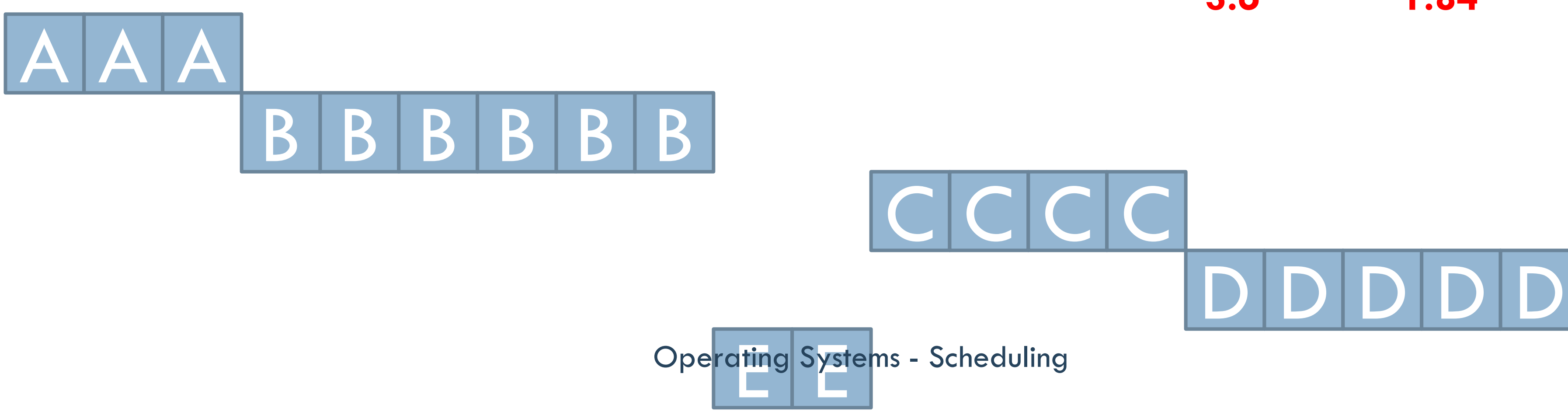
SJF

- ***Shortest Job First.***
- Non-preemptive algorithm.
- Selects shortest job.
- It only can be applied if duration of each job is known beforehand.
- Starvation possibility:
  - If short jobs are continuously arriving, longer jobs never are executed.

SJF

Process	Arrival	Service	Init	End	Return	Wait	Normalized Return
A	0	3	0	3	3	0	3/3=1
B	2	6	3	9	7	1	7/6=1.16
C	4	4	11	15	11	7	11/4=2.75
D	6	5	15	20	14	9	14/5=2.8
E	8	2	9	11	3	1	3/2=1.5

3.6 **1.84**



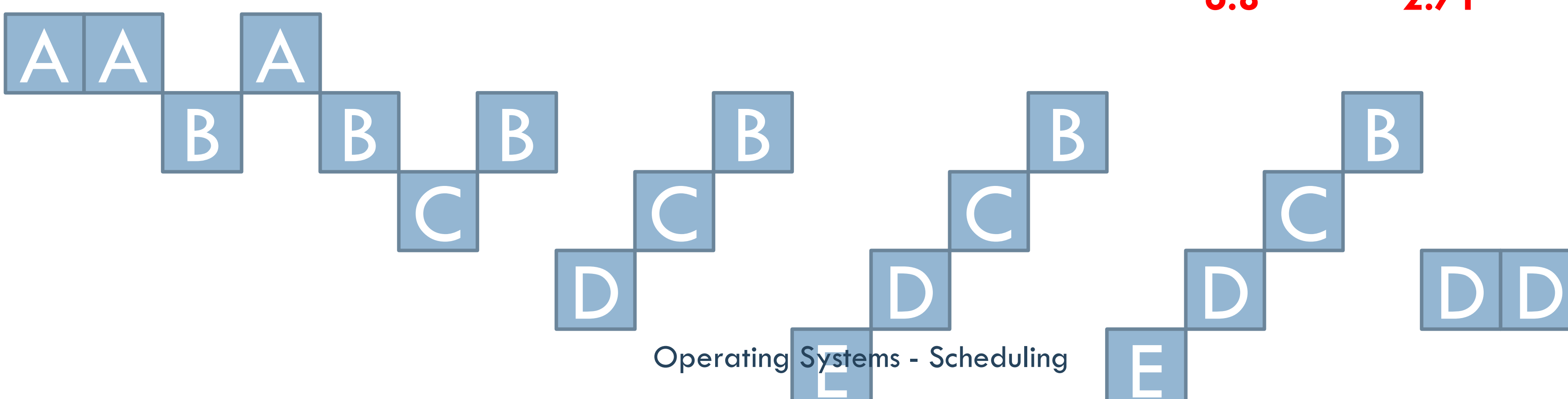
Cyclic or Round-Robin

- Keeps a FIFO queue with processes **ready** to run.
- A process is allocated into a processor for a **time slice**.
- A process goes back to the **ready** queue when:
  - Its time slice expires.
  - An event that took it to the blocked queue happens.
- A process goes to the **blocked** queue when:
  - Starts waiting for an event.
- It is a preemptive algorithm.
- It is important to remind that every context switch leads to a delay:
  - Time slice >> Context switch time

Round-Robin (q=1)

Process	Arrival	Service	Init	End	Return	Wait	Normalized Return
A	0	3	0	4	4	1	4/3=1.33
B	2	6	2	18	16	10	16/6=2.66
C	4	4	5	17	13	9	13/4=3.25
D	6	5	7	20	14	9	14/5=2.8
E	8	2	10	15	7	5	7/2=3.5

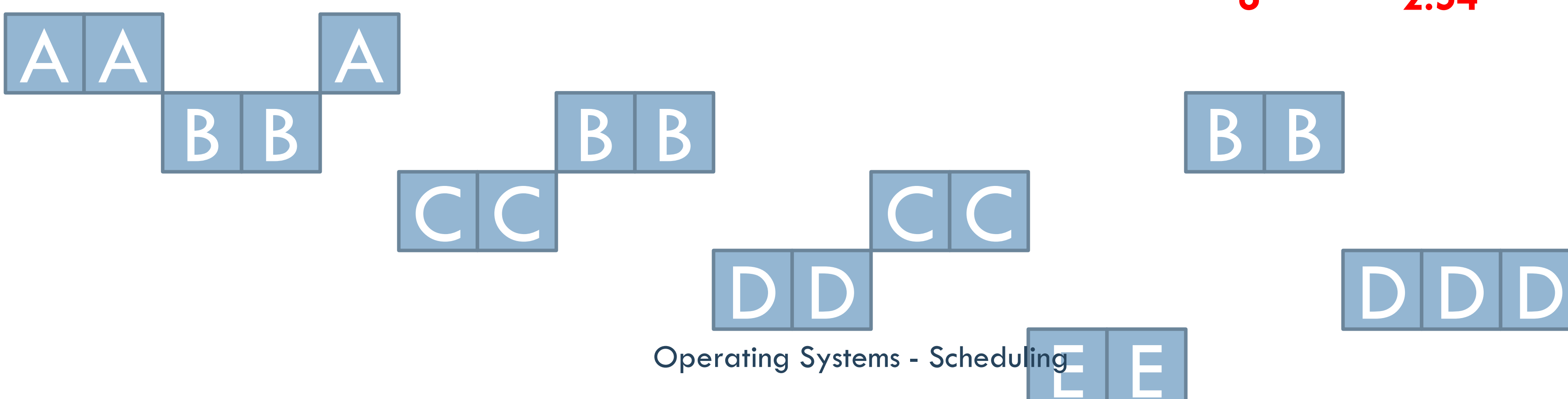
6.8 **2.71**



Round-Robin (q=2)

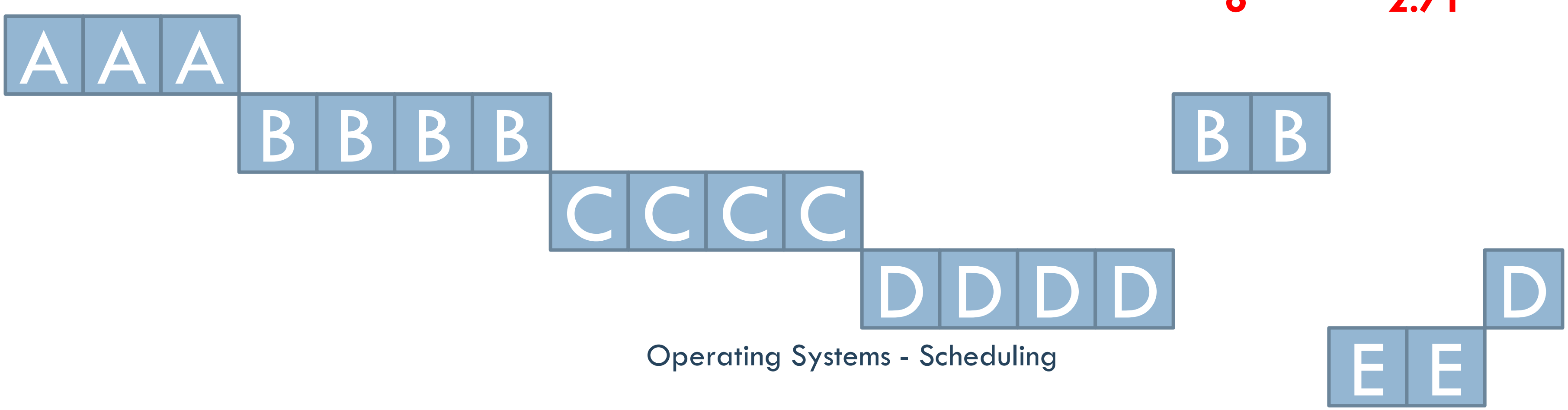
Process	Arrival	Service	Init	End	Return	Wait	Normalized Return
A	0	3	0	5	4	1	4/3=1.33
B	2	6	2	17	16	10	16/6=2.66
C	4	4	5	13	13	9	13/4=3.25
D	6	5	9	20	14	9	14/5=2.8
E	8	2	13	15	7	5	7/2=3.5

6 **2.54**



Round-Robin (q=4)

Process	Arrival	Service	Init	End	Return	Wait	Normalized Return
A	0	3	0	3	3	0	3/3=1
B	2	6	3	17	15	9	15/6=2.5
C	4	4	7	11	7	3	7/4=1.75
D	6	5	11	20	14	9	14/5=2.8
E	8	2	17	19	11	9	11/2=5.5
						6	**2.71**



Priority Scheduling

- Each process has a priority assigned to it.
- Select first processes with higher priority.

- Alternatives:
  - Fixed priorities → starvation problem.
  - **Solution**: aging mechanisms.

Scheduling in Windows

- Main characteristics:
  - Priority based and uses time slices.
  - Preemptive scheduling.
  - Scheduling with processor affinity.

- Scheduling at thread level (not process level).

- A thread may be evited from CPU if another with higher priority becomes ready.

- Scheduling decisions:
  - New threads → Ready.
  - Blocked threads receiving its event → Ready.
  - Thread leaves processor if time slice expires, it terminates or becomes blocked.

Summary

- Process creation implies memory image and PCB creation.
- A process transitions through different states during its execution.
- Operating system is responsible for process scheduling.
- Scheduling may be preemptive or non-preemptive.
- Different process scheduling algorithms may favor a certain type of processes.
- Modern Operating systems use preemptive scheduling.