CPE 460 Operating System Design

Lecture 3: Once Upon a Process

Ahmed Tamrawi

February 15, 2017

Any program to run **must** be loaded in memory

```
// File: test.c
#include <stdio.h>

int main() {
    printf(“I love Mansaf!\n”);
    return 0;
}
```

```
gcc -o test test.c
```

```
// File: test.c
#include <stdio.h>

int main() {
    printf("I love Mansaf!\n");
    return 0;
}
```

```
gcc –o test test.c
```

Program becomes process when executable file loaded into memory

```c
// File: test.c
#include <stdio.h>

int main() {
    printf(“I love Mansaf!\n”);
    return 0;
}
```

```
gcc –o test test.c
```

```
objdump -d test
```

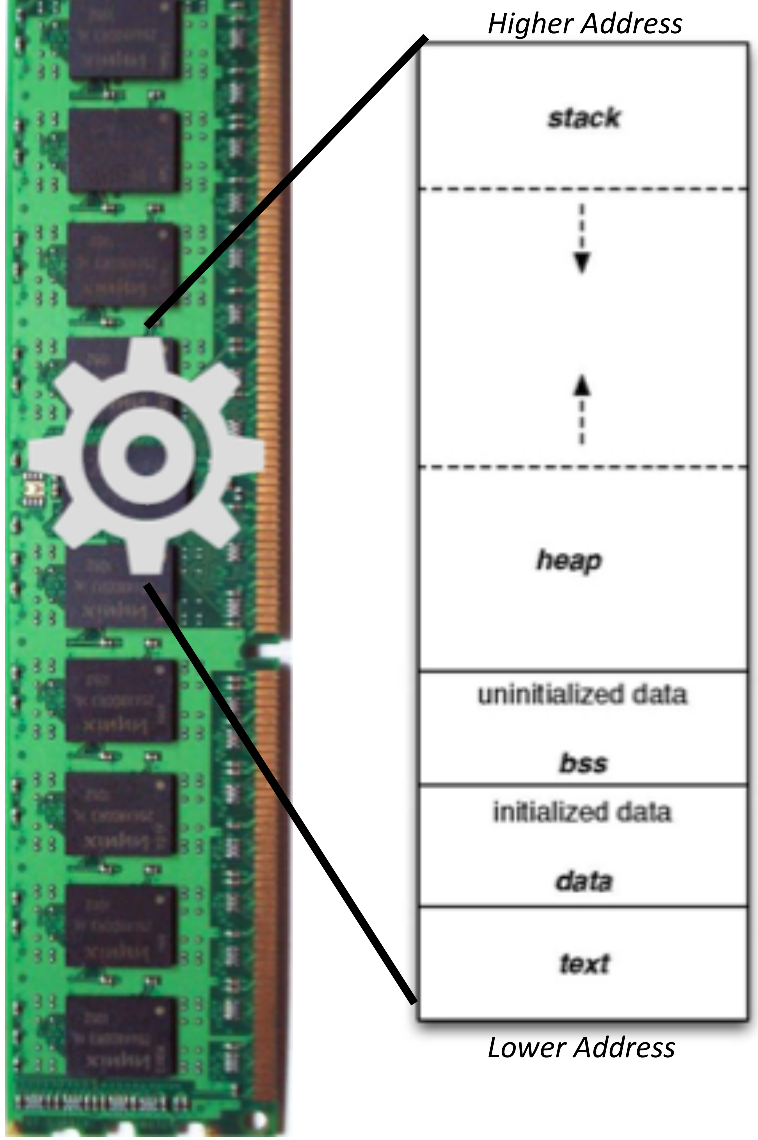
```
Disassembly of section __TEXT,__text:
__text:
100000f50:      55          pushq   %rbp
100000f51:      48 89 e5            movq    %rsp, %rbp
100000f54:      48 83 ec 10         subq    $16, %rsp
100000f58:      48 8d 3d 3b 00 00 00    leaq    59(%rip), %rdi
100000f5f:      c7 45 fc 00 00 00 00    movl    $0, -4(%rbp)
100000f66:      b0 00       movb    $0, %al
100000f68:      e8 0d 00 00 00  callq   13
100000f6d:      31 c9       xorl    %ecx, %ecx
100000f6f:      89 45 f8            movl    %eax, -8(%rbp)
100000f72:      89 c8       movl    %ecx, %eax
100000f74:      48 83 c4 10         addq    $16, %rsp
100000f78:      5d          popq    %rbp
100000f79:      c3          retq

_main:
100000f50:      55          pushq   %rbp
100000f51:      48 89 e5            movq    %rsp, %rbp
100000f54:      48 83 ec 10         subq    $16, %rsp
100000f58:      48 8d 3d 3b 00 00 00    leaq    59(%rip), %rdi
100000f5f:      c7 45 fc 00 00 00 00    movl    $0, -4(%rbp)
100000f66:      b0 00       movb    $0, %al
100000f68:      e8 0d 00 00 00  callq   13
100000f6d:      31 c9       xorl    %ecx, %ecx
100000f6f:      89 45 f8            movl    %eax, -8(%rbp)
100000f72:      89 c8       movl    %ecx, %eax
100000f74:      48 83 c4 10         addq    $16, %rsp
100000f78:      5d          popq    %rbp
100000f79:      c3          retq
Disassembly of section __TEXT,__stubs:
__stubs:
100000f7a:      ff 25 90 00 00 00       jmpq    *144(%rip)
Disassembly of section __TEXT,__stub_helper:
__stub_helper:
100000f80:      4c 8d 1d 81 00 00 00    leaq    129(%rip), %r11
100000f87:      41 53       pushq   %r11
100000f89:      ff 25 71 00 00 00       jmpq    *113(%rip)
100000f8f:      90          nop
100000f90:      68 00 00 00 00  pushq   $0
100000f95:      e9 e6 ff ff ff  jmp     -26 <__stub_helper>
```

http://www.thegeekstuff.com/2012/09/objdump-examples/?utm_source=feedburner

https://jvns.ca/blog/2014/09/06/how-to-read-an-executable/

Process Memory Layout

https://en.wikipedia.org/wiki/Data_segment



Stack Area contains the program stack, a LIFO structure. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The **stack area** contains temporary data: function parameters, return addresses, and local variables.

Heap Area is the memory that is dynamically allocated during process run time. The heap area is managed by `malloc`, `calloc`, `realloc`, and `free`, which may use the `brk` and `sbrk` system calls to adjust its size

BSS Data Segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

Initialized Data Segment contains any global or static variables which have a pre-defined value and can be modified

Text (Code) Segment is one of the sections of a program in an object file or in memory, which contains executable instructions

Process execution *must* progress in ***sequential*** fashion

```c
#include <stdio.h>
int main(void) {
    return 0;
}
```

```
$ gcc memory-layout.c -o memory-layout
$ size memory-layout
text       data       bss        dec        hex        filename
960        248        8          1216       4c0        memory-layout
```

```c
#include <stdio.h>
int global;
int main(void) {
    return 0;
}
```

```
$ gcc memory-layout.c -o memory-layout
$ size memory-layout
text       data       bss        dec        hex        filename
960        248        12         1216       4c0        memory-layout
```

```c
#include <stdio.h>
int global;
int main(void) {
    static int i;
    return 0;
}
```

```
$ gcc memory-layout.c -o memory-layout
$ size memory-layout
text       data       bss        dec        hex        filename
960        248        16         1216       4c0        memory-layout
```

http://www.geeksforgeeks.org/memory-layout-of-c-program/

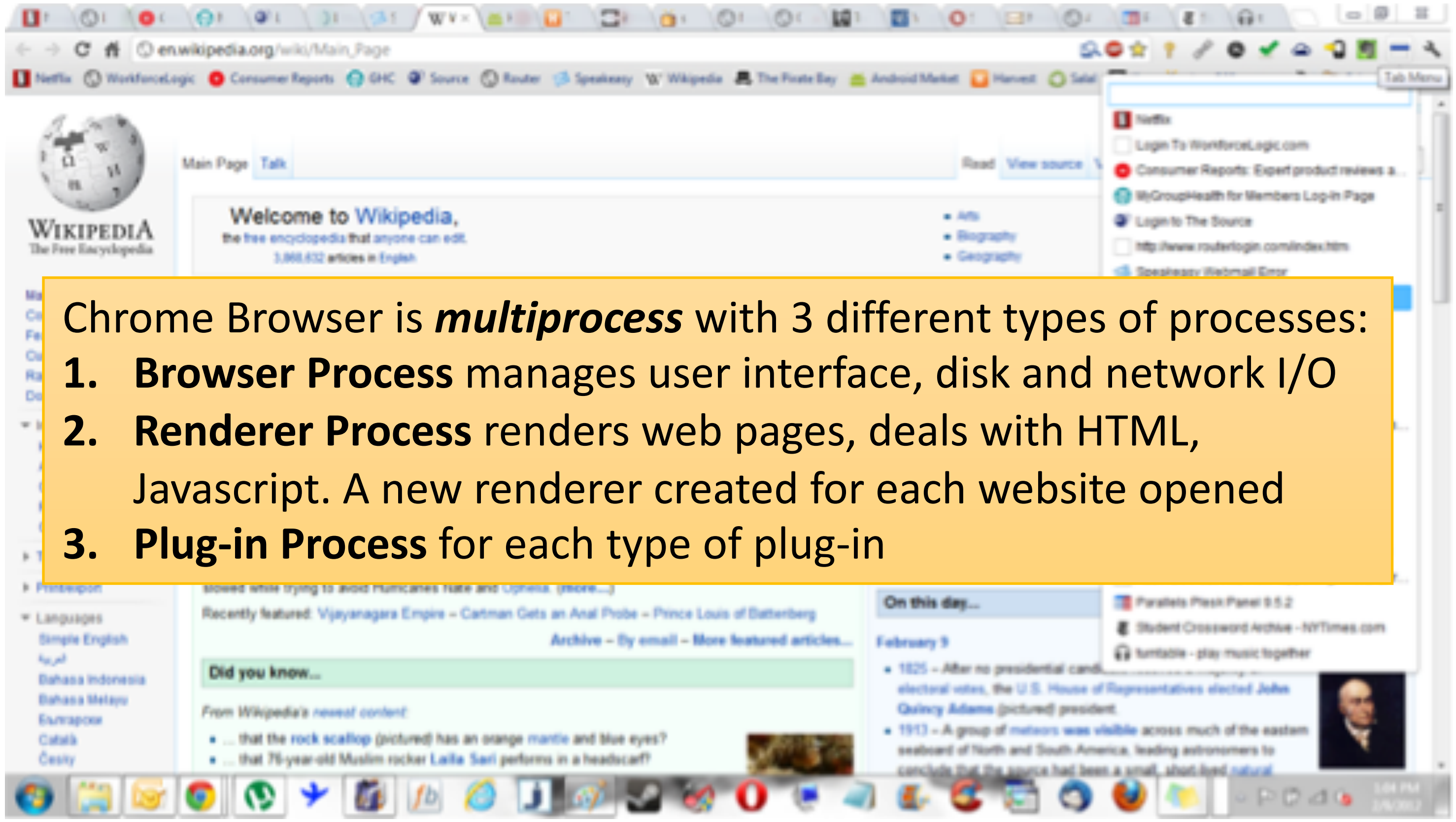
```c
#include <stdio.h>
int global = 10;
int main(void) {
    static int i = 100;
    return 0;
}
```

```
$ gcc memory-layout.c -o memory-layout
$ size memory-layout
text       data       bss        dec        hex        filename
960        256        8          1216       4c0        memory-layout
```

```c
#include <stdio.h>
int main(void) {
    printf(“hello\n”)’
    return 0;
}
```

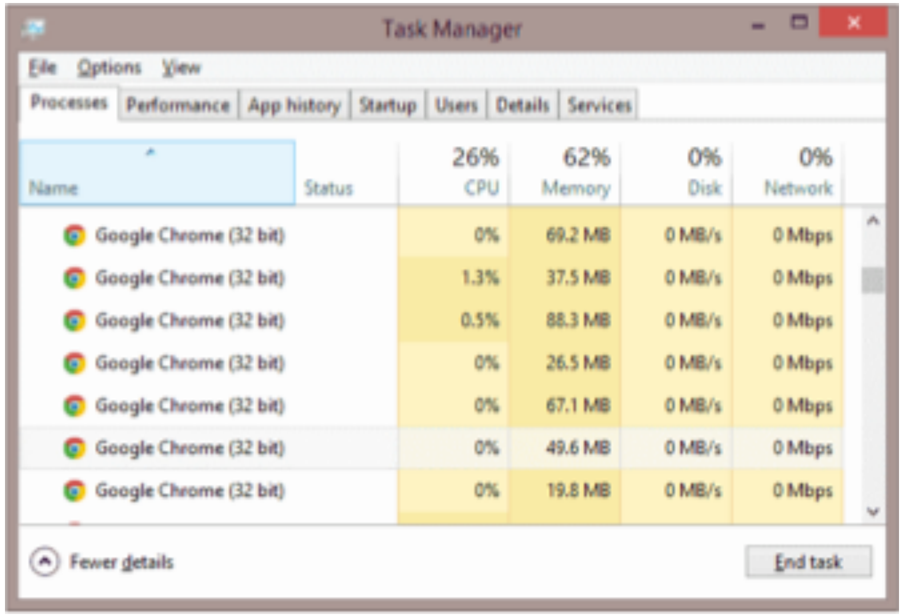
```
$ gcc memory-layout.c -o memory-layout
$ size memory-layout
text       data       bss        dec        hex        filename
960        248        8          1216       4c0        memory-layout
```

One program can be several processes



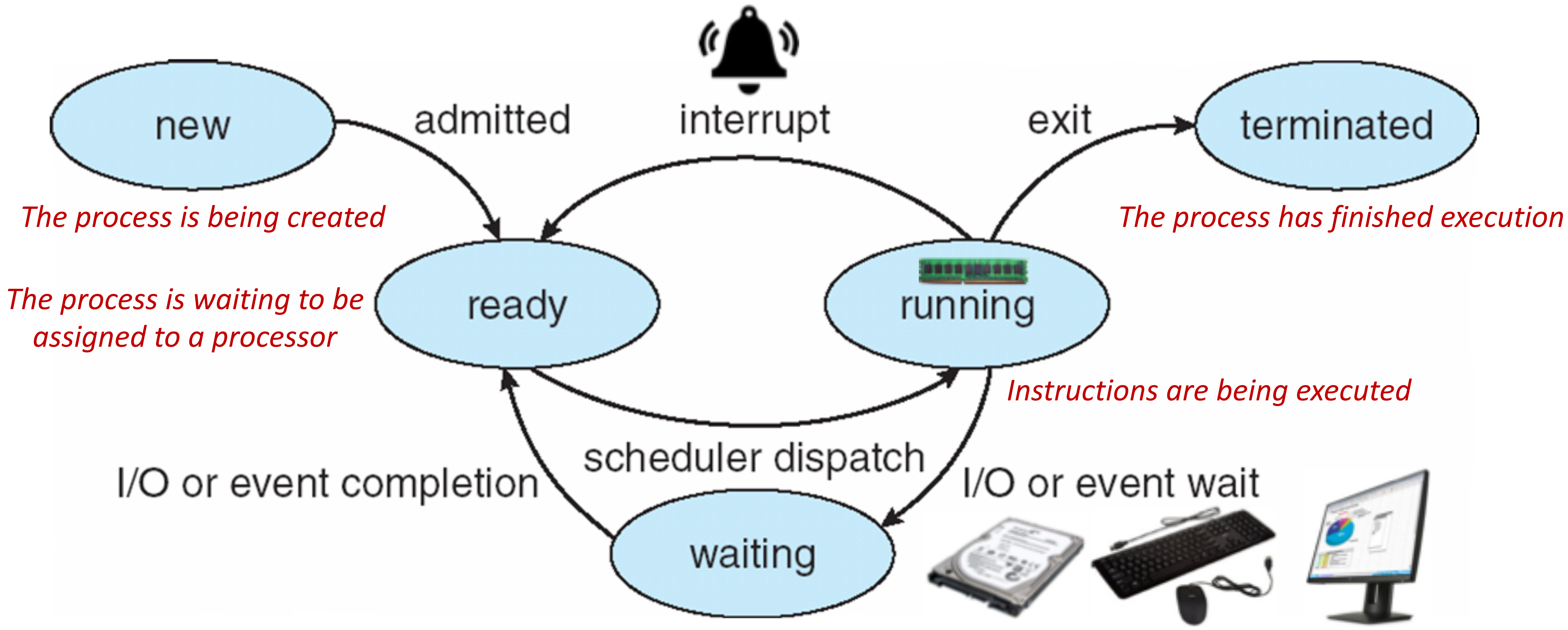
Chrome Browser is ***multiprocess*** with 3 different types of processes:

1. **Browser Process** manages user interface, disk and network I/O
2. **Renderer Process** renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
3. **Plug-in Process** for each type of plug-in



Name	Status	26% CPU	62% Memory	0% Disk	0% Network
Google Chrome (32 bit)		0%	69.2 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		1.3%	37.5 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0.5%	88.3 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	26.5 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	67.1 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	49.6 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	19.8 MB	0 MB/s	0 Mbps

Process State



new

admitted

interrupt

exit

terminated

The process is being created

The process has finished execution

The process is waiting to be assigned to a processor

ready

running

Instructions are being executed

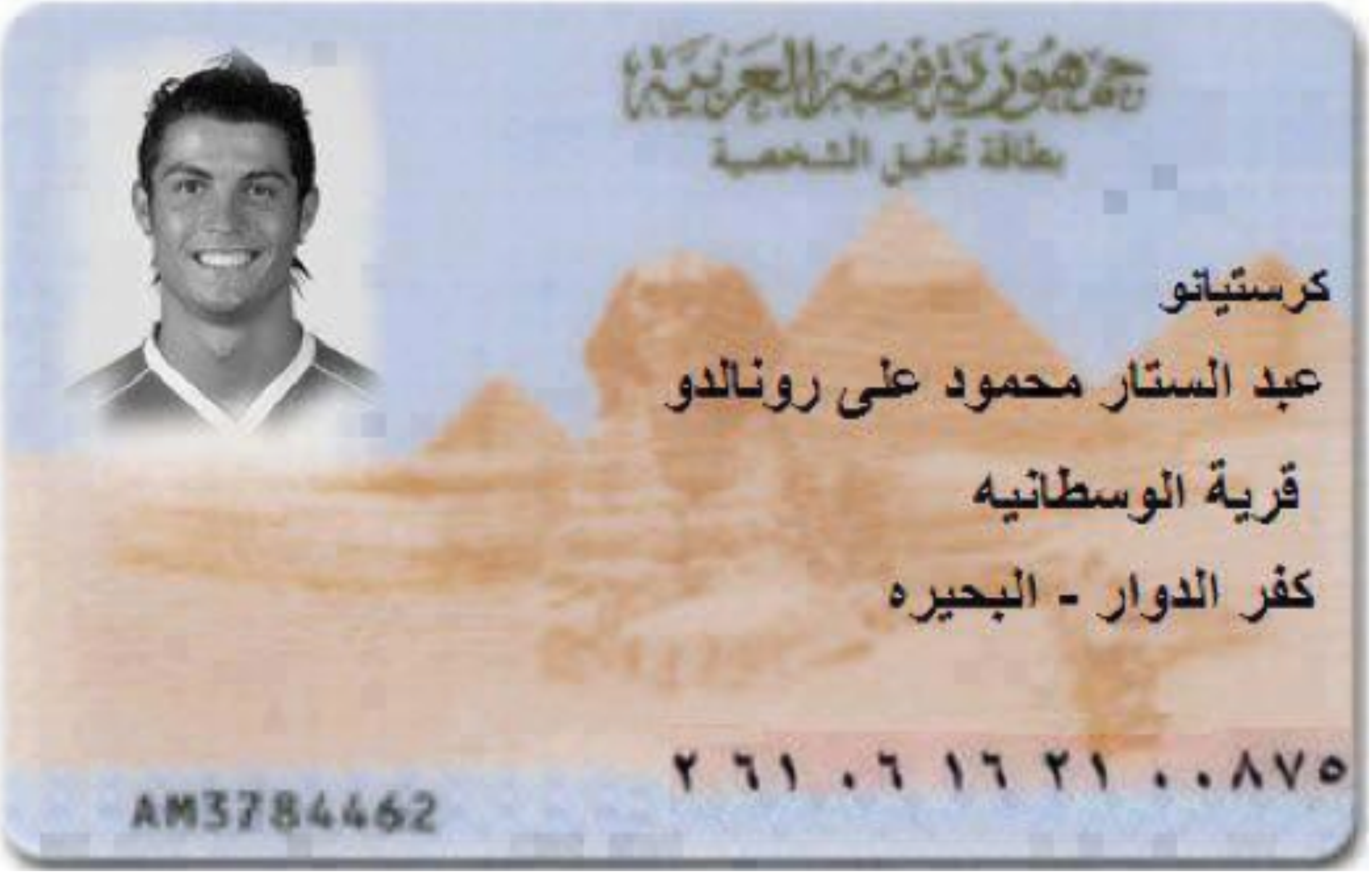
scheduler dispatch

I/O or event completion

I/O or event wait

waiting

The process is waiting for some event to occur



جمهورية مصر العربية
بطاقة تحقيق الشخصية

كرستيانو
عبد الستار محمود على رونالدو
قرية الوسطانيه
كفر الدوار - البحيره

٢ ٦١ ٠٦ ١٦ ٢١ ٠٠٨٧٥

AM3784462

OPERATING SYSTEM KINGDOM

Process Control Block (PCB)

PROCESS

PROCESS STATE:

PROGRAM COUNTER:

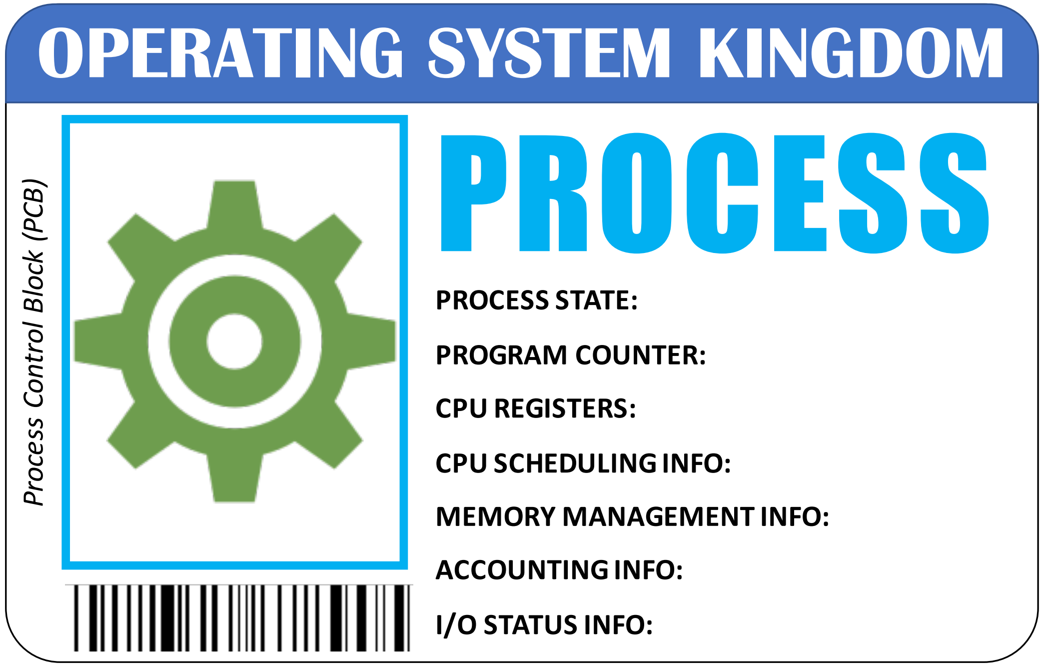
CPU REGISTERS:

CPU SCHEDULING INFO:

MEMORY MANAGEMENT INFO:

ACCOUNTING INFO:

I/O STATUS INFO:



OPERATING SYSTEM KINGDOM

PROCESS

Process Control Block (PCB)

PROCESS STATE:

PROGRAM COUNTER:

CPU REGISTERS:

CPU SCHEDULING INFO:

MEMORY MANAGEMENT INFO:

ACCOUNTING INFO:

I/O STATUS INFO:

Process Number: a unique identification number for each process in the operating system.

Process State: new, ready, running, waiting, terminated.

Program Counter: A pointer to the address of the next instruction to be executed for this process

CPU Registers: Contents of all process-centric registers. Tis state information must be saved when an ***interrupt*** occurs, to allow the process to be continued correctly afterward.

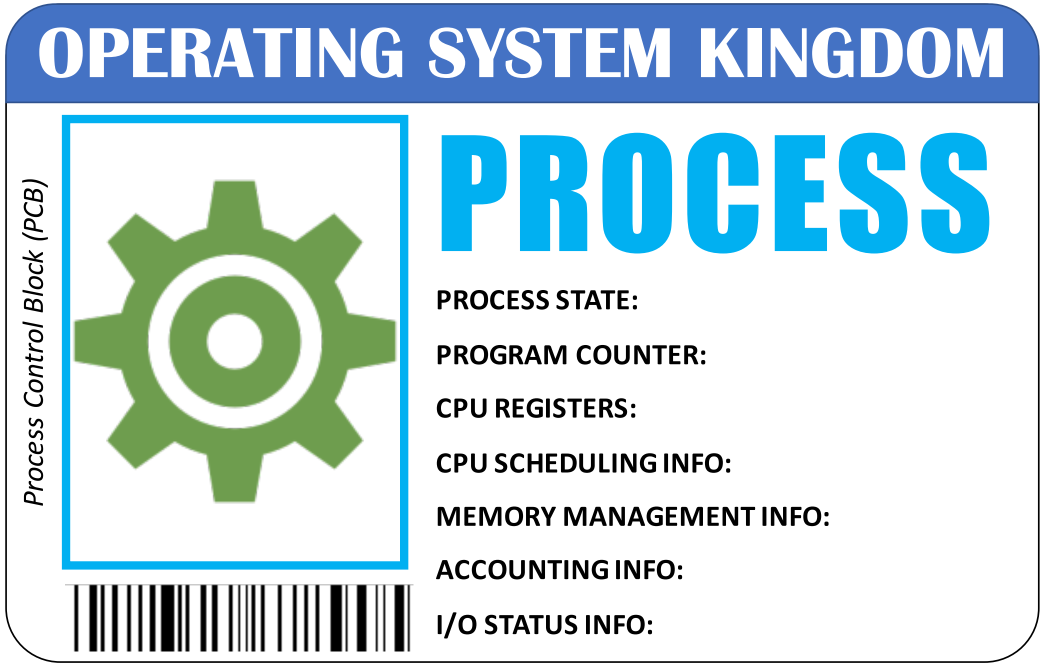
CPU Scheduling Info: Priorities, scheduling queue pointers and other scheduling parameters (*Chapter 6*)

Memory Management Info: Memory allocated to the process such as: base/limit registers and page/segment tables (*Chapter 7*)

Accounting Info: Amount of CPU and real time used, time limits, account numbers, job or process numbers.

I/O Status Info: The list of I/O devices allocated to process, list of open files

Operating Systems *differ* in Process Representation



PROCESS STATE:

PROGRAM COUNTER:

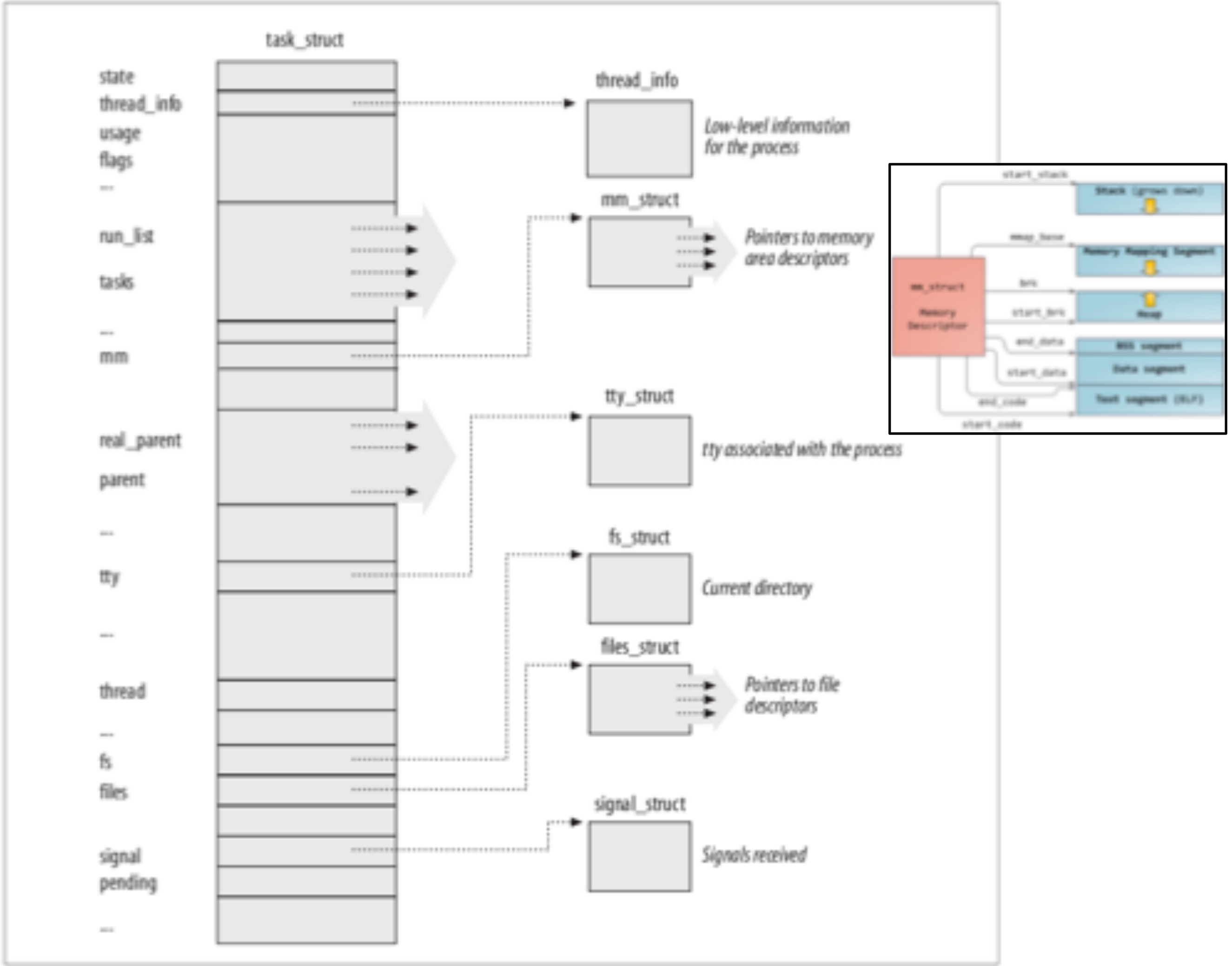
CPU REGISTERS:

CPU SCHEDULING INFO:

MEMORY MANAGEMENT INFO:

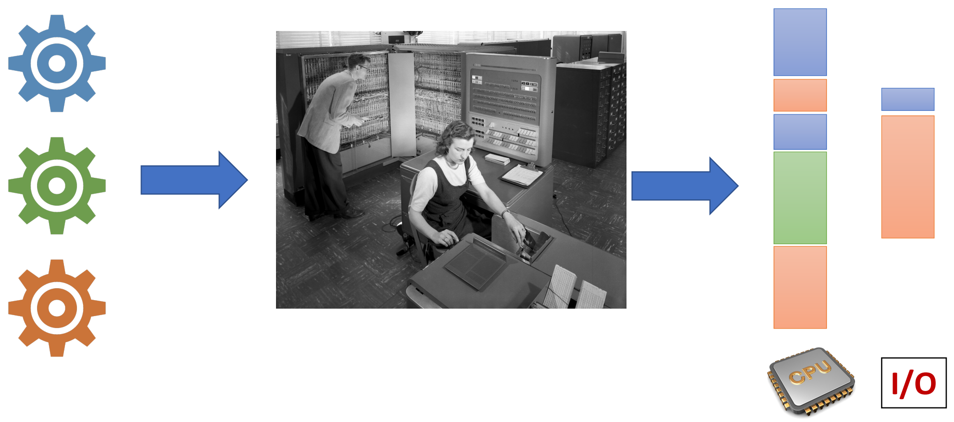
ACCOUNTING INFO:

I/O STATUS INFO:



https://github.com/torvalds/linux/blob/master/include/linux/sched.h#L1501
http://www.tldp.org/LDP/tlk/ds/ds.html

Multiprogramming (Batch System)

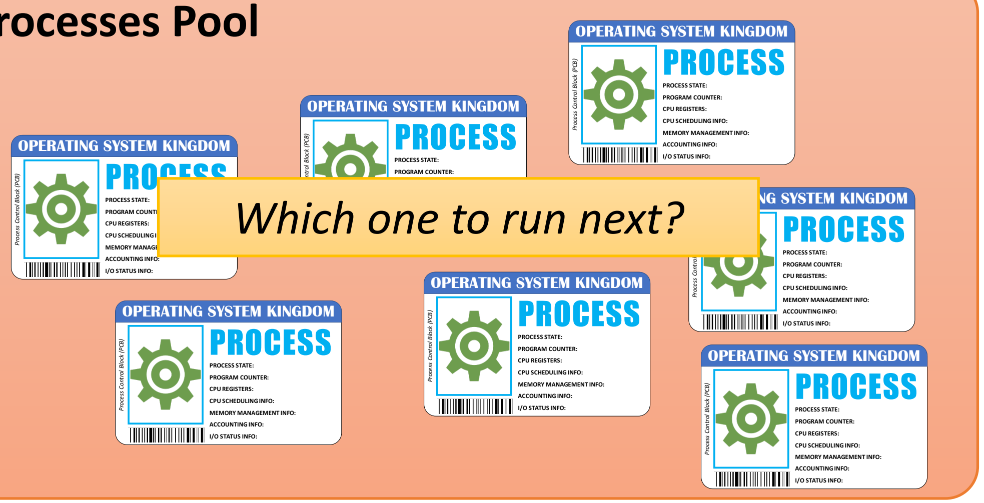


Maximize CPU use, quickly switch processes onto CPU for time sharing

Maximize throughput by increasing the number of processes that are completed per time unit

Maximize response time by decreasing the time from the submission of a request until the first response is produced

Processes Pool



Which one to run next?

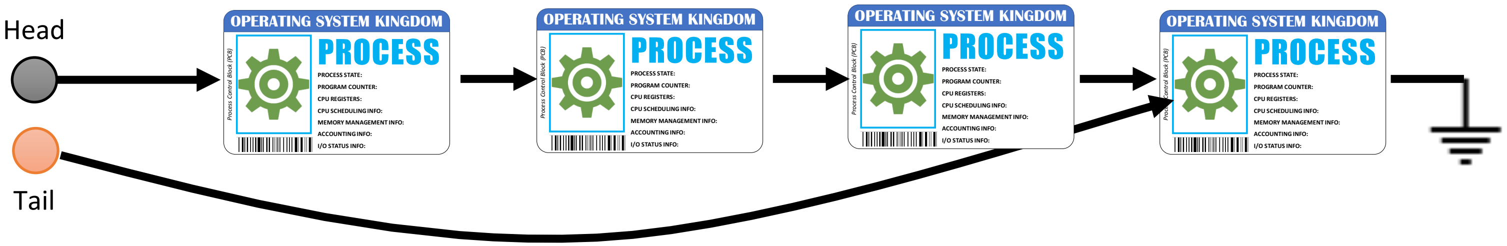
Timesharing (Multitasking)

Process Scheduler

Process scheduler selects among available processes for next execution on CPU

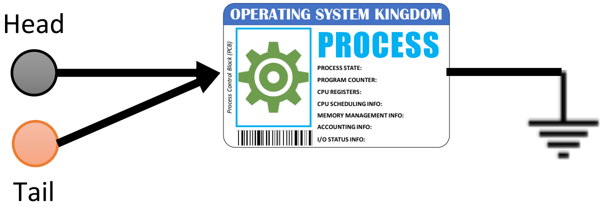
Maintains scheduling queues of processes

Processes migrate among the various queues



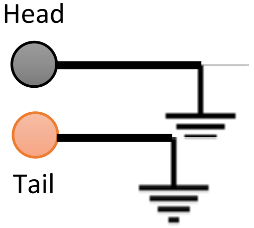
Job Queue

Set of all processes in the system



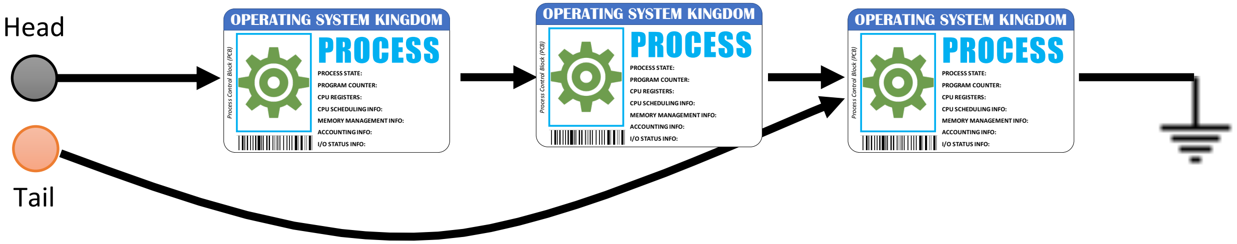
Ready Queue

Set of all processes residing in main memory, ready and waiting to execute



Disk 1 Queue

Device queue – Set of processes waiting for an I/O on Disk 1



I/O Queue

Device queue – Set of processes waiting for an I/O device

Representation of Process Scheduling

Medium-Term Scheduling

swaps out process from memory, then ***swaps*** it ***in*** ready queue

process swapping scheduler

Reduces the degree of multiprogramming

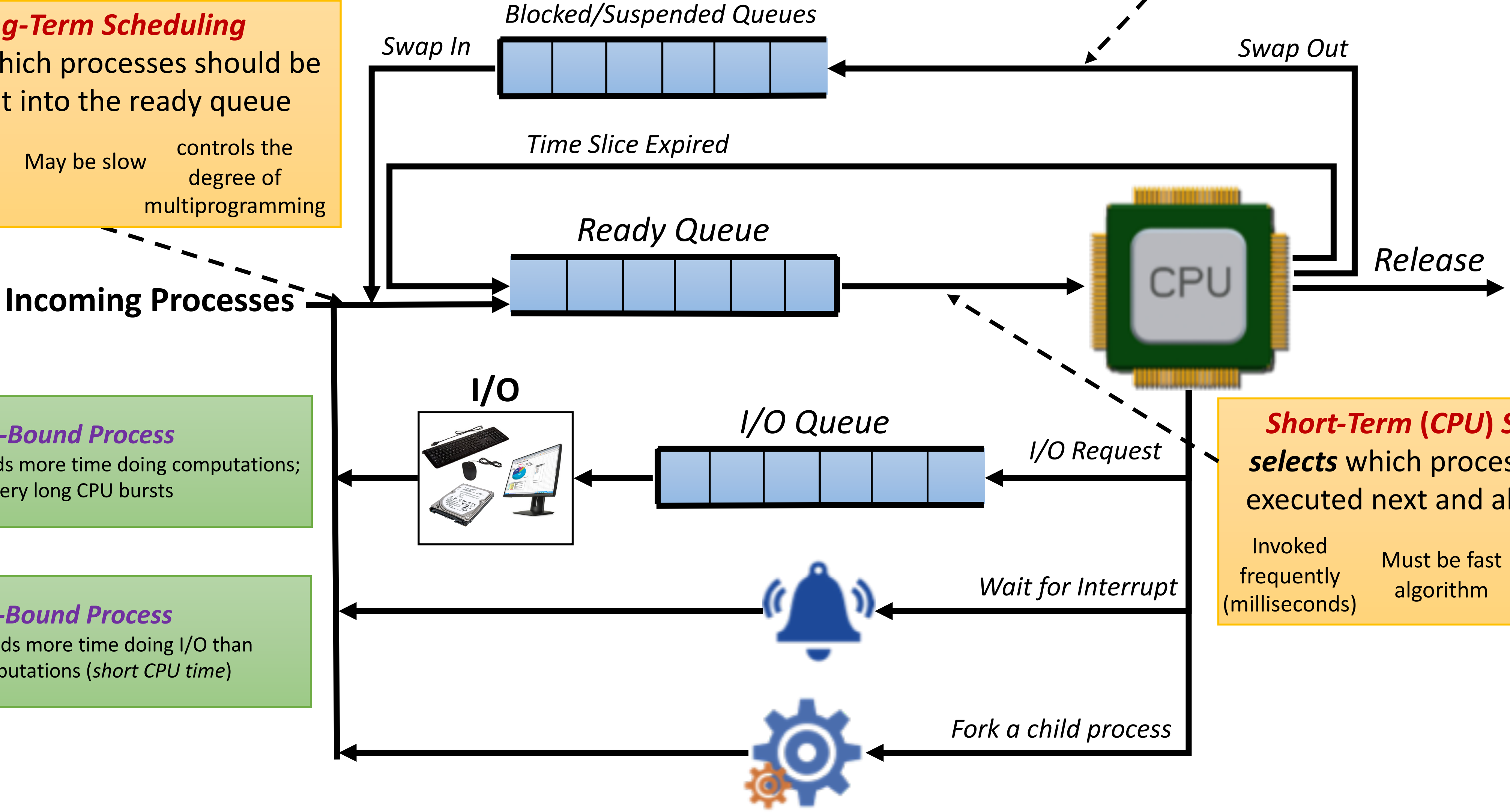
Long-Term Scheduling

selects which processes should be brought into the ready queue

Invoked **in**frequently (secs, mins)

May be slow

controls the degree of multiprogramming



Short-Term (CPU) Scheduling

selects which process should be executed next and allocates CPU

Invoked frequently (milliseconds)

Must be fast algorithm

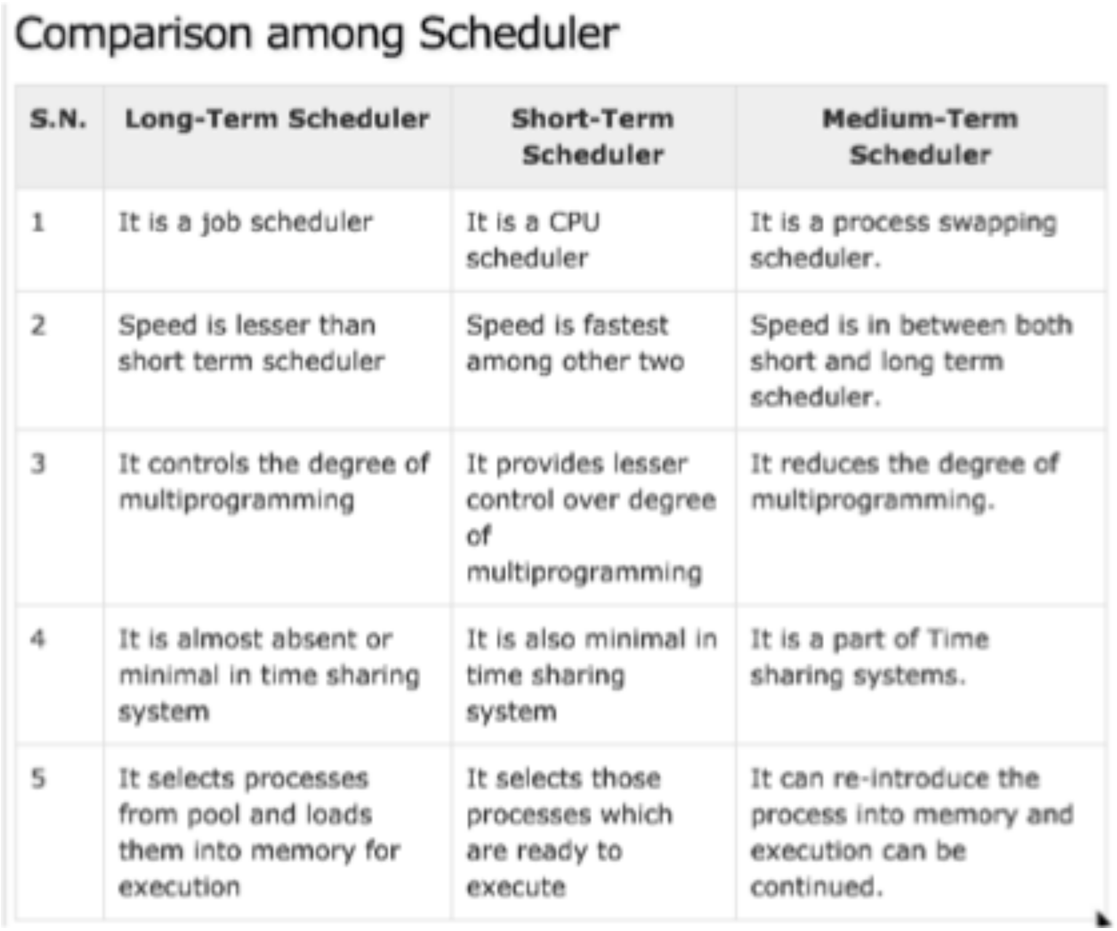
Sometimes the only scheduler in a system

CPU-Bound Process

spends more time doing computations; few very long CPU bursts

I/O-Bound Process

spends more time doing I/O than computations (*short CPU time*)



Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm

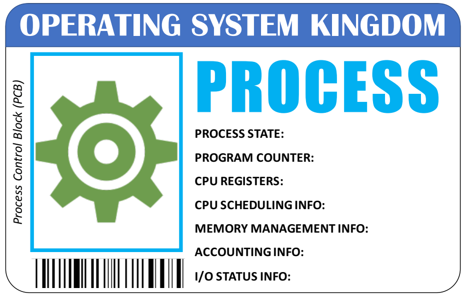
Context Switching

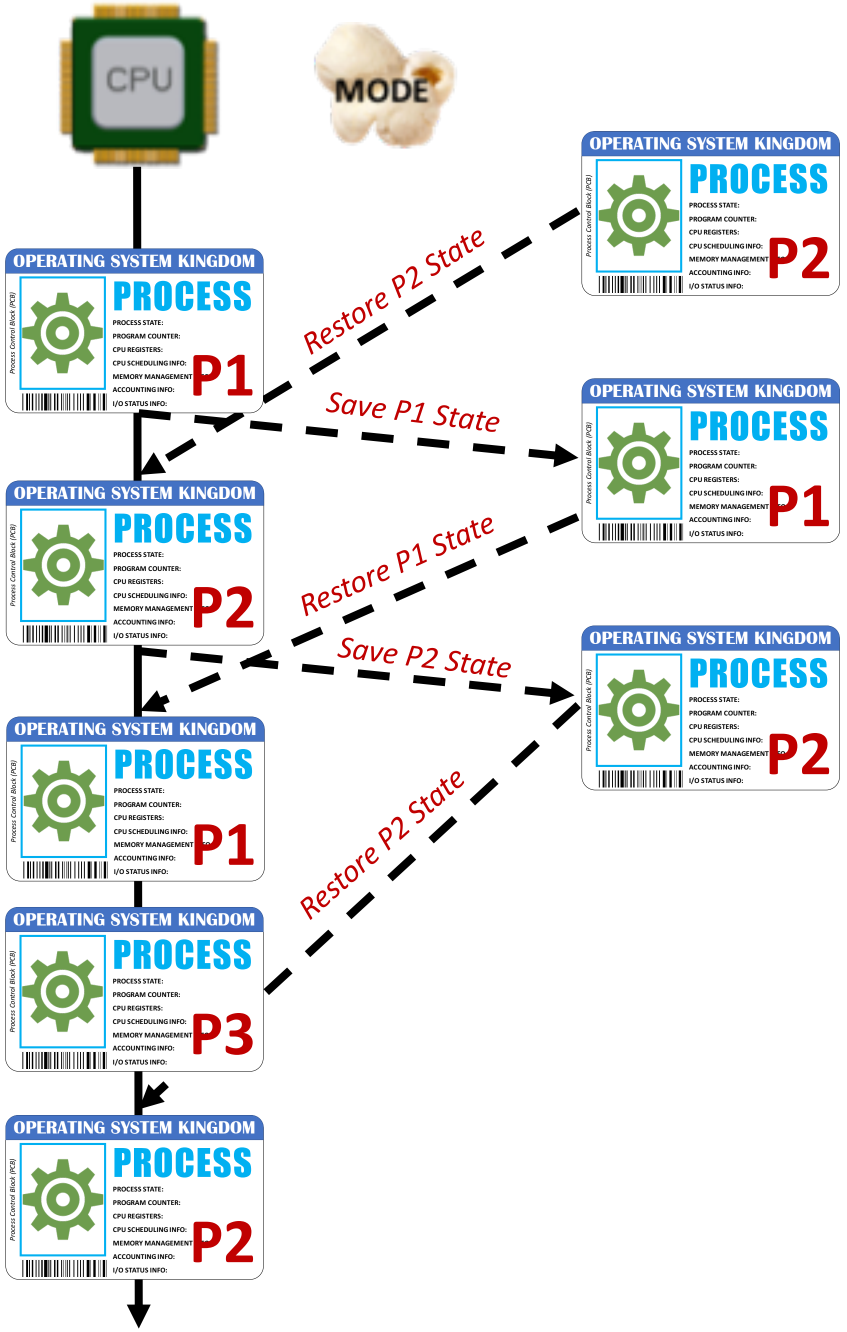
enables multiple processes to share a single CPU

The mechanism to store and restore **the state or context** of a CPU in **Process Control Block** so that a process execution can be resumed from the same point at a later time

*When the scheduler switches, the CPU switches from executing one process to another process, the system must **save the state "Context"** of the old process and **load the saved state "Context"** for the new process*

Context





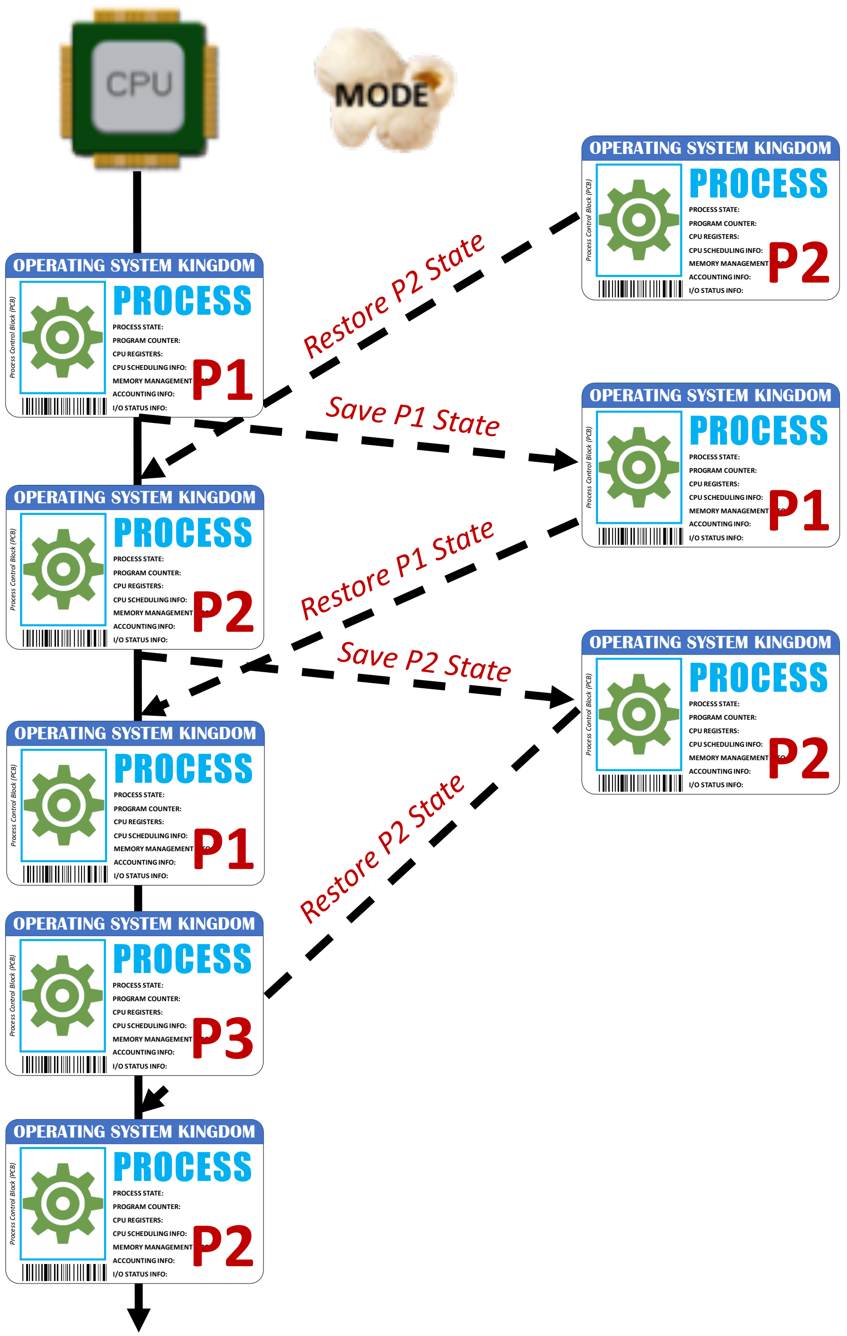
Context Switching

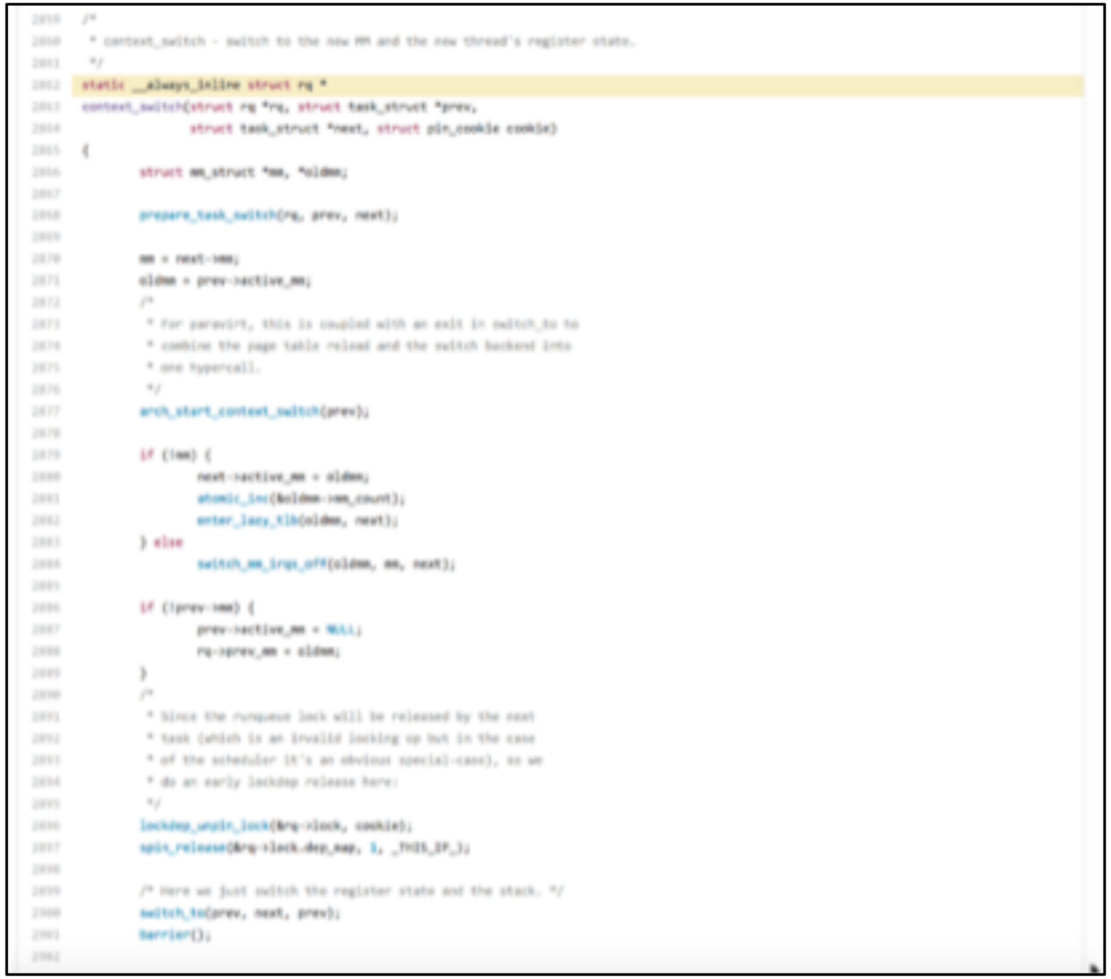
enables multiple processes to share a single CPU

Context switches are **computationally intensive** since register and memory state must be saved and restored

The more complex the OS and the PCB; the longer the context switching

To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers so that multiple contexts loaded at once.





```c
/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct pin_cookie cookie)
{
        struct mm_struct *mm, *oldmm;

        prepare_task_switch(rq, prev, next);

        mm = next->mm;
        oldmm = prev->active_mm;
        /*
         * For paravirt, this is coupled with an exit in switch_to to
         * combine the page table reload and the switch backend into
         * one hypercall.
         */
        arch_start_context_switch(prev);

        if (!mm) {
                next->active_mm = oldmm;
                atomic_inc(&oldmm->mm_count);
                enter_lazy_tlb(oldmm, next);
        } else
                switch_mm_irqs_off(oldmm, mm, next);

        if (!prev->mm) {
                prev->active_mm = NULL;
                rq->prev_mm = oldmm;
        }
        /*
         * Since the runqueue lock will be released by the next
         * task (which is an invalid locking op but in the case
         * of the scheduler it's an obvious special-case), so we
         * do an early lockdep release here:
         */
        lockdep_unpin_lock(&rq->lock, cookie);
        spin_release(&rq->lock.dep_map, 1, _THIS_IP_);

        /* Here we just switch the register state and the stack. */
        switch_to(prev, next, prev);
        barrier();
```

https://github.com/torvalds/linux/blob/master/kernel/sched/core.c#L2862

Create, Delete Communication Connection
Message Passing Model Host/Process Name
Shared-Memory Model
Transfer Status Information
Attach/Detach Remote Devices

Create/Terminate/Load/Execute Process
Get/Set Process Attributes
Wait for Time/Event
wait event, signal event
Allocate/Free/Dump Memory
Locks for Process Synchronization

System Call

Provide Abstractions

Control access to resources
Get and set permissions
Allow and deny user access

Create/Delete/Open/Close/Read/Write File
Get/Set File Attributes

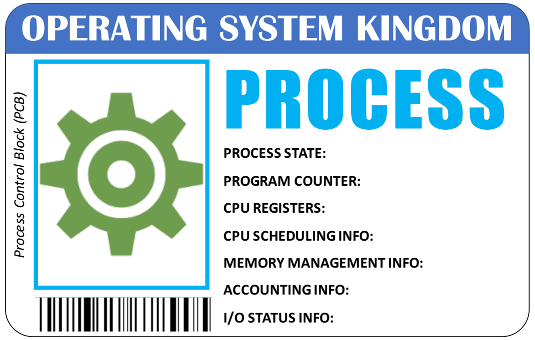
Get/Set Time or Date
Get/Set System Data

Request/Release/Read/Write Device
Get/Set Device Attributes
Logically Attach/Detach devices

Process Creation

Parent *process creates* ***children*** *processes, which, in turn create other processes, forming a* ***tree of processes***

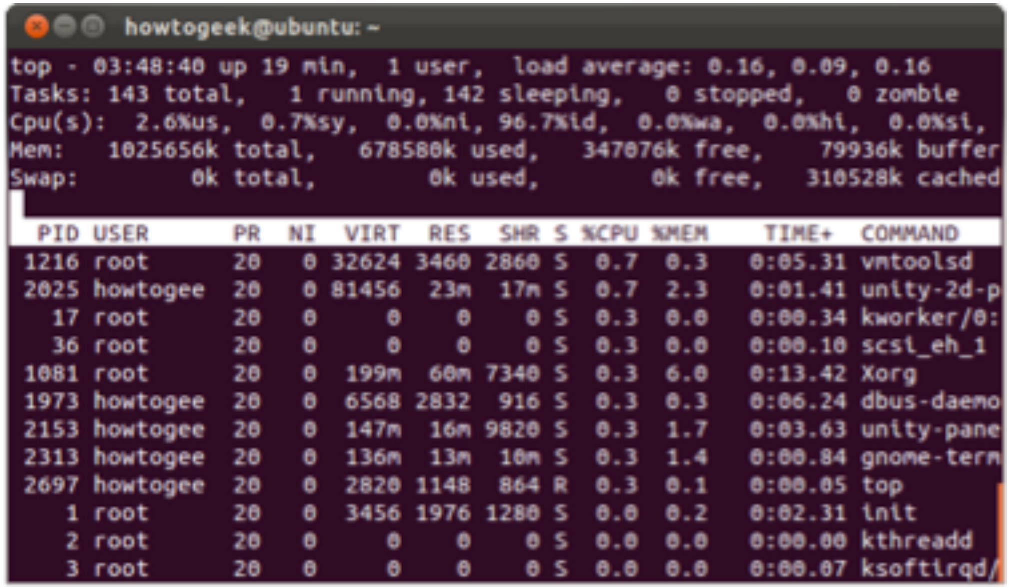
Process identified and managed via a process identifier (PID) – **Unique ID**


OPERATING SYSTEM KINGDOM

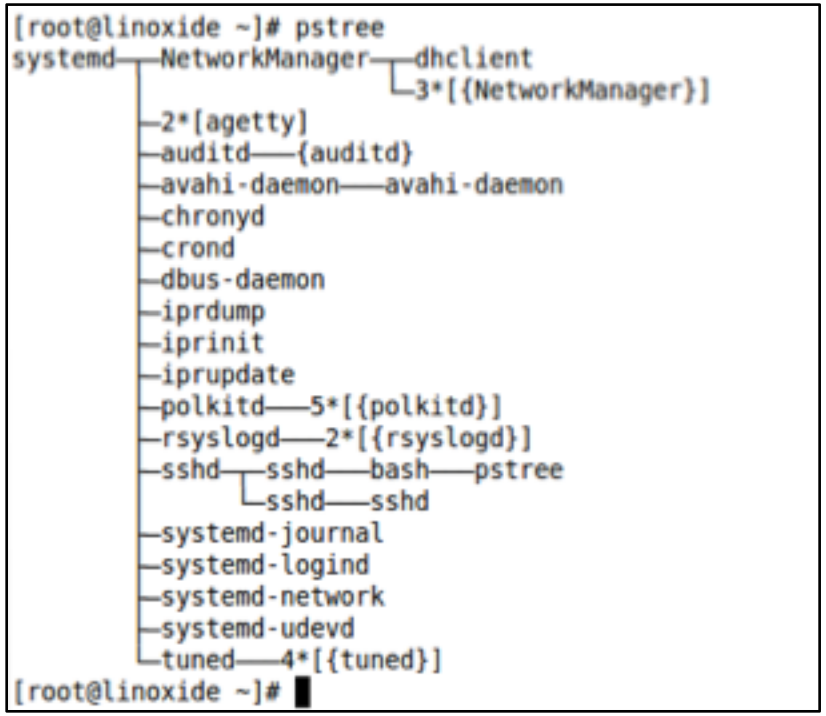
PROCESS

Process Control Block (PCB)

PROCESS STATE:
PROGRAM COUNTER:
CPU REGISTERS:
CPU SCHEDULING INFO:
MEMORY MANAGEMENT INFO:
ACCOUNTING INFO:
I/O STATUS INFO:



```
top - 03:48:40 up 19 min,  1 user,  load average: 0.16, 0.09, 0.16
Tasks: 143 total,   1 running, 142 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.6%us,  0.7%sy,  0.0%ni, 96.7%id,  0.0%wa,  0.0%hi,  0.0%si,
Mem:   1025656k total,   678580k used,   347076k free,    79936k buffer
Swap:        0k total,        0k used,        0k free,   310528k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1216 root      20   0 32624 3460 2860 S  0.7  0.3   0:05.31 vmtoolsd
 2025 howtogee  20   0 81456  23m  17m S  0.7  2.3   0:01.41 unity-2d-p
   17 root      20   0     0    0    0 S  0.3  0.0   0:00.34 kworker/0:
   36 root      20   0     0    0    0 S  0.3  0.0   0:00.10 scsi_eh_1
 1081 root      20   0  199m  60m 7340 S  0.3  6.0   0:13.42 Xorg
 1973 howtogee  20   0  6568 2832  916 S  0.3  0.3   0:06.24 dbus-daemo
 2153 howtogee  20   0  147m  16m 9820 S  0.3  1.7   0:03.63 unity-pane
 2313 howtogee  20   0  136m  13m  10m S  0.3  1.4   0:00.84 gnome-term
 2697 howtogee  20   0  2820 1148  864 R  0.3  0.1   0:00.05 top
    1 root      20   0  3456 1976 1280 S  0.0  0.2   0:02.31 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.07 ksoftirqd/
```



```
[root@linoxide ~]# pstree
systemd─┬─NetworkManager─┬─dhclient
        │                └─3*[{NetworkManager}]
        ├─2*[agetty]
        ├─auditd───{auditd}
        ├─avahi-daemon───avahi-daemon
        ├─chronyd
        ├─crond
        ├─dbus-daemon
        ├─iprdump
        ├─iprinit
        ├─iprupdate
        ├─polkitd───5*[{polkitd}]
        ├─rsyslogd───2*[{rsyslogd}]
        ├─sshd─┬─sshd───bash───pstree
        │      └─sshd───sshd
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-network
        ├─systemd-udevd
        └─tuned───4*[{tuned}]
[root@linoxide ~]# █
```

First process to run is the "`systemd`" process that is started at **system boot**. This is the grand parent of all processes in the whole system

If a process dies, then its orphan children are re-parented to the "`systemd`" process

- Parent and children share all resources
- Children share subset of parent’s resources
- Parent and child share no resources

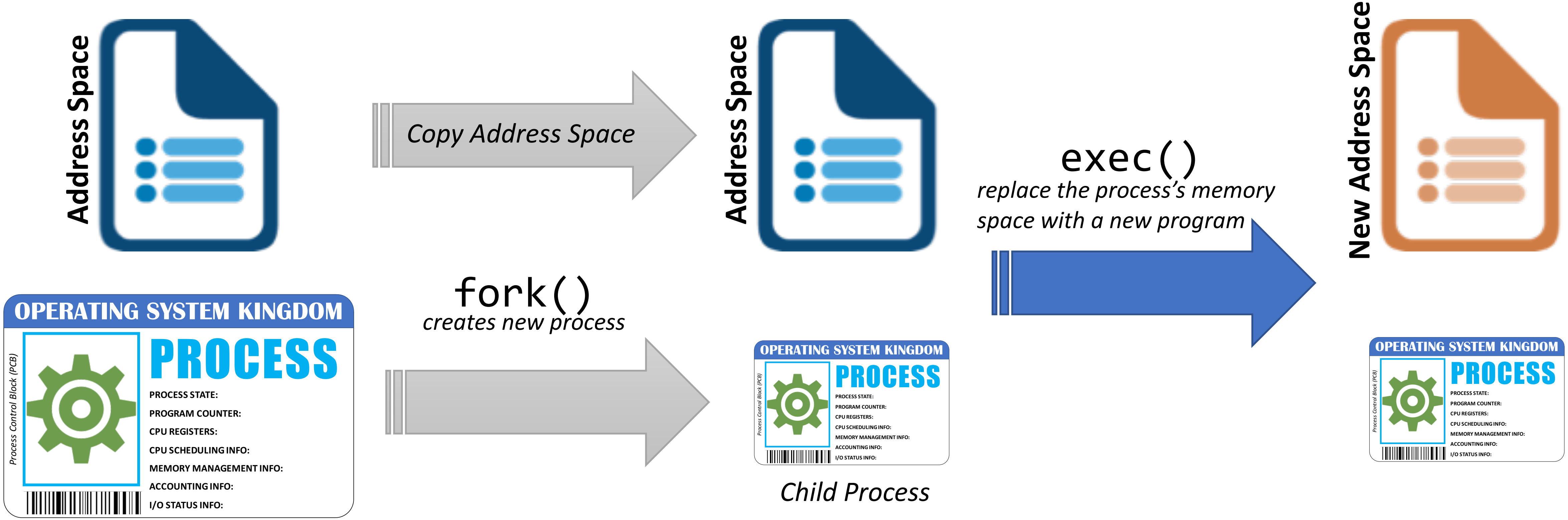
Execution

- Parent and children execute concurrently
- Parent waits until children terminate

Address Space

- Child duplicate of parent
- Child has a program loaded into it

Process Creation



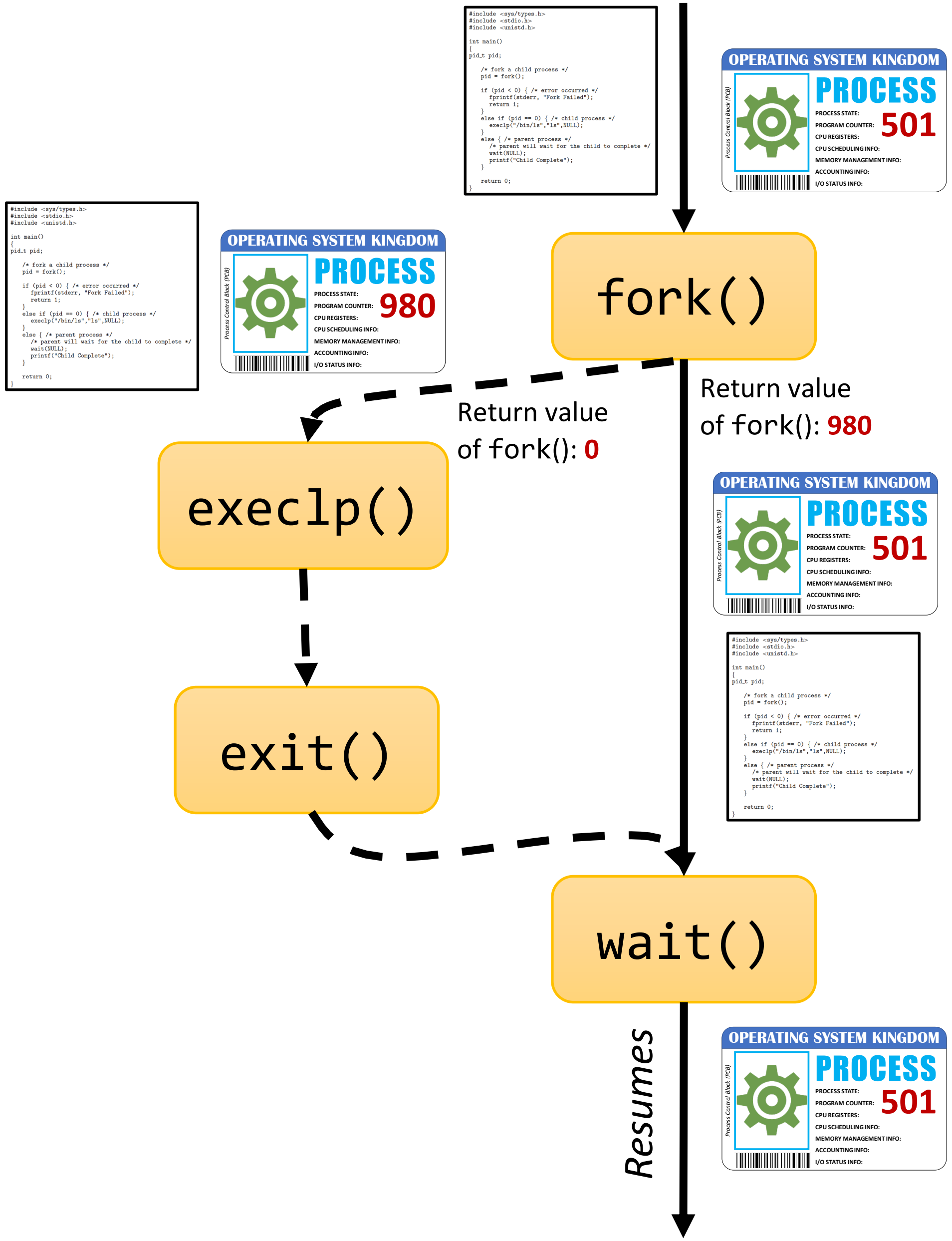
Copy Address Space

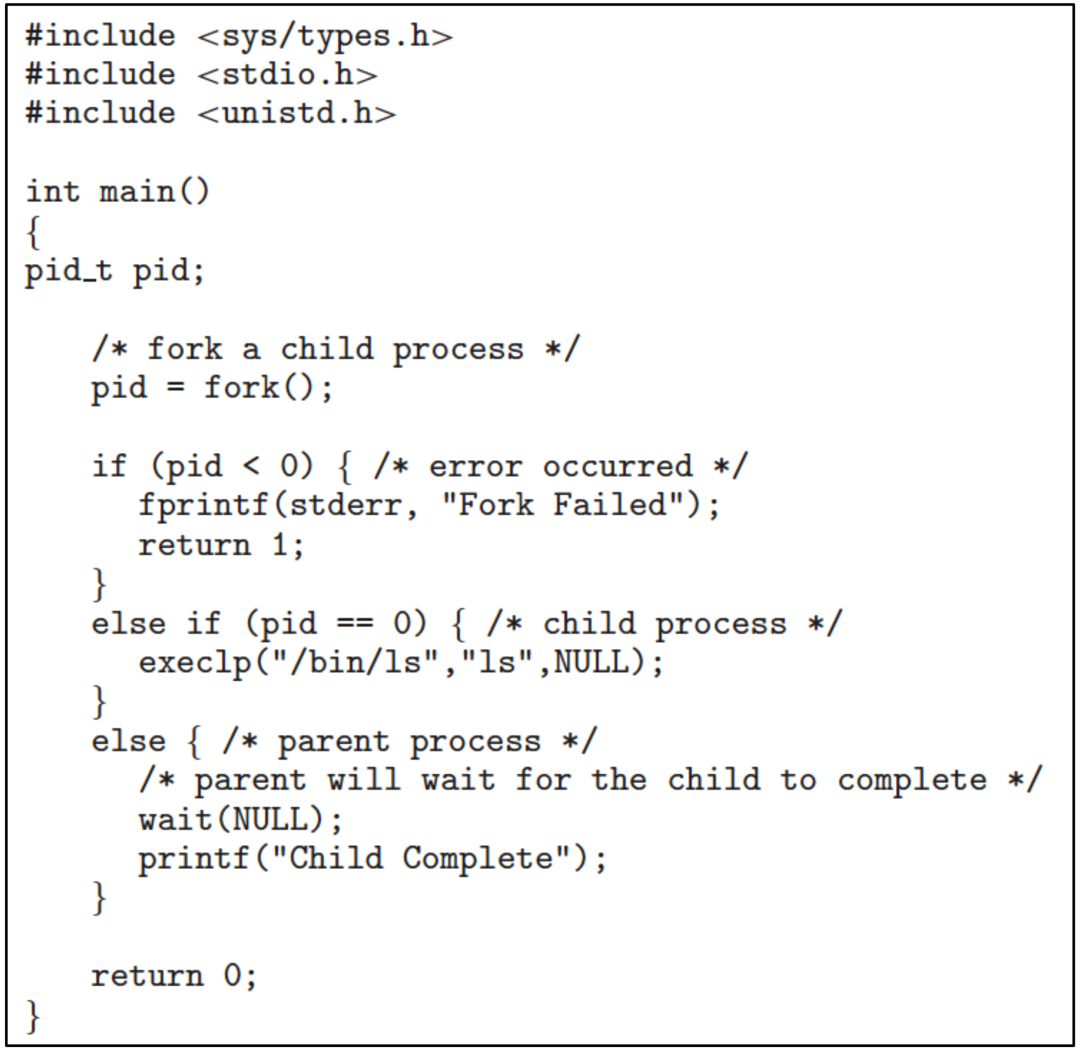
`fork()`
creates new process

Child Process

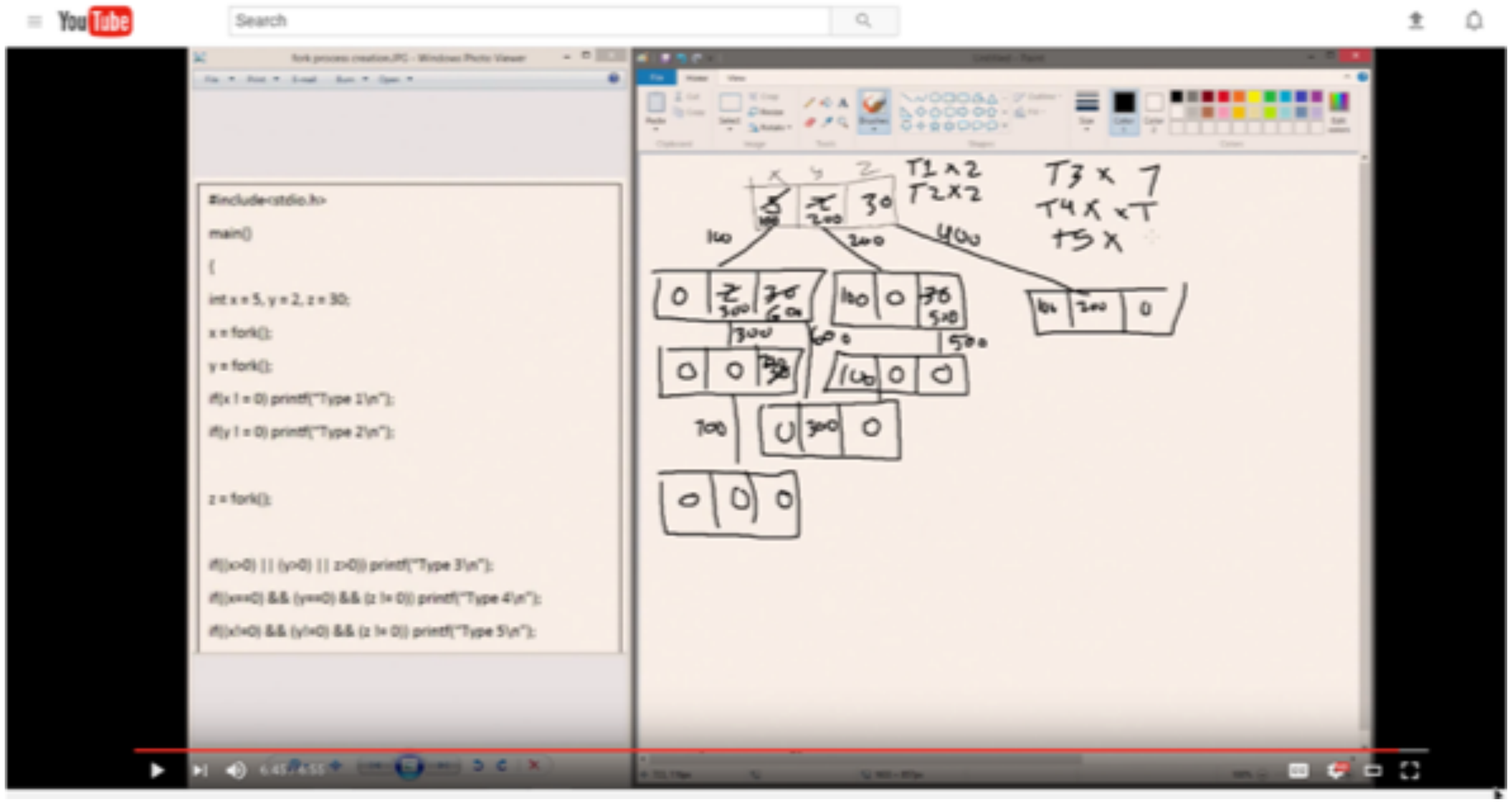
`exec()`
replace the process's memory space with a new program

Process Creation





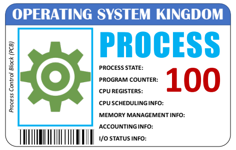
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```



```
#include<stdio.h>

main()

{

int x = 5, y = 2, z = 30;

x = fork();

y = fork();

if(x ! = 0) printf("Type 1\n");

if(y ! = 0) printf("Type 2\n");


z = fork();


if((x>0) || (y>0) || z>0)) printf("Type 3\n");

if((x==0) && (y==0) && (z != 0)) printf("Type 4\n");

if((x!=0) && (y!=0) && (z != 0)) printf("Type 5\n");
```

https://www.youtube.com/watch?v=WcsZvdlLkPw

```
int x = 5, y = 2, z = 30;

x = fork();

y = fork();

if(x != 0){
    printf(“Type 1\n”);
}

if(y != 0){
    printf(“Type 2\n”);
}

z = fork();

if((x > 0) || (y > 0) || (z > 0)){
    printf(“Type 3\n”);
}

if((x == 0) && (y == 0) && (z != 0)){
    printf(“Type 4\n”);
}

if((x != 0) && (y != 0) && (z != 0)){
    printf(“Type 5\n”);
}
```



5	**2**	**30**
x	y	z

Process Termination

Process executes last statement and then asks the OS to delete it using the **`exit`**() system call

Parent may terminate the execution of children processes using the **`abort`**() system call

Child has exceeded allocated resources OR Task assigned to child is no longer required

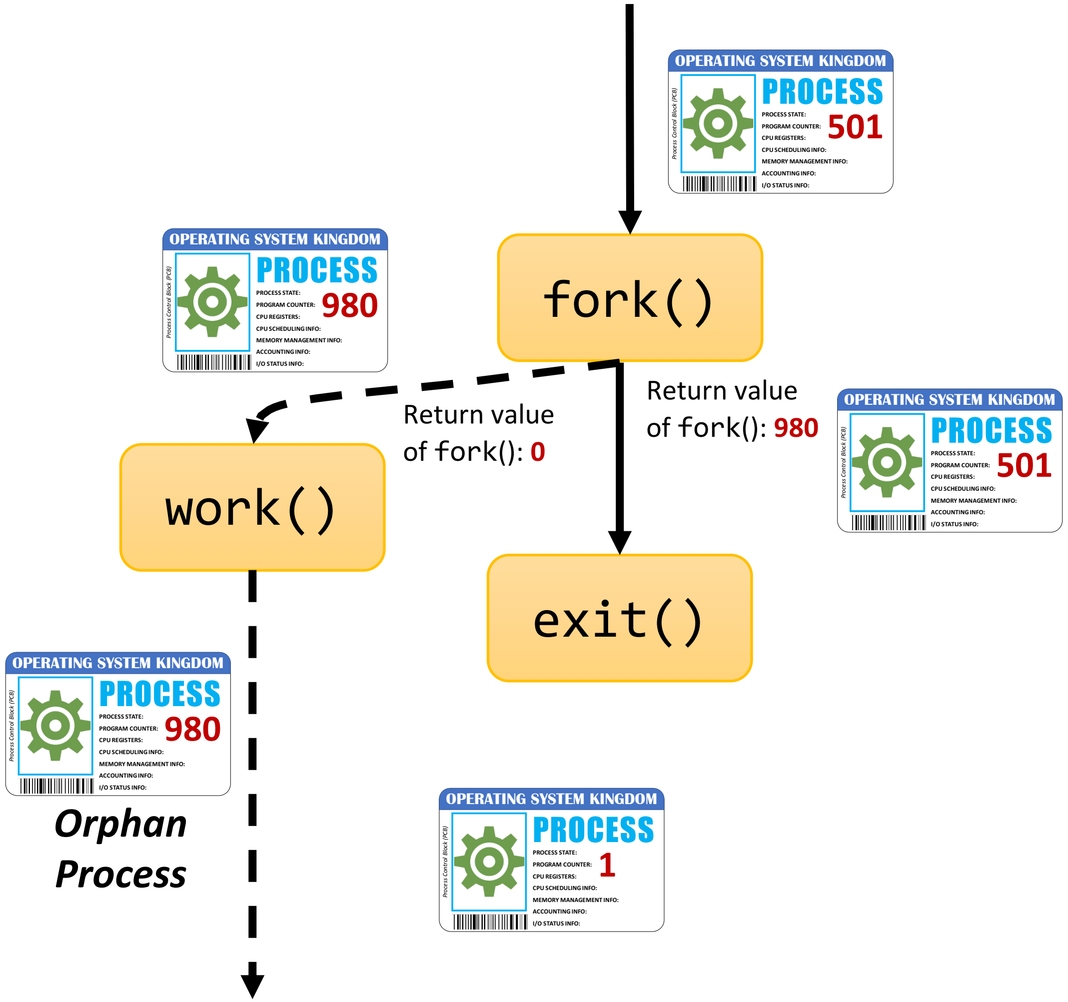
The parent process may wait for termination of a child process by using **`wait`**()

Returns status data from child to parent via **`pid = wait(&status);`**

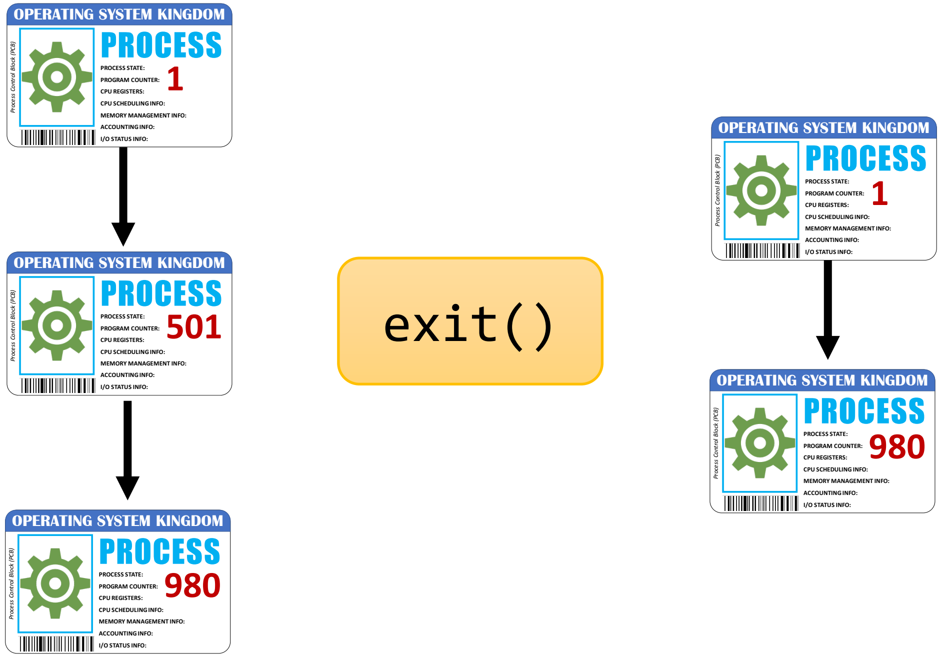
Process' resources are deallocated by OS

Orphan Process

A child process whose parent process has finished or terminated, though it remains running itself



Orphan Process

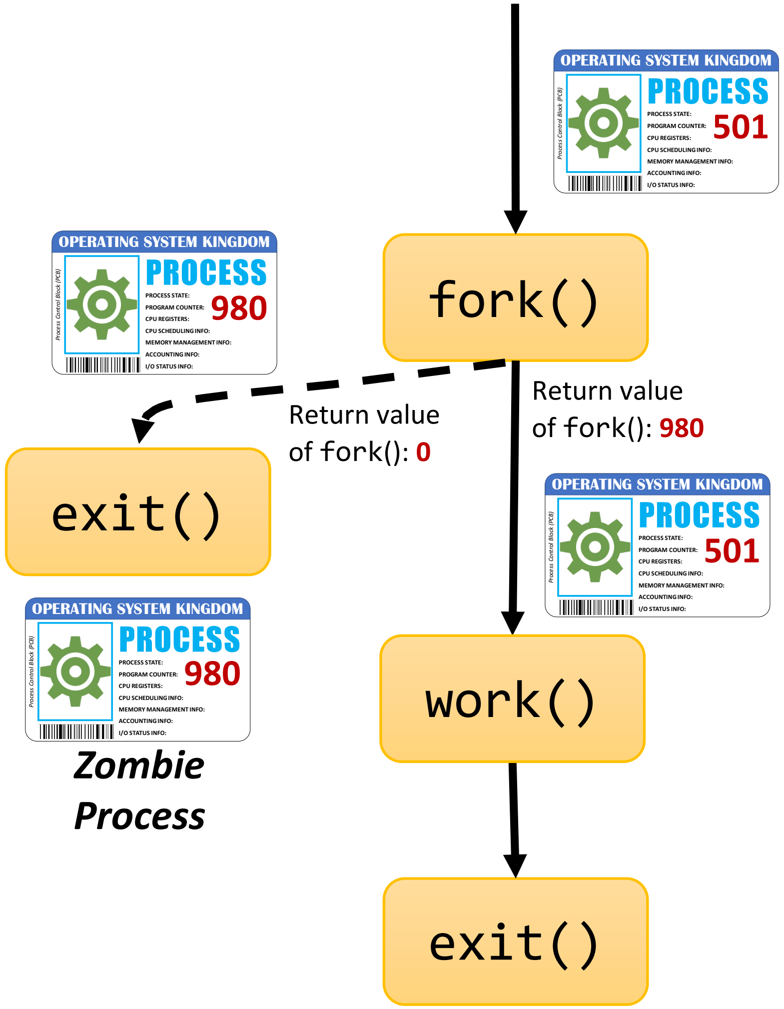


Some operating systems do not allow "**Orphan**" processes to exists. If a process terminates, then all its children must also be terminated

Some operating systems *re-parent (adopt)* all orphan processes to the *init* or *systemd* process

Zombie Process

A child process that has completed execution but has not yet been reaped



`fork()`

Return value of `fork()`: 0

Return value of `fork()`: 980

`exit()`

Zombie Process

`work()`

`exit()`

The entry for child process is still needed to allow the parent process to read its child's exit status: once the exit status is read via the **`wait()`**, the zombie's entry is removed from the process table and it is said to be "reaped"

A child process always first becomes a zombie before being removed from the resource table.

It requires a system re-boot

Representation of Process Scheduling

Medium-Term Scheduling
swaps out process from memory, then ***swaps*** it ***in*** ready queue

process swapping scheduler

Reduces the degree of multiprogramming

Long-Term Scheduling
selects which processes should be brought into the ready queue

Invoked **in**frequently (secs, mins)

May be slow

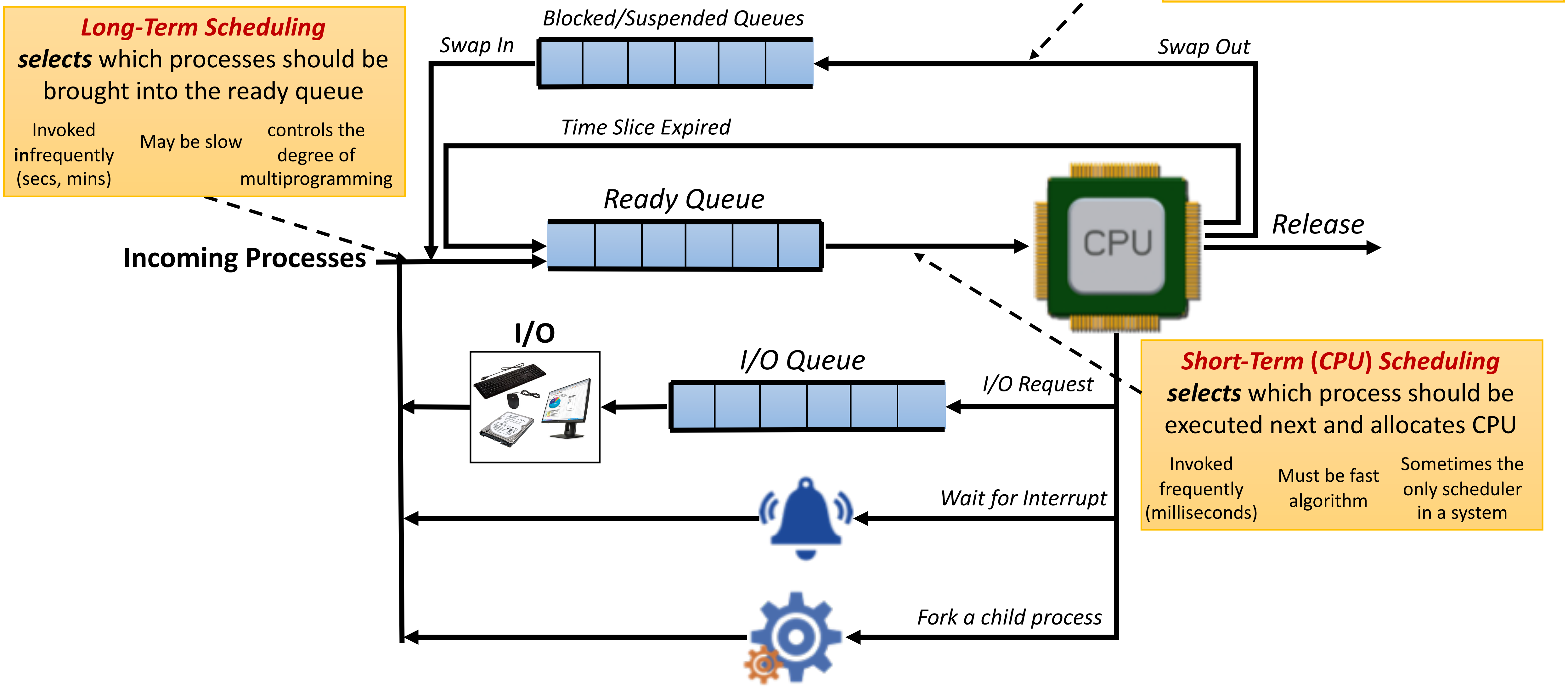
controls the degree of multiprogramming

Short-Term (CPU) Scheduling
selects which process should be executed next and allocates CPU

Invoked frequently (milliseconds)

Must be fast algorithm

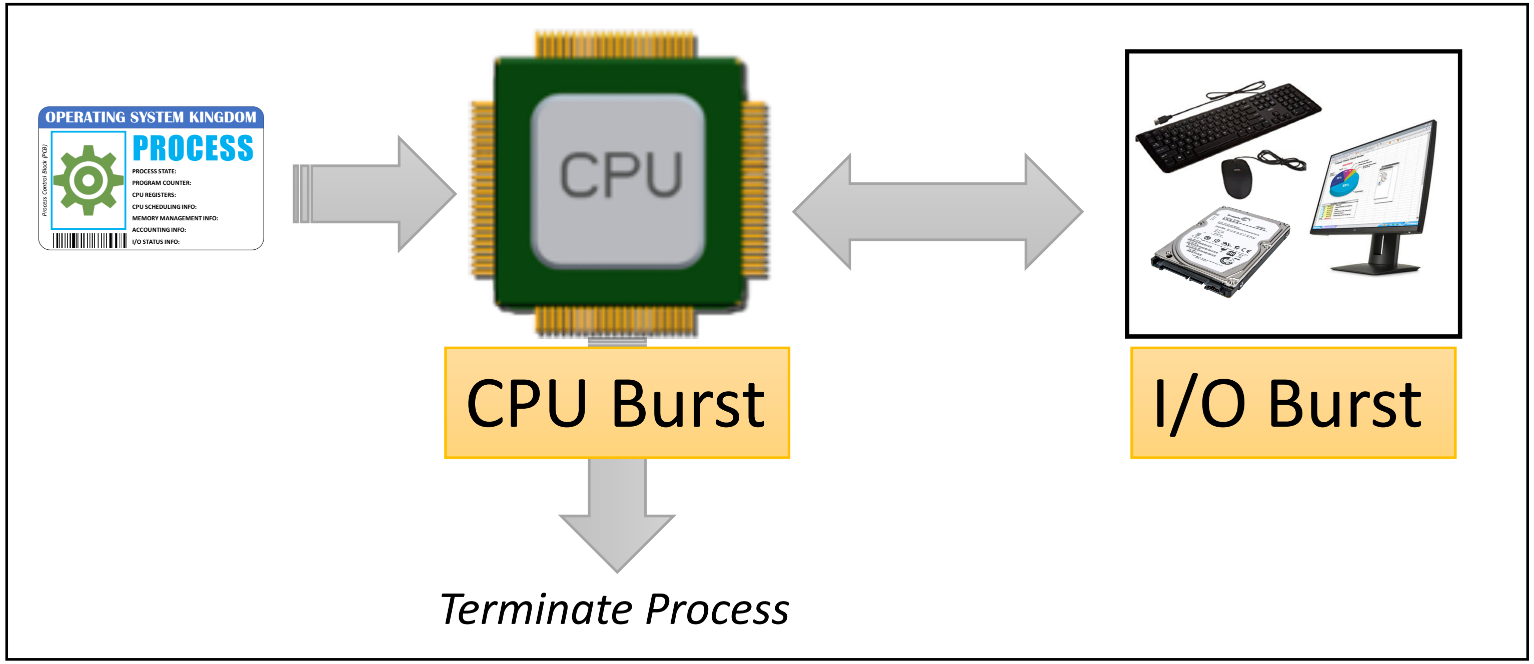
Sometimes the only scheduler in a system

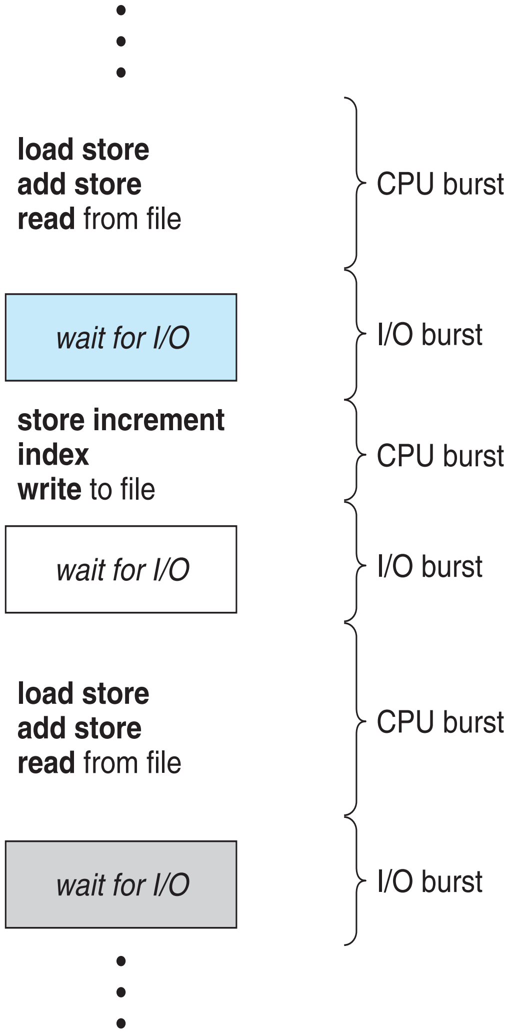


CPU Scheduling

CPU scheduler selects among available processes for next execution on CPU

Process Execution Cycle




load store
add store
read from file

wait for I/O

store increment
index
write to file

wait for I/O

load store
add store
read from file

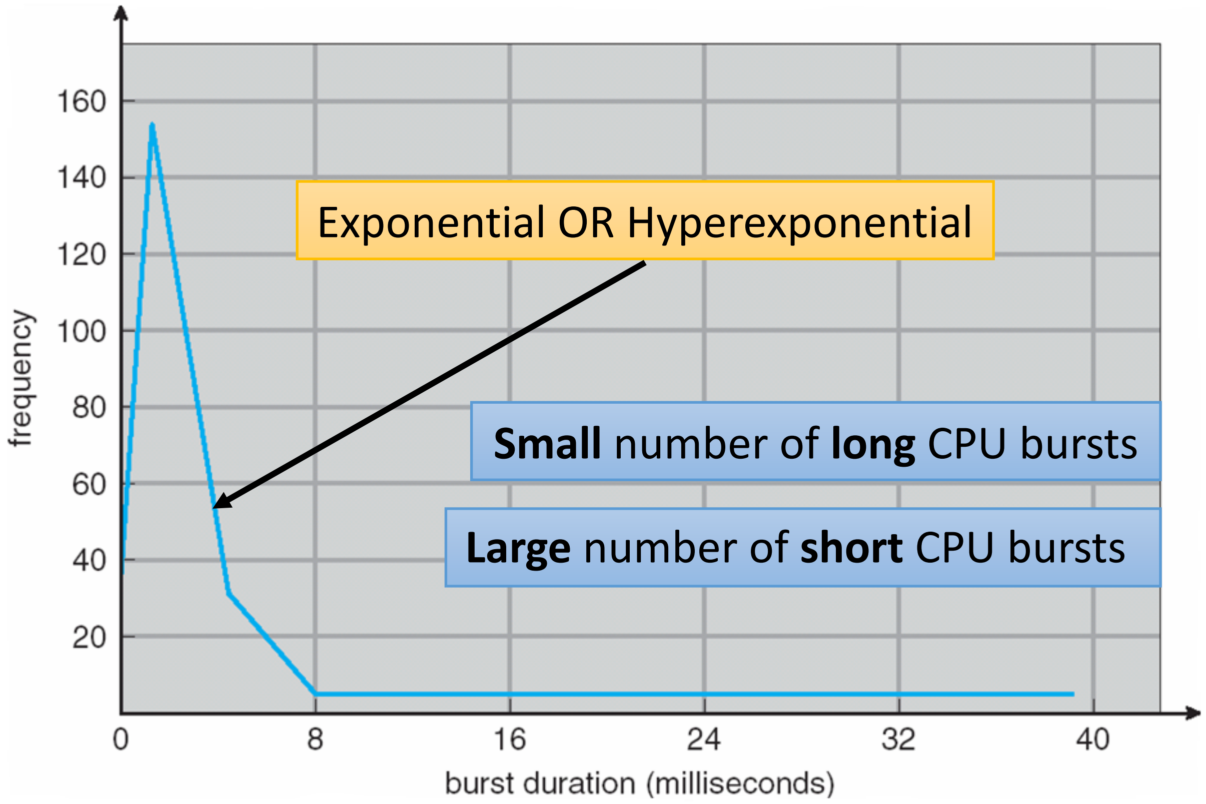
wait for I/O

CPU burst distribution is of main concern

CPU burst distribution is of **main concern** to select appropriate **CPU-scheduling algorithm**

CPU-Bound Process
spends more time doing computations; few very long CPU bursts

I/O-Bound Process
spends more time doing I/O than computations (*short CPU bursts*)



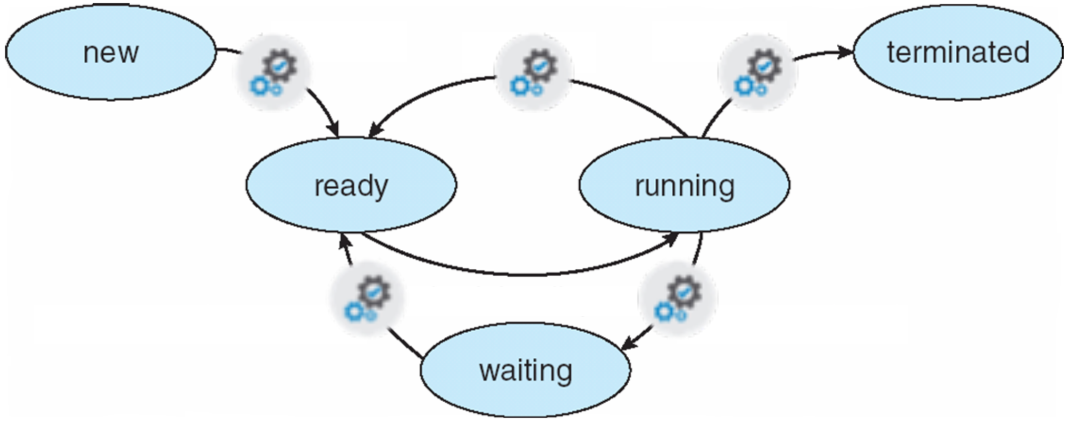
Exponential OR Hyperexponential

Small number of **long** CPU bursts

Large number of **short** CPU bursts

Preemptive Scheduling

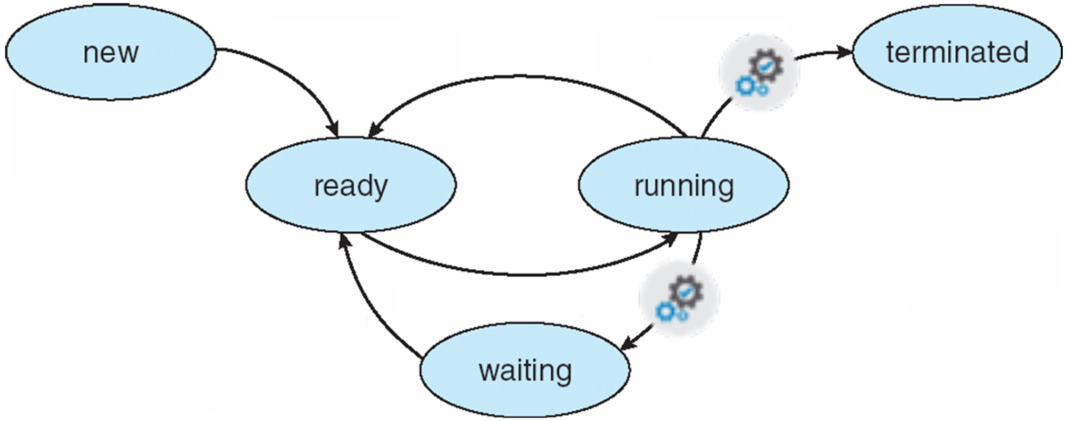
VS



The act of temporarily interrupting a process being allocated to the CPU, without requiring its cooperation, and with the intention of resuming the process at a later time.

Consider access to shared data
Consider preemption while in kernel mode
Consider interrupts occurring during crucial OS activities

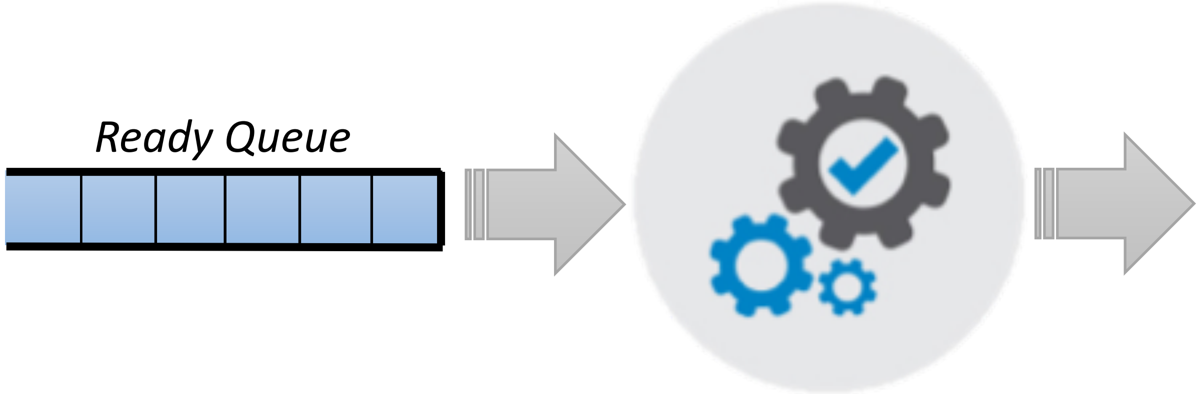
Nonpreemptive Scheduling

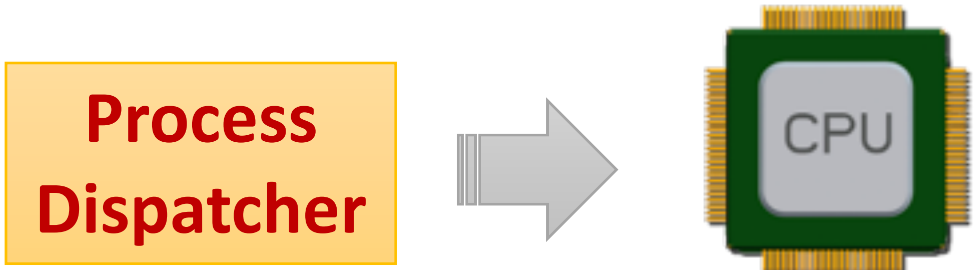


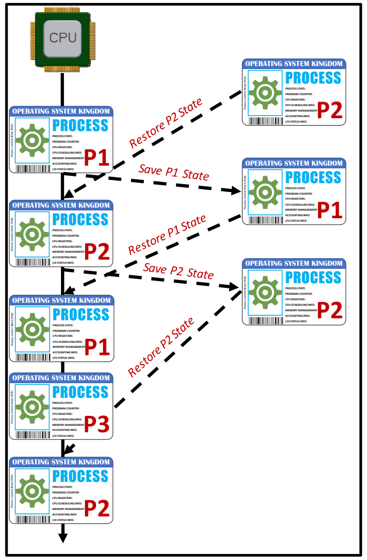
Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state

Process Dispatcher

Gives control of the CPU to the process selected by the short-term scheduler







The dispatcher should be as fast as possible, since it is invoked during every process switch

Dispatch latency – time it takes for the dispatcher to stop one process and start another running

- Receives control in Kernel Mode via interrupt.
- Context Switching
- Switching to User Mode
- Jumping to the proper location in the user program to restart that program

CPU Scheduling

There are many CPU scheduling algorithm, what are the criteria to compare among them?

CPU Utilization
keep the CPU as busy as possible

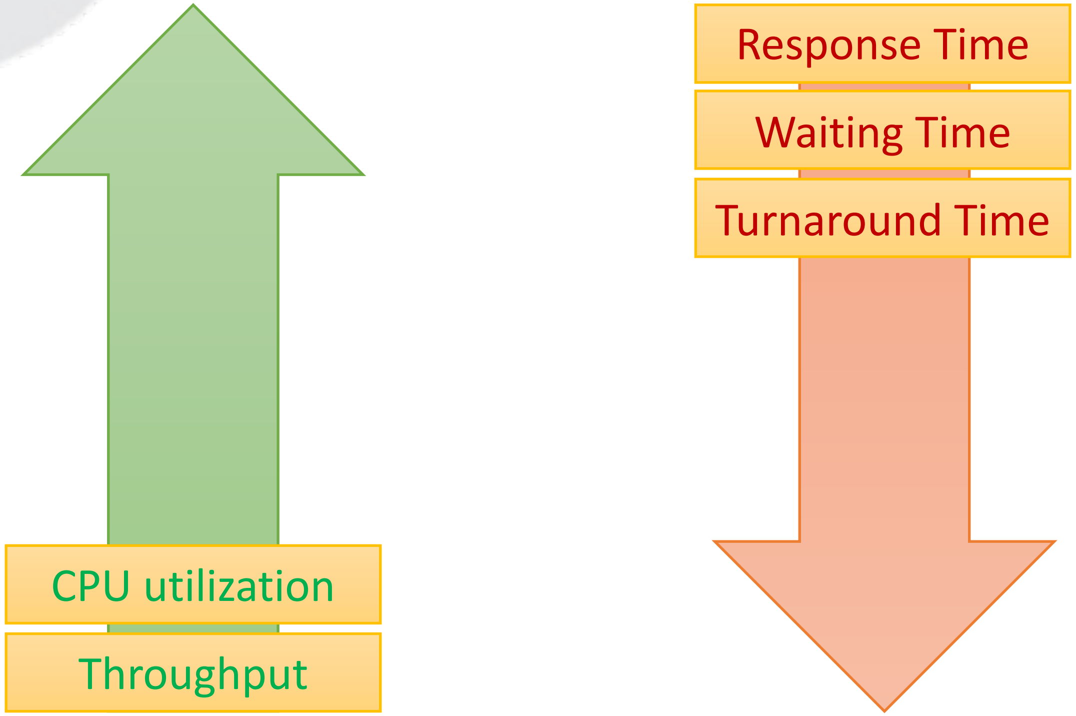
Throughput
Number of processes that complete their execution per time unit

Turnaround Time
amount of time to execute a particular process

Waiting Time
amount of time a process has been waiting in the ready queue

Response Time
amount of time it takes from when a request was submitted until the first response is produced (not output)

CPU Scheduling Optimization



CPU utilization

Throughput

Response Time

Waiting Time

Turnaround Time

FCFS

First Come, First Serve Scheduling

https://youtu.be/w9Uld56AsKE?t=11s

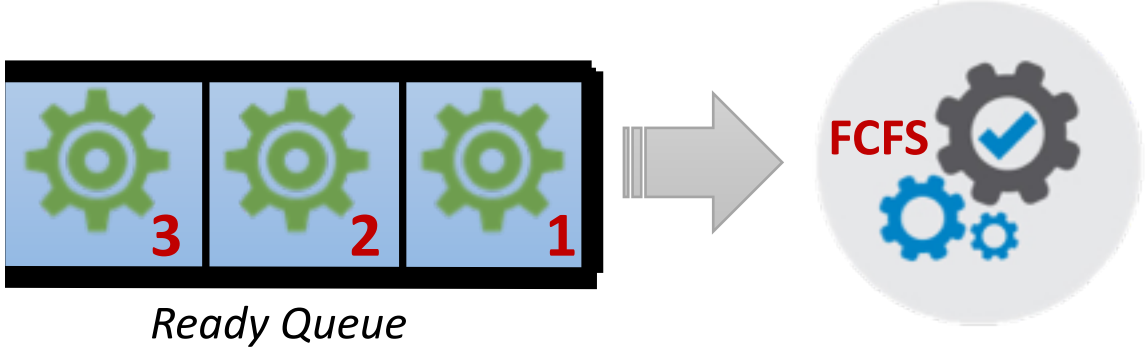
http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/fcfs.htm

FCFS

The process that requests the CPU first is allocated the CPU first

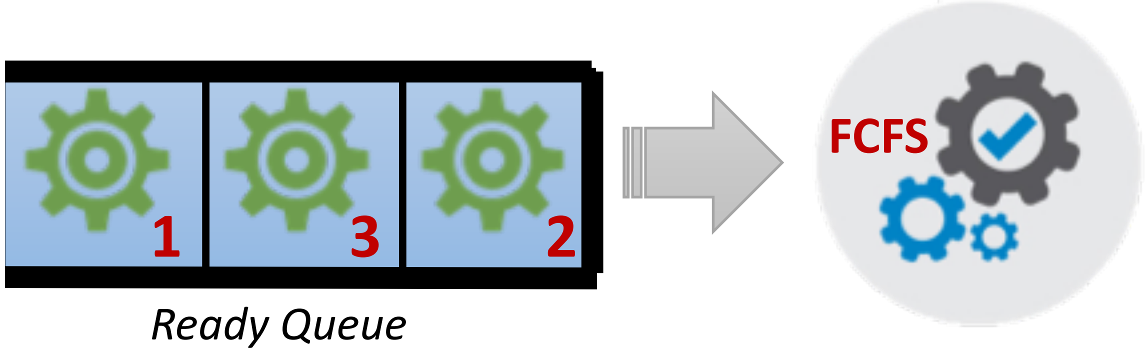
Waiting Time

amount of time a process has been waiting in the ready queue



Gantt Chart

⌛1 0 ⌛2 24 ⌛3 27 ⌛AVG 17



⌛1 6 ⌛2 0 ⌛3 3 ⌛AVG 3

FCFS

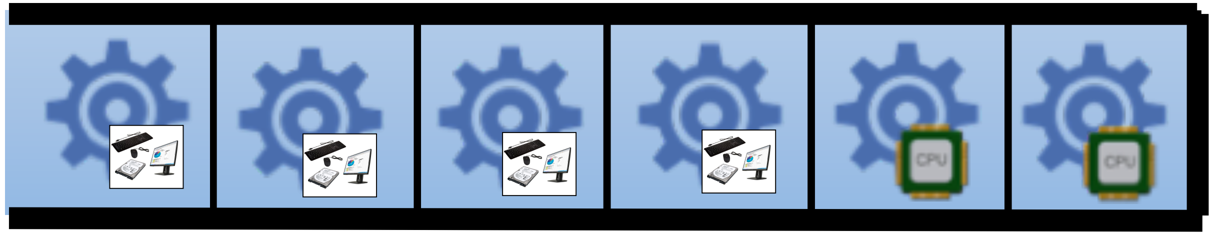
Nonpreemptive Scheduling

Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or requesting I/O

FCFS is troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Convoy Effect

Occurs when short process behind long process. All the short processes wait for the one big process to get off the CPU



Convoy Effect results in ***lower CPU and device utilization*** than might be possible if the shorter processes were allowed to go first.

Shortest Job First Scheduling

https://youtu.be/w9Uld56AsKE?t=38s

http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/sjf.htm

Shortest Job First Scheduling

The CPU is allocated to the process with the least CPU burst

Associate with each process the length of its next CPU burst

Use these lengths to schedule the process with the shortest time

Optimal by giving min. avg. waiting Time. The difficulty is knowing the length of the next CPU request

Preemptive SJF

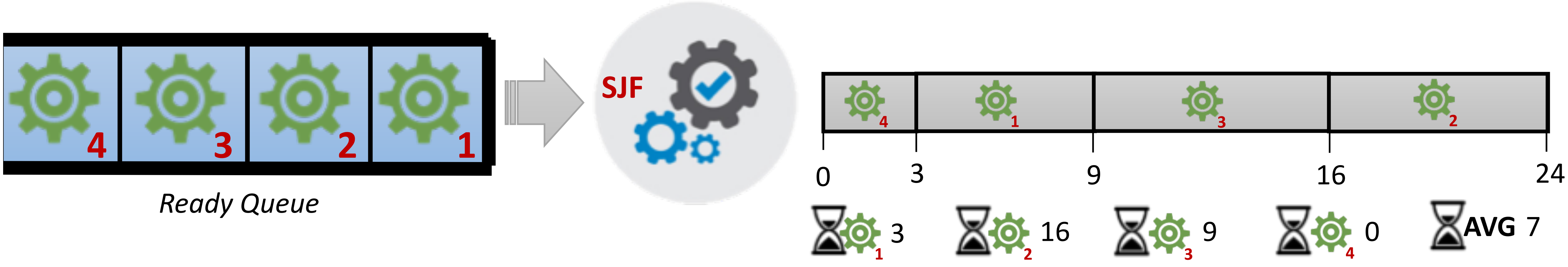
Interrupt the process being allocated to the CPU, if there is another process has arrived with lesser CPU time than the remaining CPU time for the running process

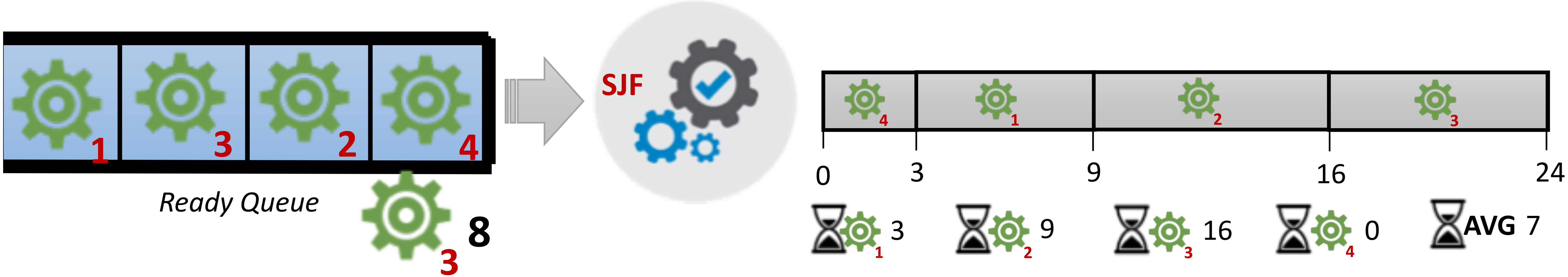
Nonpreemptive SJF

Once the CPU has been allocated to a process with the highest priority, the process keeps the CPU until it releases the CPU either by terminating or requesting I/O

SJF

The CPU is allocated to the process with the least CPU burst





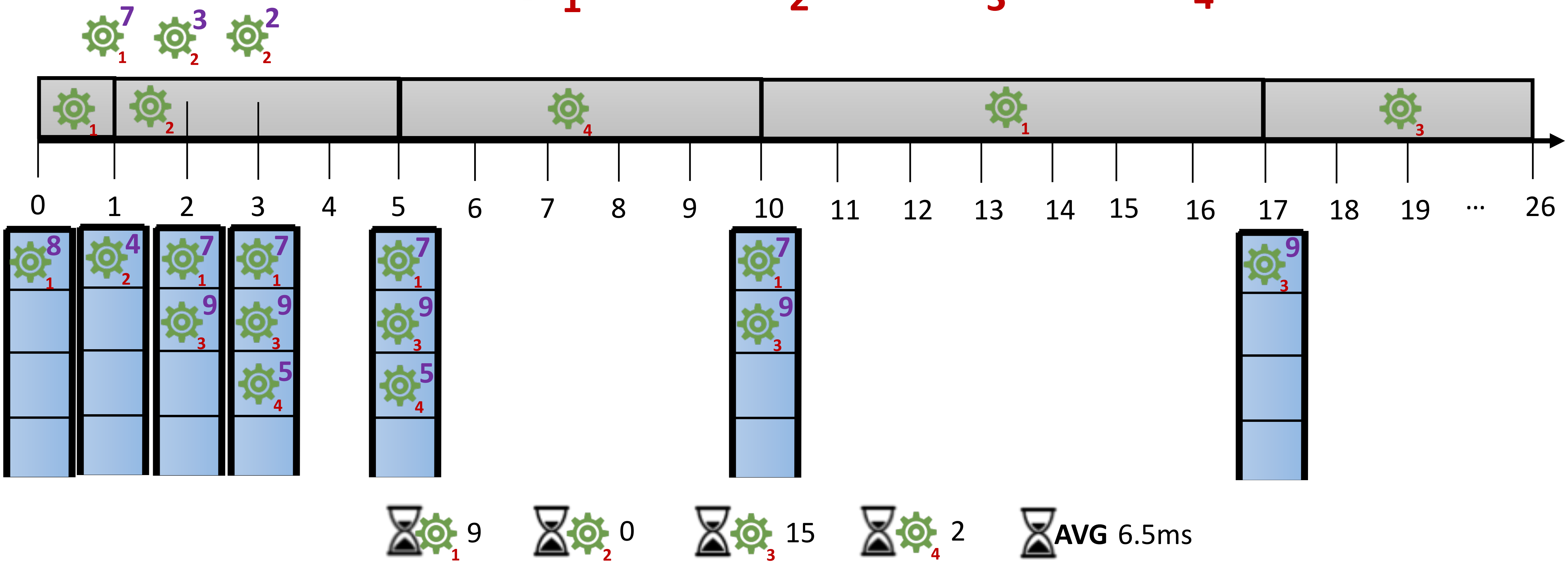
SJF

Preemptive

Shortest Remaining Time First

Preemptive SJF

Interrupt the process being allocated to the CPU, if there is another process has arrived with lesser CPU time than the remaining CPU time for the running process



Priority Scheduling

https://www.youtube.com/watch?v=rcOBx752m-Q

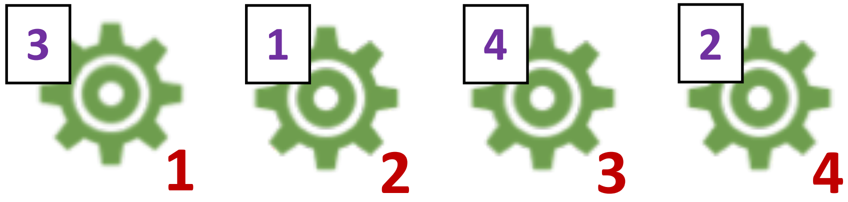
http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/priority.htm

Priority Scheduling

The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)

SJF is a priority scheduling where priority is the inverse of predicted next CPU burst time

A priority number (integer) is associated with each process



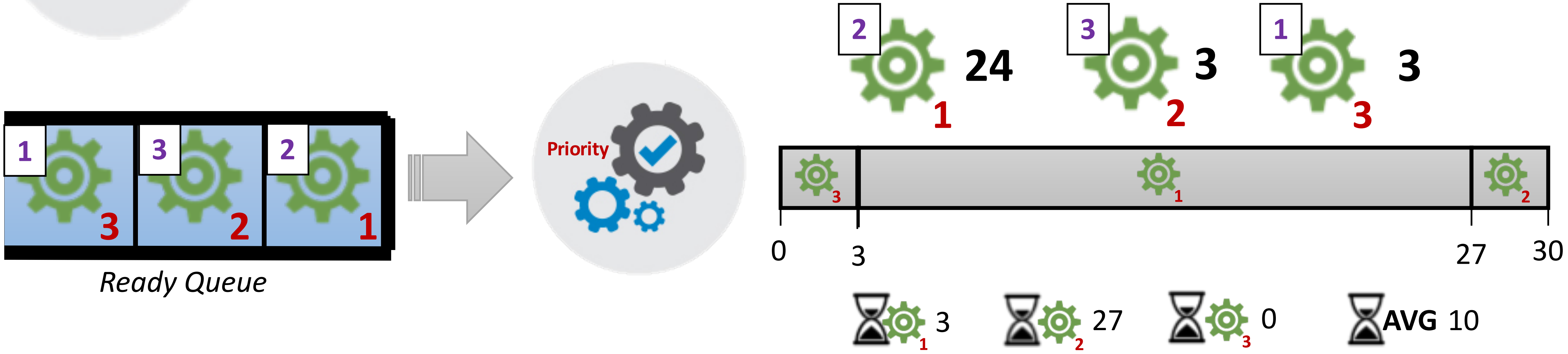
Preemptive Priority

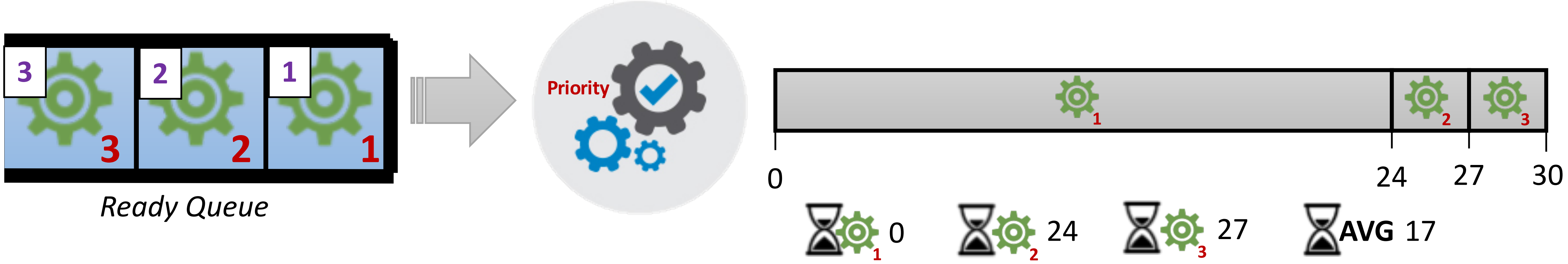
Interrupt the process being allocated to the CPU, if there is another process has arrived with higher priority than the priority of the running process

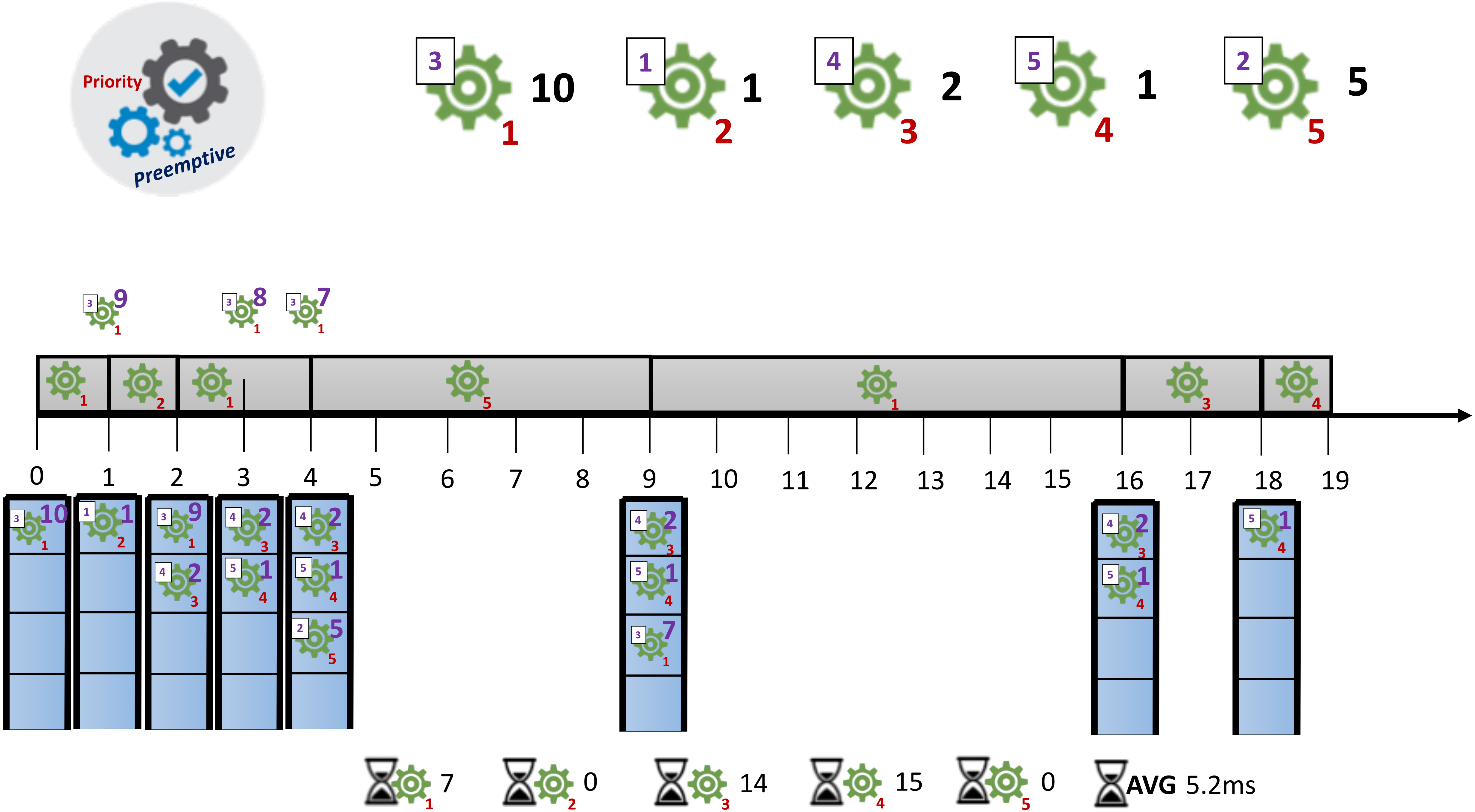
Nonpreemptive Priority

Once the CPU has been allocated to a process with the highest priority, the process keeps the CPU until it releases the CPU either by terminating or requesting I/O

The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)






Priority
Preemptive
3 10 1
1 1 2
4 2 3
5 1 4
2 5 5
3 9 1
3 8 1
3 7 1
1
2
1
5
1
3
4
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
3 10 1
1 1 2
3 9 1
4 2 3
4 2 3
5 1 4
4 2 3
5 1 4
2 5 5
4 2 3
5 1 4
3 7 1
4 2 3
5 1 4
5 1 4
1 7
2 0
3 14
4 15
5 0
AVG 5.2ms

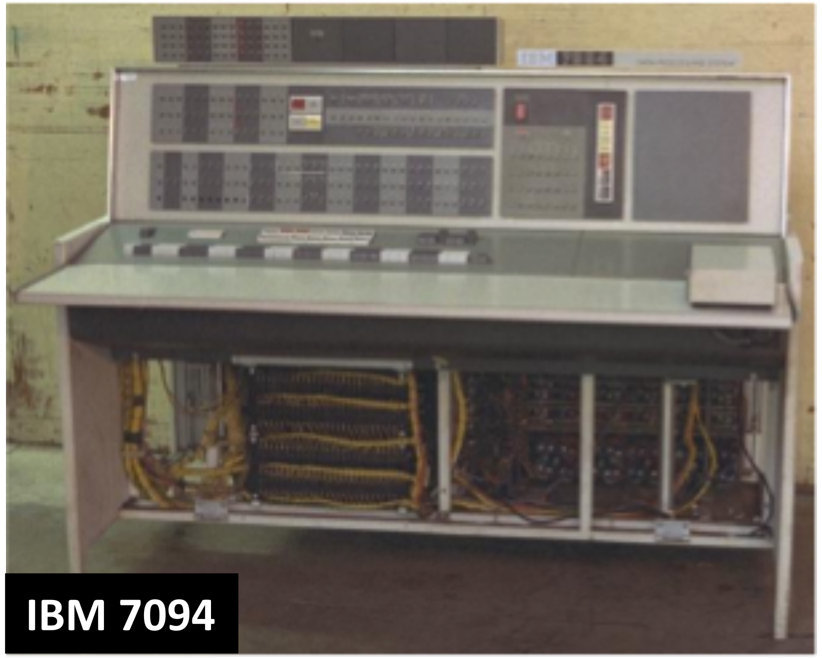
Priority Scheduling

The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)

Problem Starvation

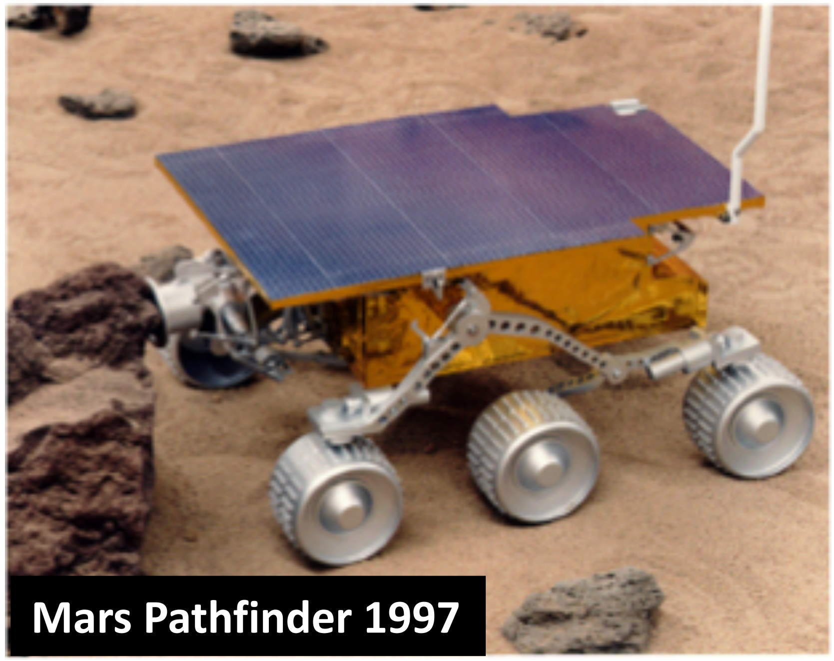
low priority processes may never execute

Aging – as time progresses increase the priority of the process



IBM 7094

When they shut it down in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run



Mars Pathfinder 1997

http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

Round Robin Scheduling

https://youtu.be/w9UId56AsKE?t=1m31s

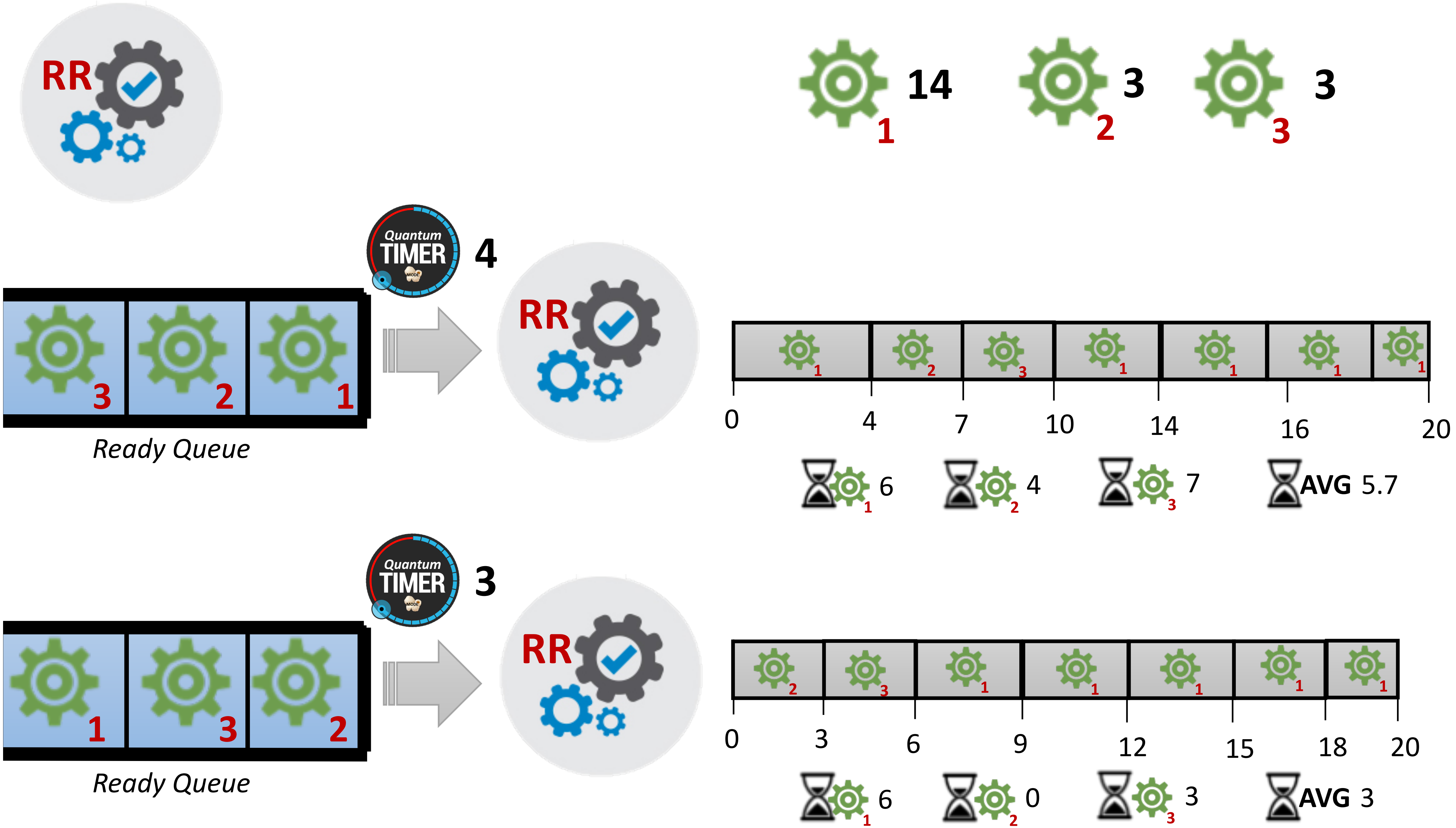
http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/rr.htm

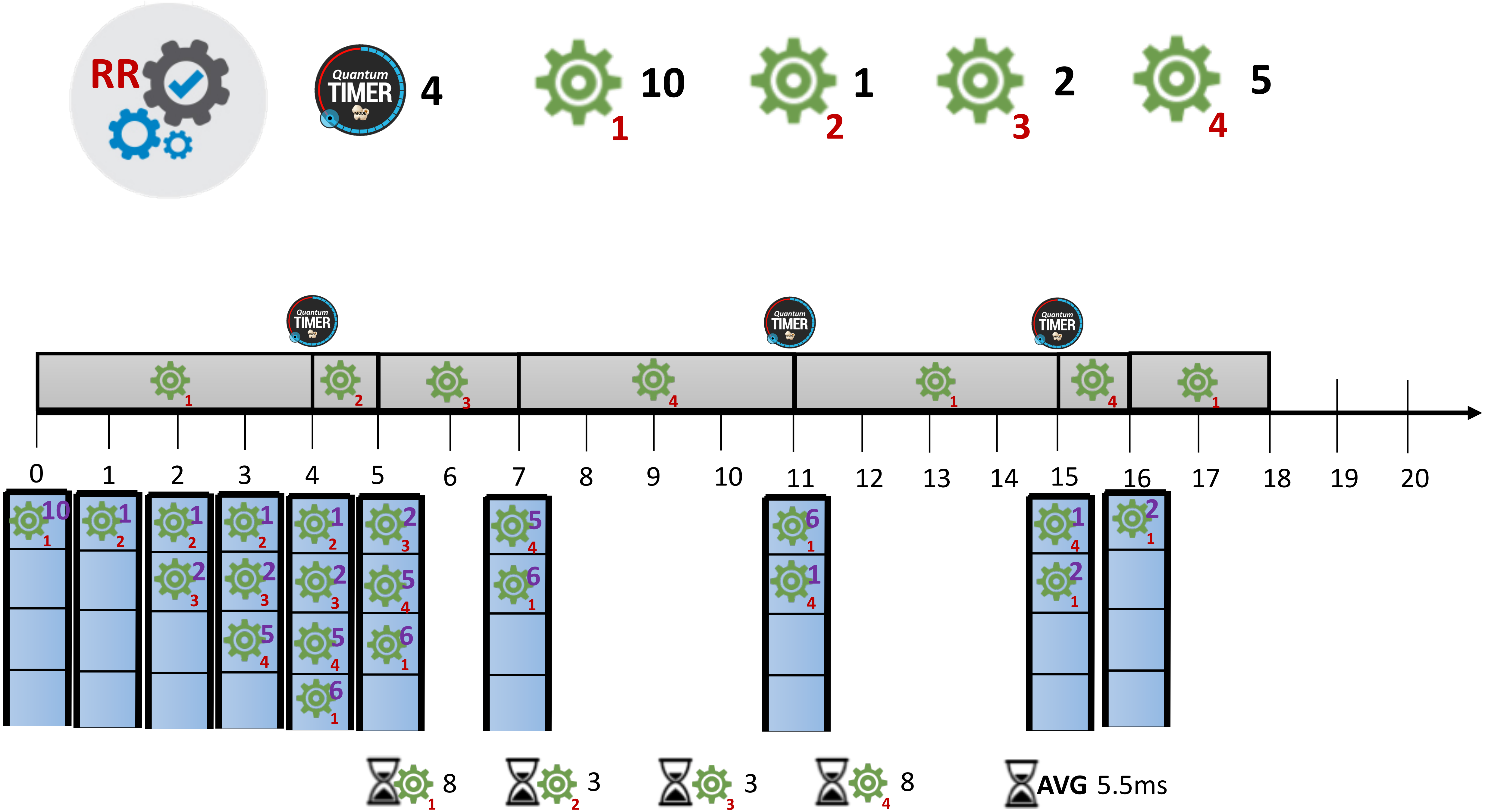
Round Robin Scheduling

Typically, higher average ***turnaround*** *than SJF, but better response*

Each process gets a **quantum time** (usually 10-100 milliseconds) of CPU time. After this time has elapsed, the process is preempted and added to the end of the ready queue

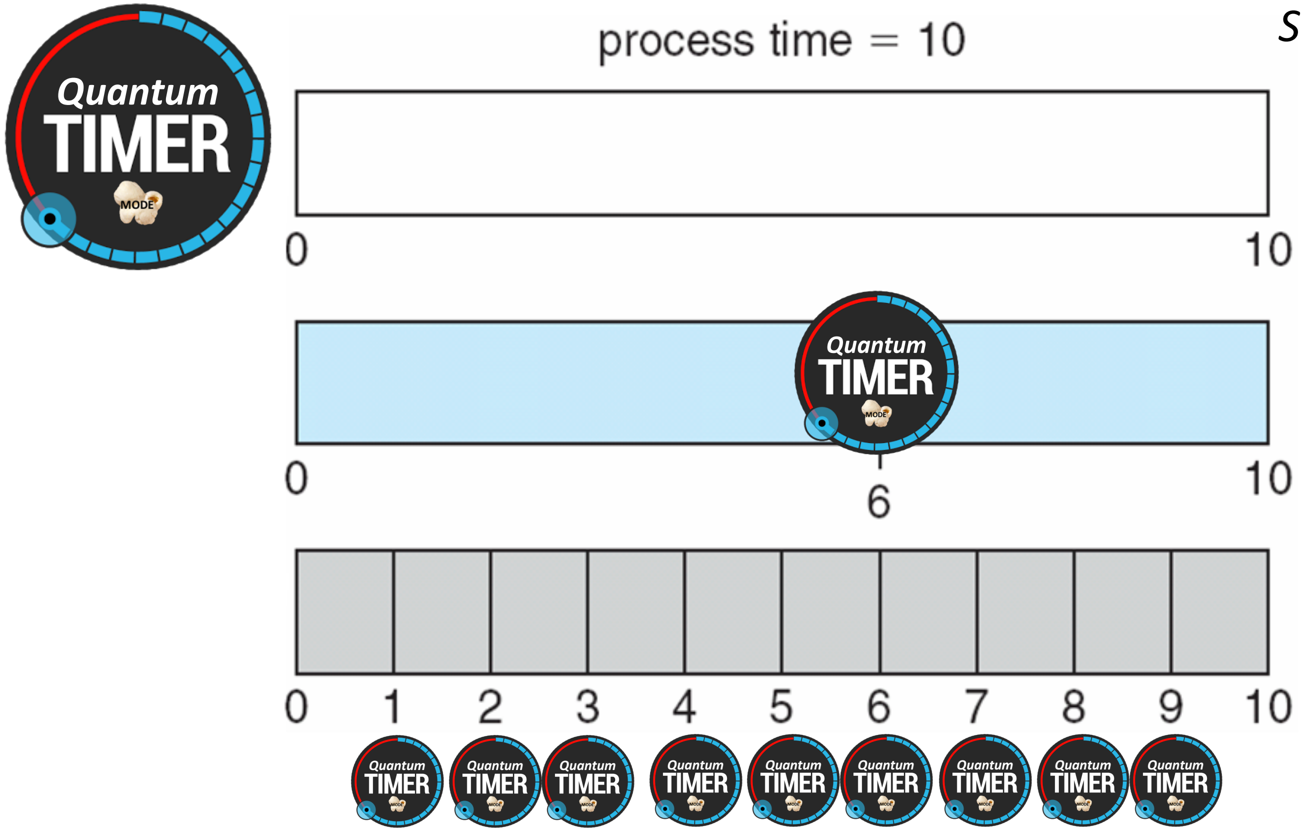
Timer interrupts every quantum to schedule next process


RR
1
14
2
3
3
3
Quantum
TIMER
4
3
2
1
Ready Queue
RR
1
2
3
1
1
1
1
0
4
7
10
14
16
20
1
6
2
4
3
7
AVG 5.7
Quantum
TIMER
3
1
3
2
Ready Queue
RR
2
3
1
1
1
1
1
0
3
6
9
12
15
18
20
1
6
2
0
3
3
AVG 3



RR
Quantum TIMER 4
1 10
2 1
3 2
4 5
1 2 3 4 1 4 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
10 1
1 2
1 2 2 3
1 2 2 3 5 4
1 2 2 3 5 4 6 1
2 3 5 4 6 1
5 4 6 1
6 1 1 4
1 4 2 1
2 1
1 8
2 3
3 3
4 8
AVG 5.5ms

Quantum Time and Context Switching Time



Context Switching
0
1
9

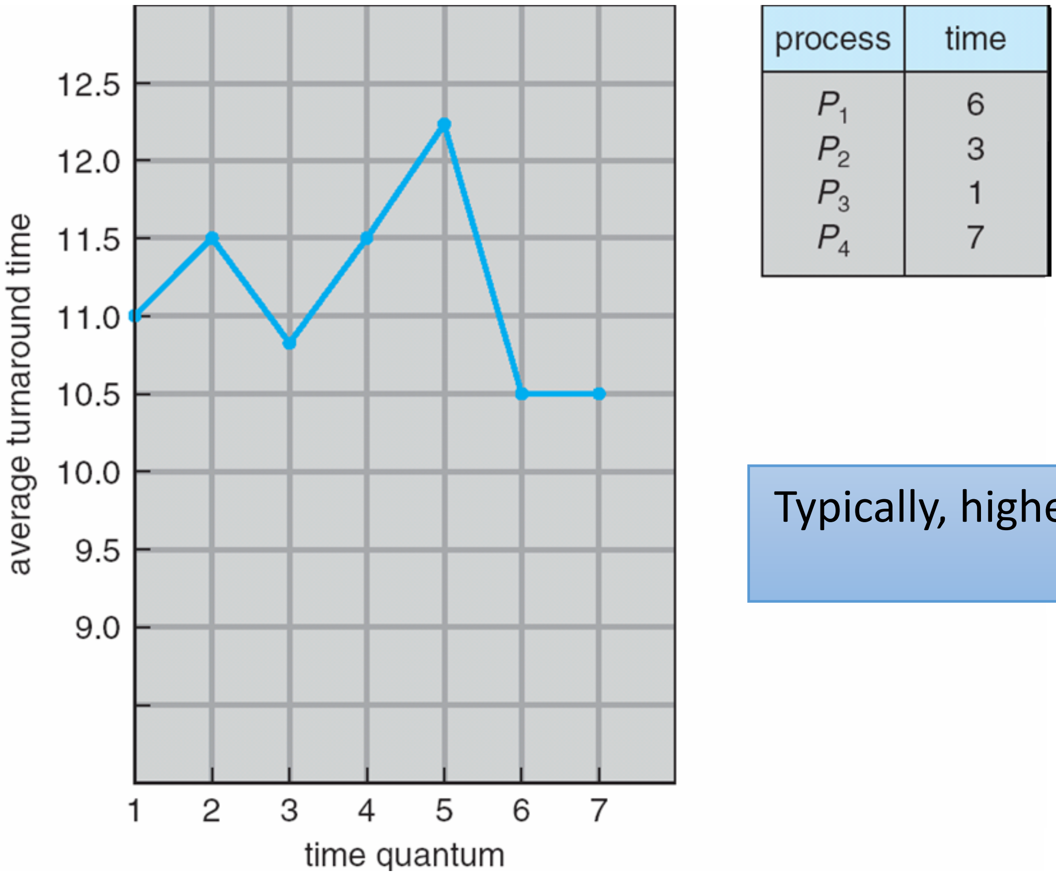
Time quantum should be large compared with the context switch time, it should not be too large.

If the time quantum is too large, RR scheduling degenerates to an FCFS scheduling

If the time quantum is too large, RR scheduling suffer from high overhead due to context switching

Quantum Time is usually 10ms to 100ms, Context Switching Time < 10 μsec

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Turnaround Time
amount of time to execute a particular process

Response Time
amount of time it takes from when a request was submitted until the first response is produced (*not output*)

Typically, higher average turnaround than SJF, but better response

A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum

Queue Scheduling

Representation of Process Scheduling

Medium-Term Scheduling

swaps out process from memory, then ***swaps*** it ***in*** ready queue

process swapping scheduler

Reduces the degree of multiprogramming

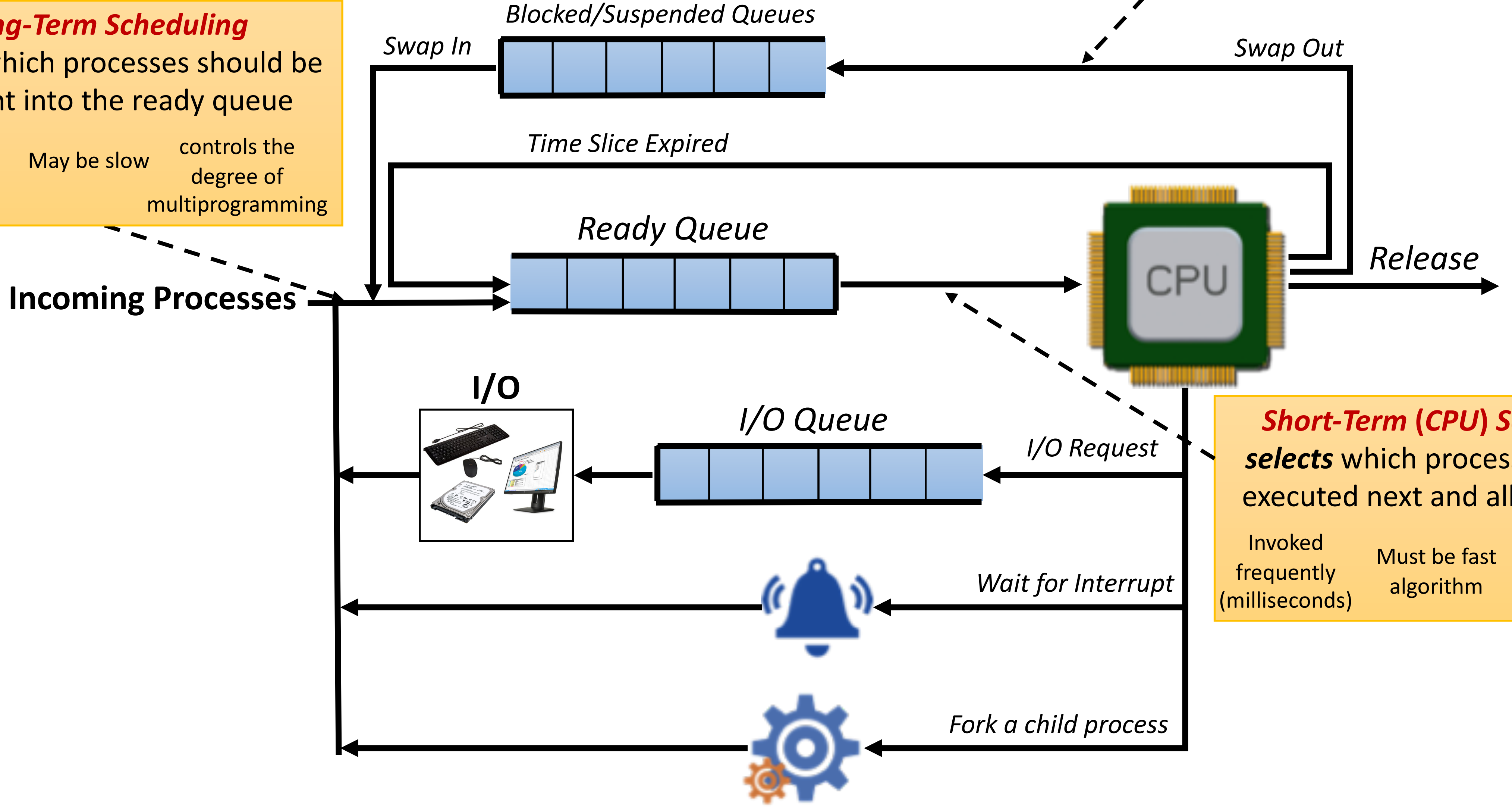
Long-Term Scheduling

selects which processes should be brought into the ready queue

Invoked **in**frequently (secs, mins)

May be slow

controls the degree of multiprogramming



Short-Term (CPU) Scheduling

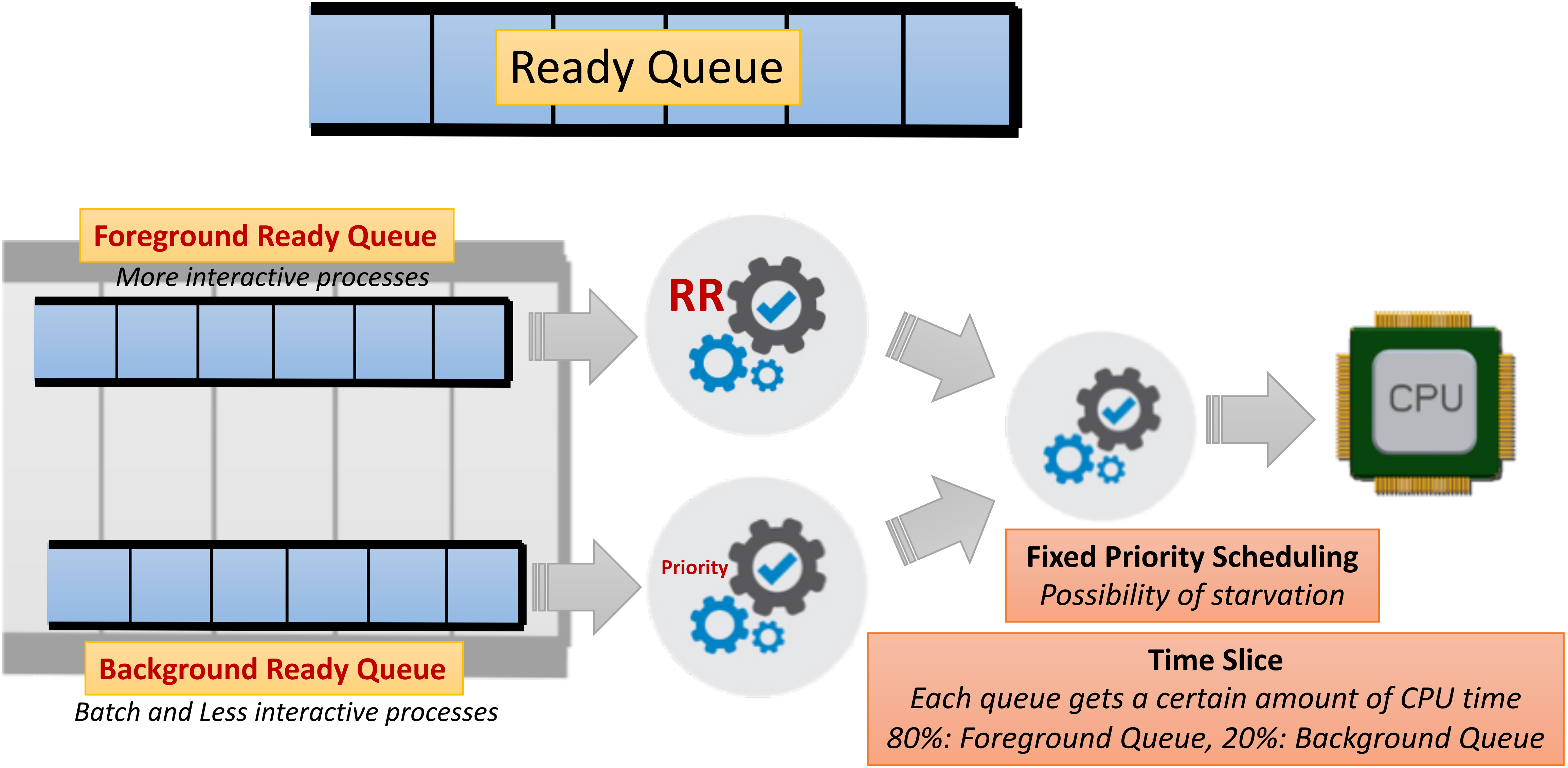
selects which process should be executed next and allocates CPU

Invoked frequently (milliseconds)

Must be fast algorithm

Sometimes the only scheduler in a system

Multilevel Ready Queue



Ready Queue

Foreground Ready Queue

More interactive processes

Background Ready Queue

Batch and Less interactive processes

Fixed Priority Scheduling

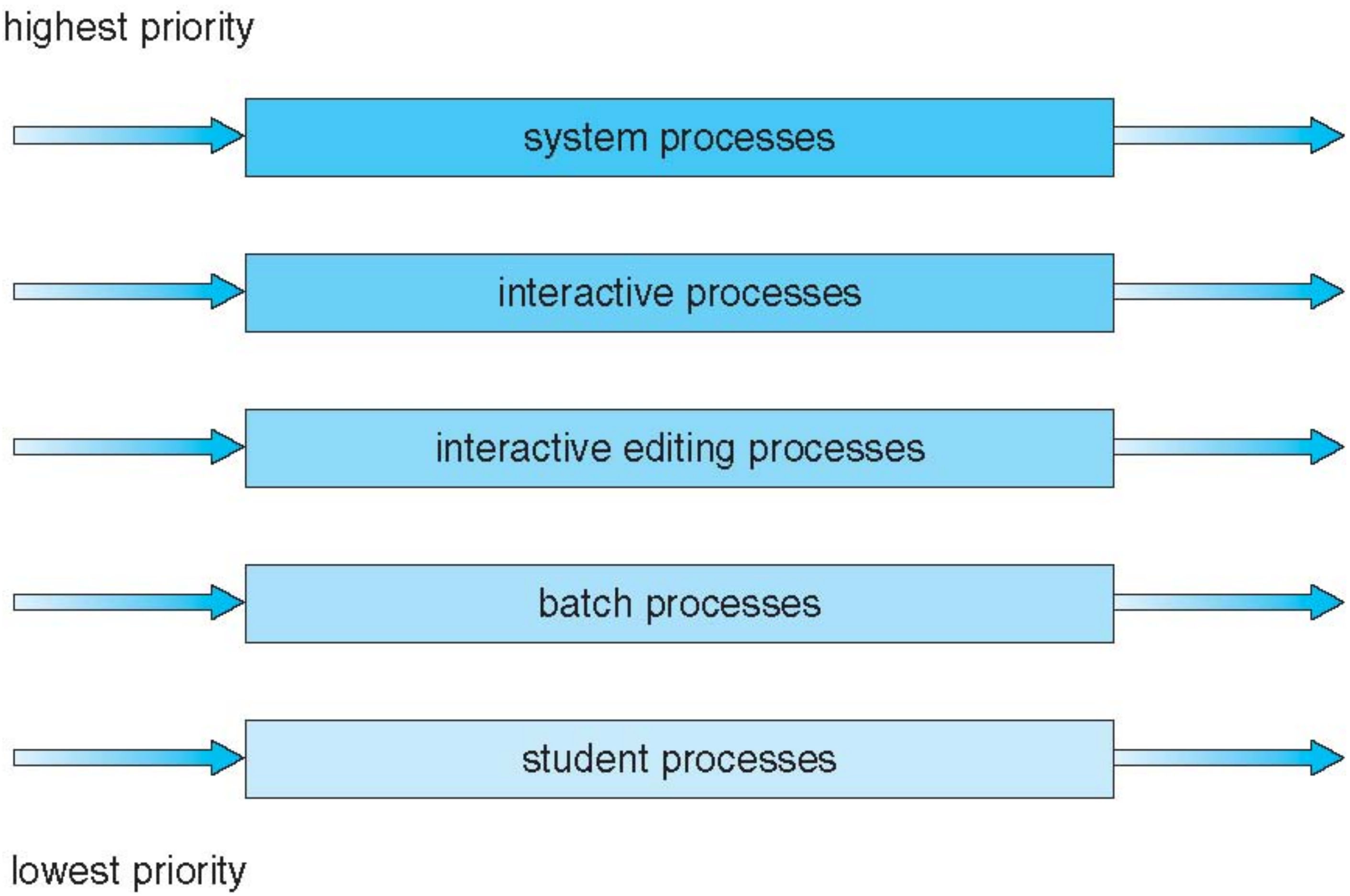
Possibility of starvation

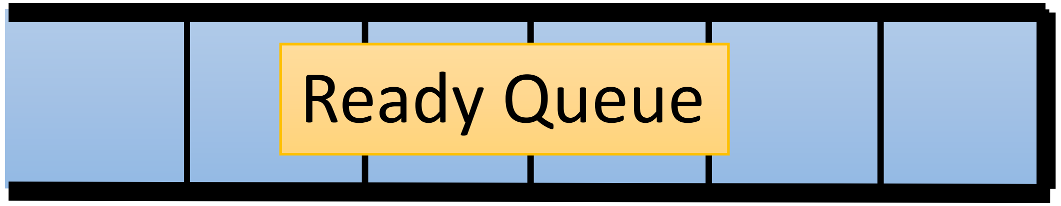
Time Slice

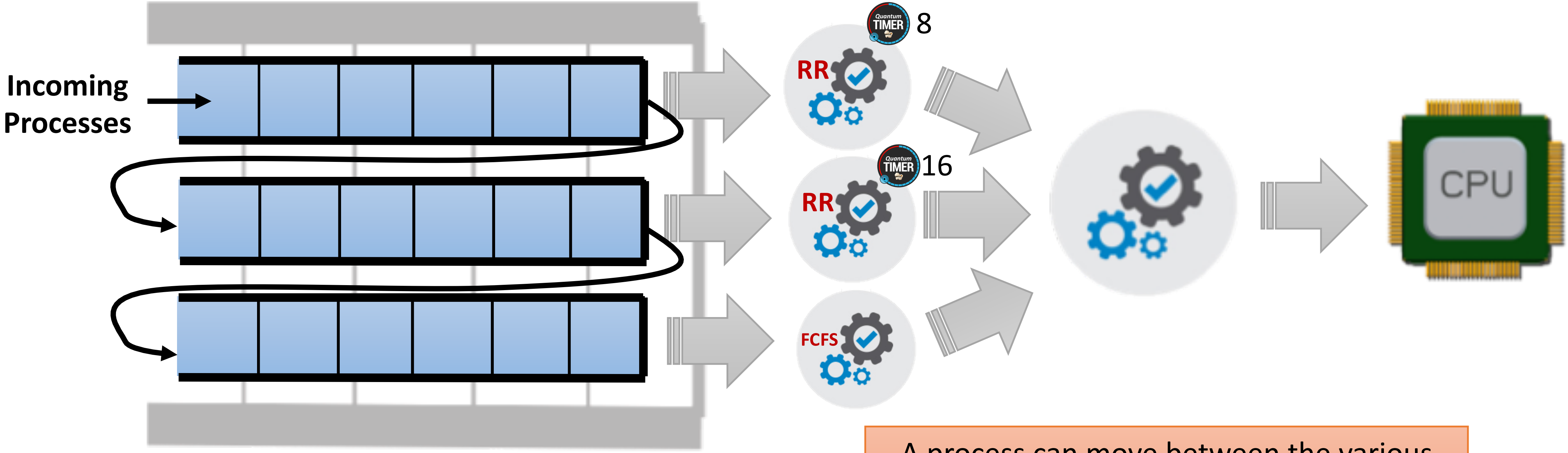
Each queue gets a certain amount of CPU time

80%: Foreground Queue, 20%: Background Queue

Multilevel Queue Scheduling







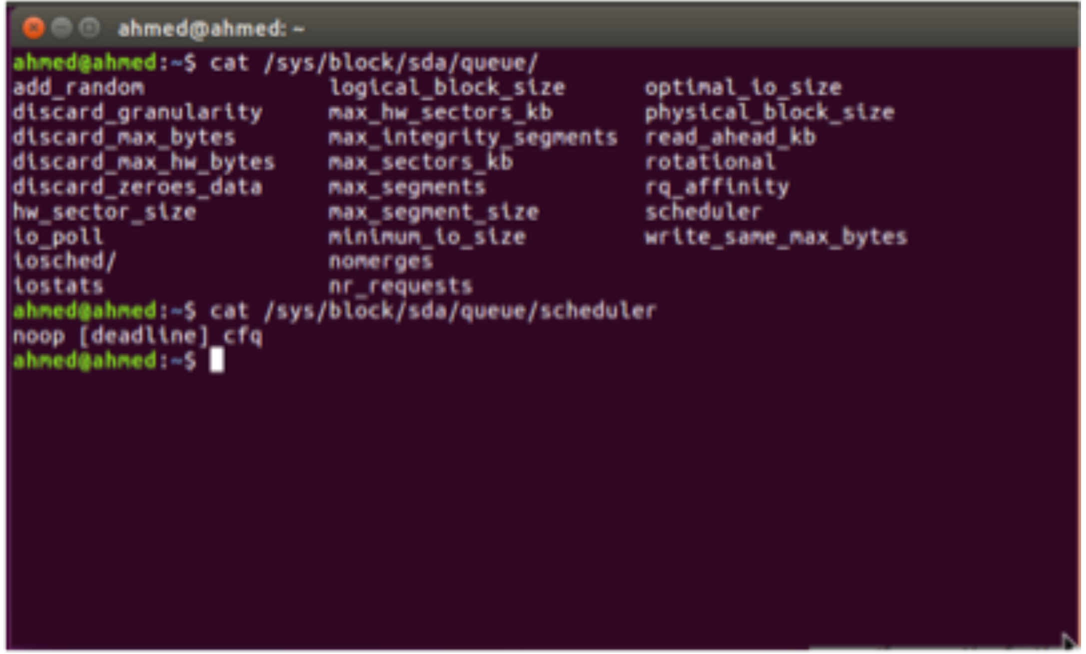
Multilevel Feedback Queue

A process can move between the various queues; aging can be implemented this way

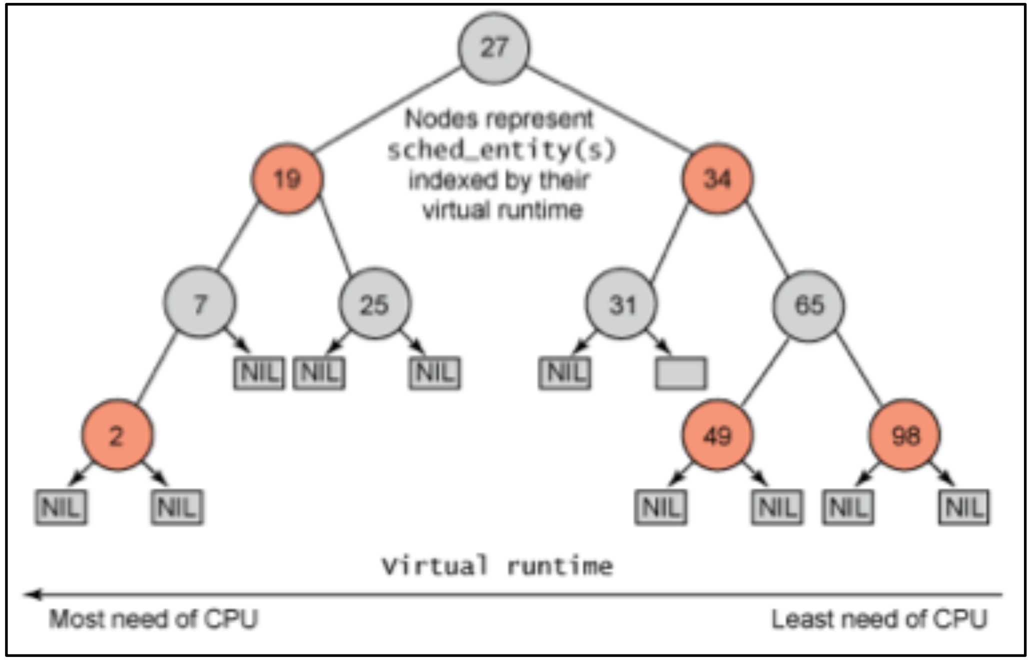
Completely Fair Scheduler

https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c
https://github.com/torvalds/linux/blob/master/include/linux/init_task.h#L200



```
ahmed@ahmed:~$ cat /sys/block/sda/queue/
add_random               logical_block_size       optimal_io_size
discard_granularity      max_hw_sectors_kb        physical_block_size
discard_max_bytes        max_integrity_segments   read_ahead_kb
discard_max_hw_bytes     max_sectors_kb           rotational
discard_zeroes_data      max_segments             rq_affinity
hw_sector_size           max_segment_size         scheduler
io_poll                  minimum_io_size          write_same_max_bytes
iosched/                 nomerges
iostats                  nr_requests
ahmed@ahmed:~$ cat /sys/block/sda/queue/scheduler
noop [deadline] cfq
ahmed@ahmed:~$ 
```



https://github.com/torvalds/linux/blob/master/include/uapi/linux/sched.h

http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/

つづく