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

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

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

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

```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`

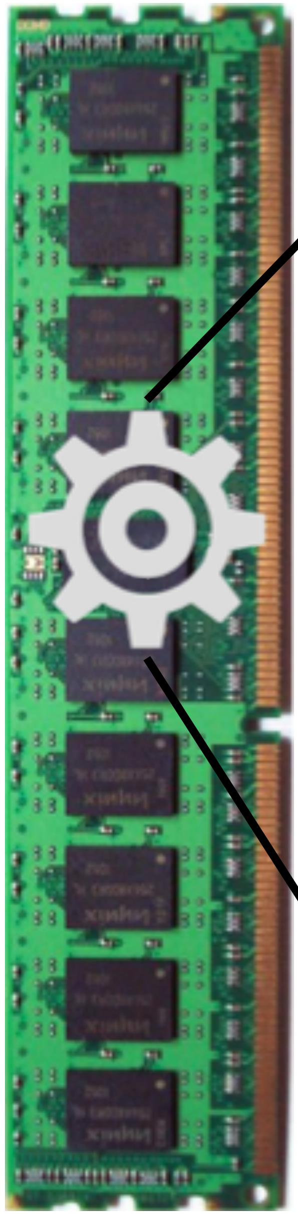
```
Disassembly of section __TEXT,__text:
__text:
100000f50:      55      pushq   %rbp
100000f51:      48 89 e5        movq    %rsp, %rbp
100000f54:      48 83 ec 10     subq    $16, %rsp
100000f58:      48 8d 3d 3b 00 00 00    leaq    59(%rip), %rdi
100000f5f:      c7 45 fc 00 00 00 00    movl    $0, -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

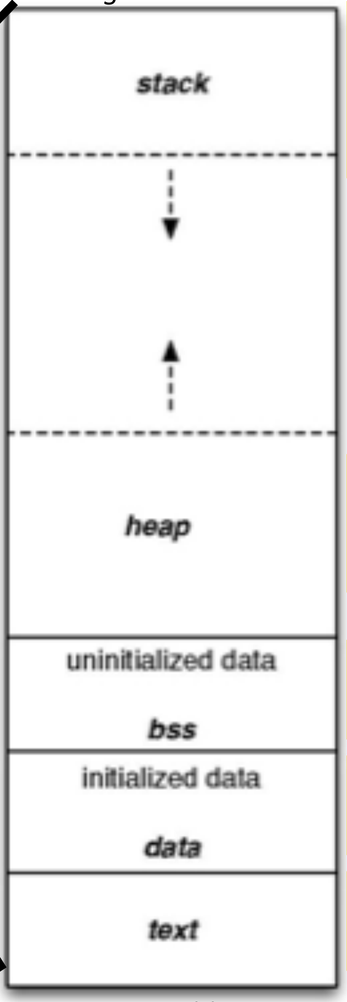
https://jvns.ca/blog/2014/09/06/how-to-read-an-executable/

Process Memory Layout

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



Higher Address



Lower Address

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/

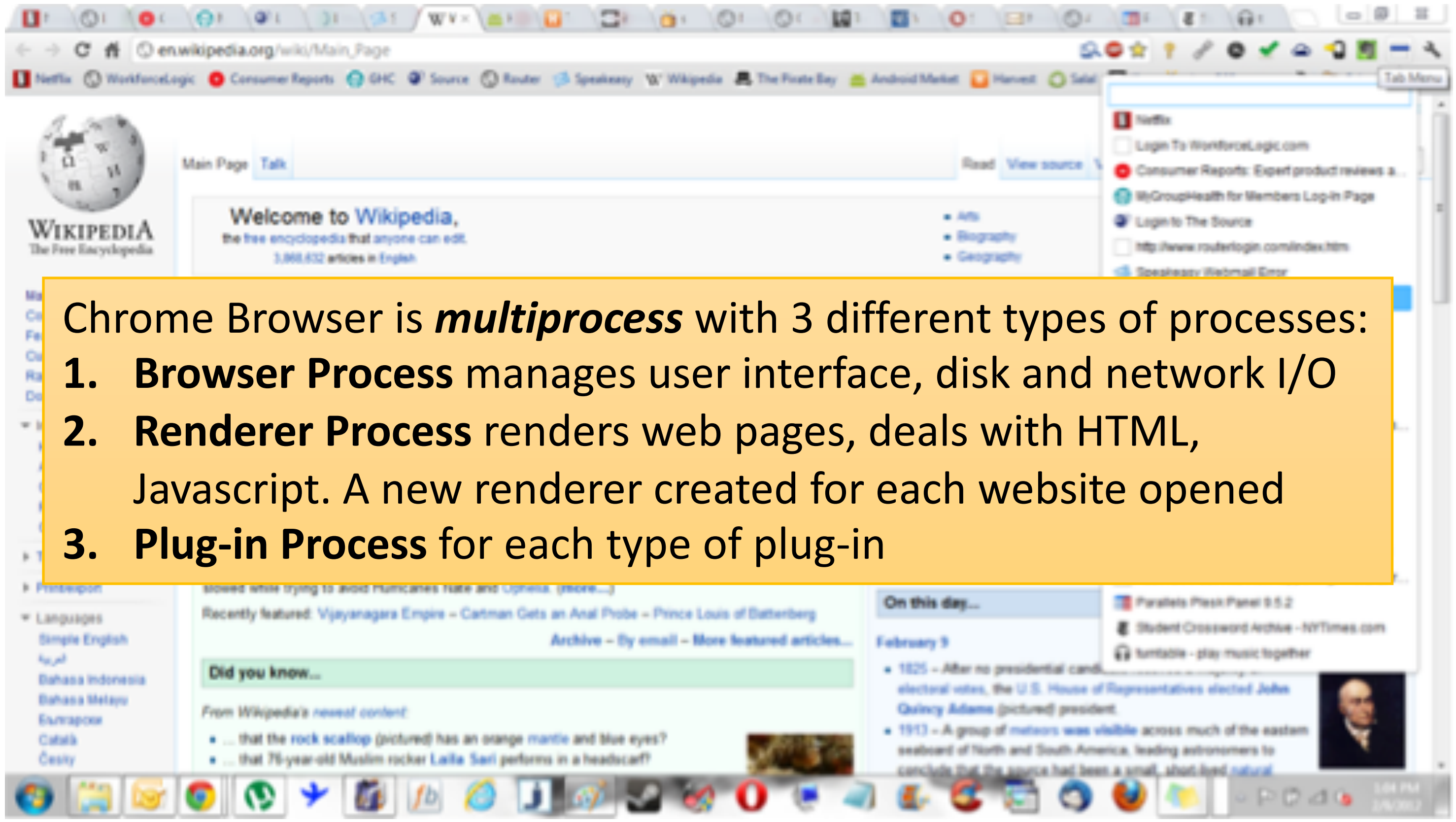
```c
#include <stdio.h>
int global = 10;
int main(void) {
    static int i = 100;
    return 0;
}
```

```
$ 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;
}
```

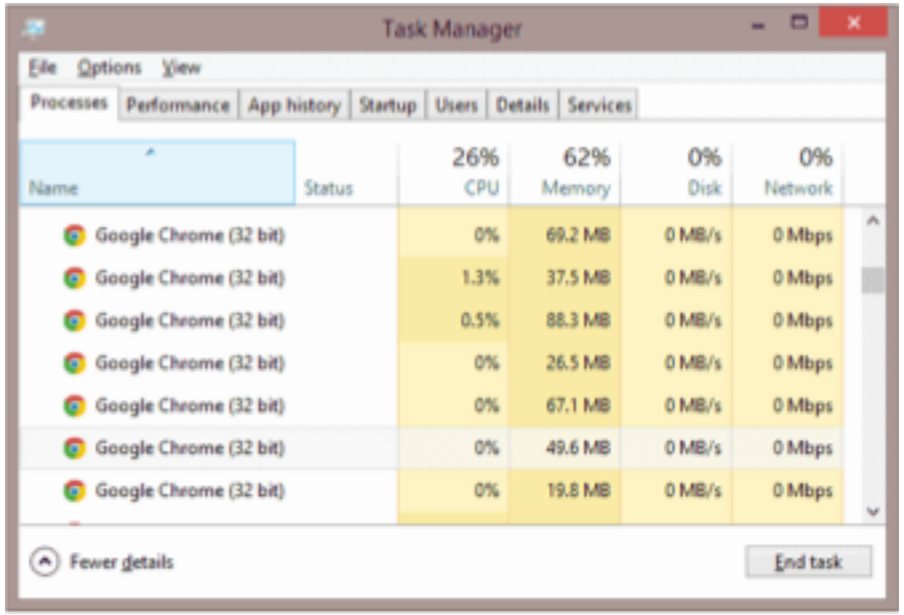
```
$ gcc memory-layout.c -o memory-layout
$ size memory-layout
text     data     bss     dec     hex     filename
960      248      8       1216    4c0     memory-layout
```

One program can be several processes

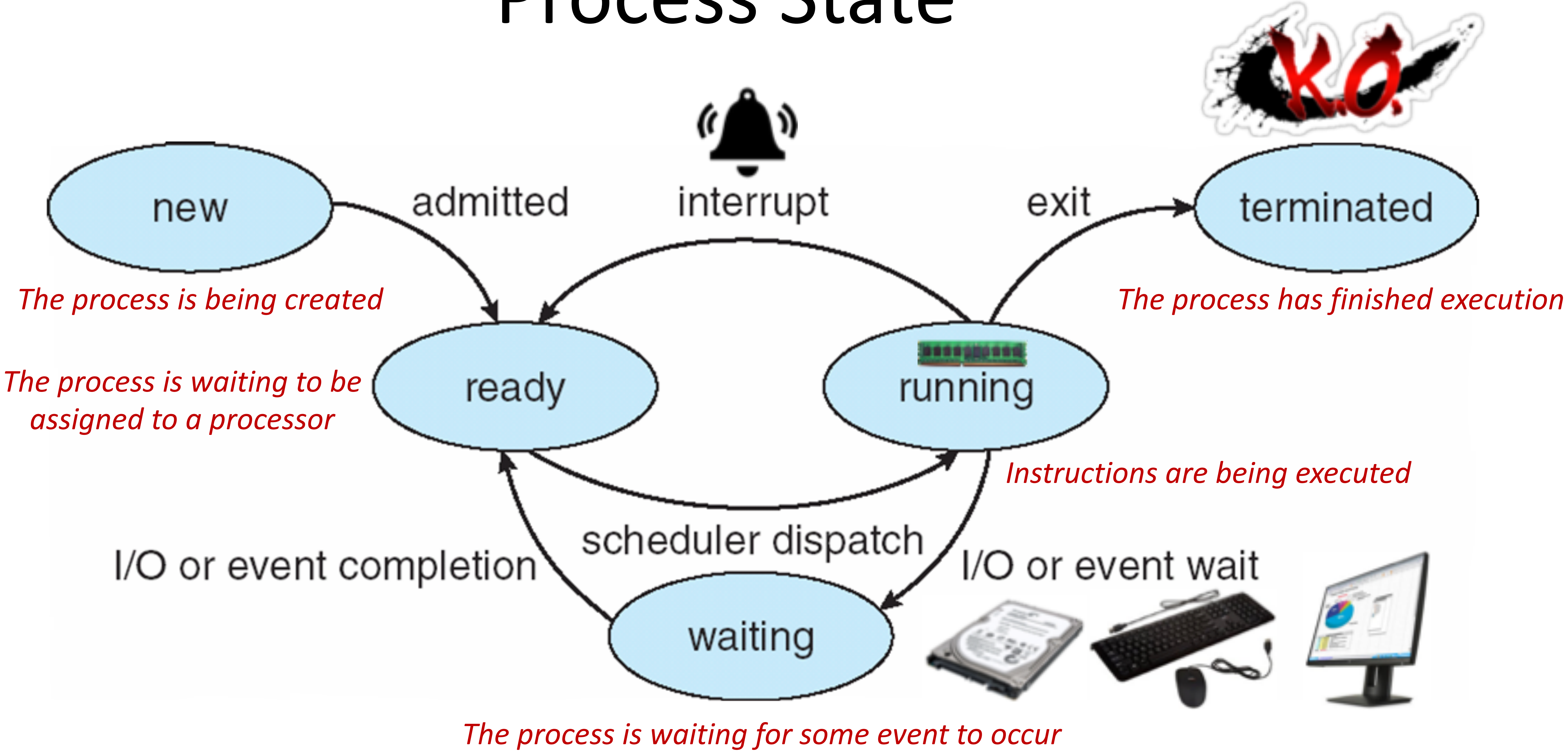


Chrome Browser is ***multiprocess*** with 3 different types of processes:

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



Process State




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:

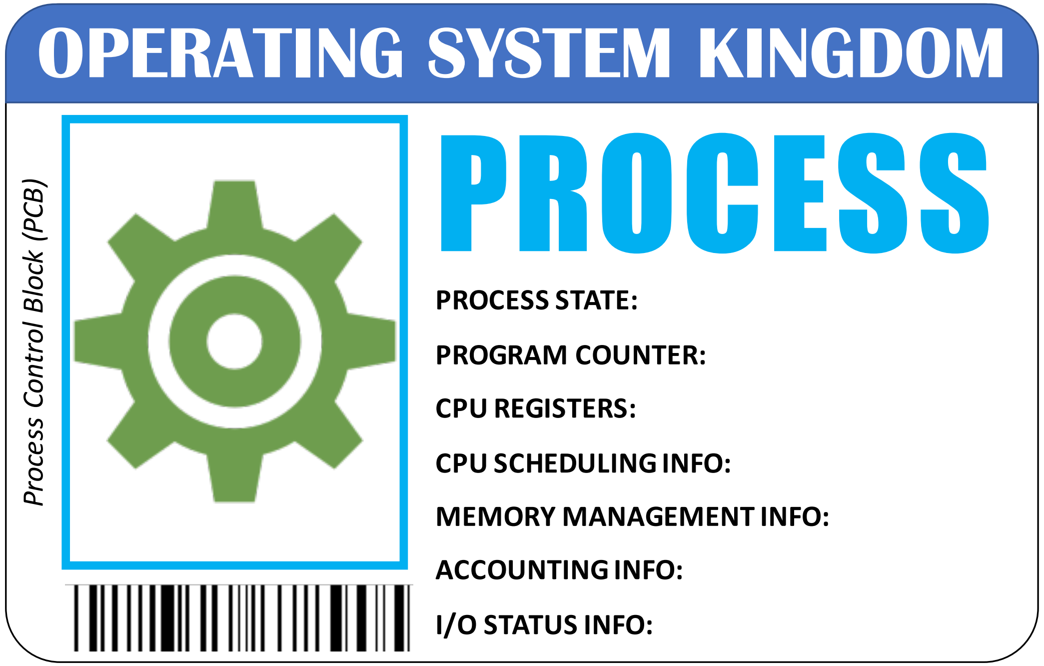
CPU REGISTERS:

CPU SCHEDULING INFO:

MEMORY MANAGEMENT INFO:

ACCOUNTING INFO:

I/O STATUS INFO:



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.

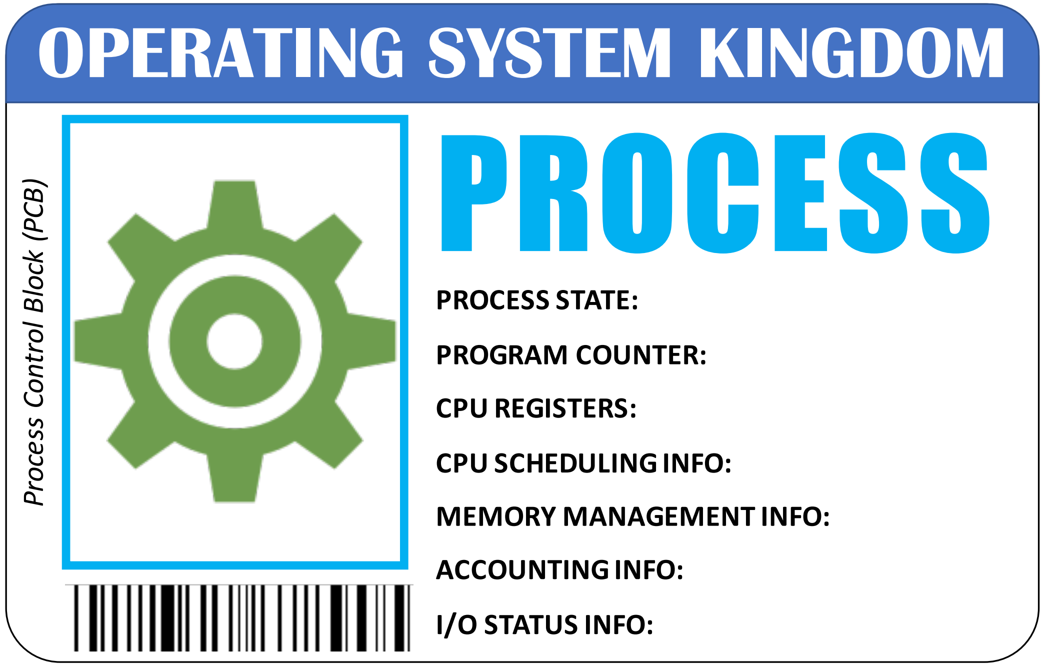
CPU Scheduling Info: Priorities, scheduling queue pointers and other scheduling parameters (*Chapter 6*)

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

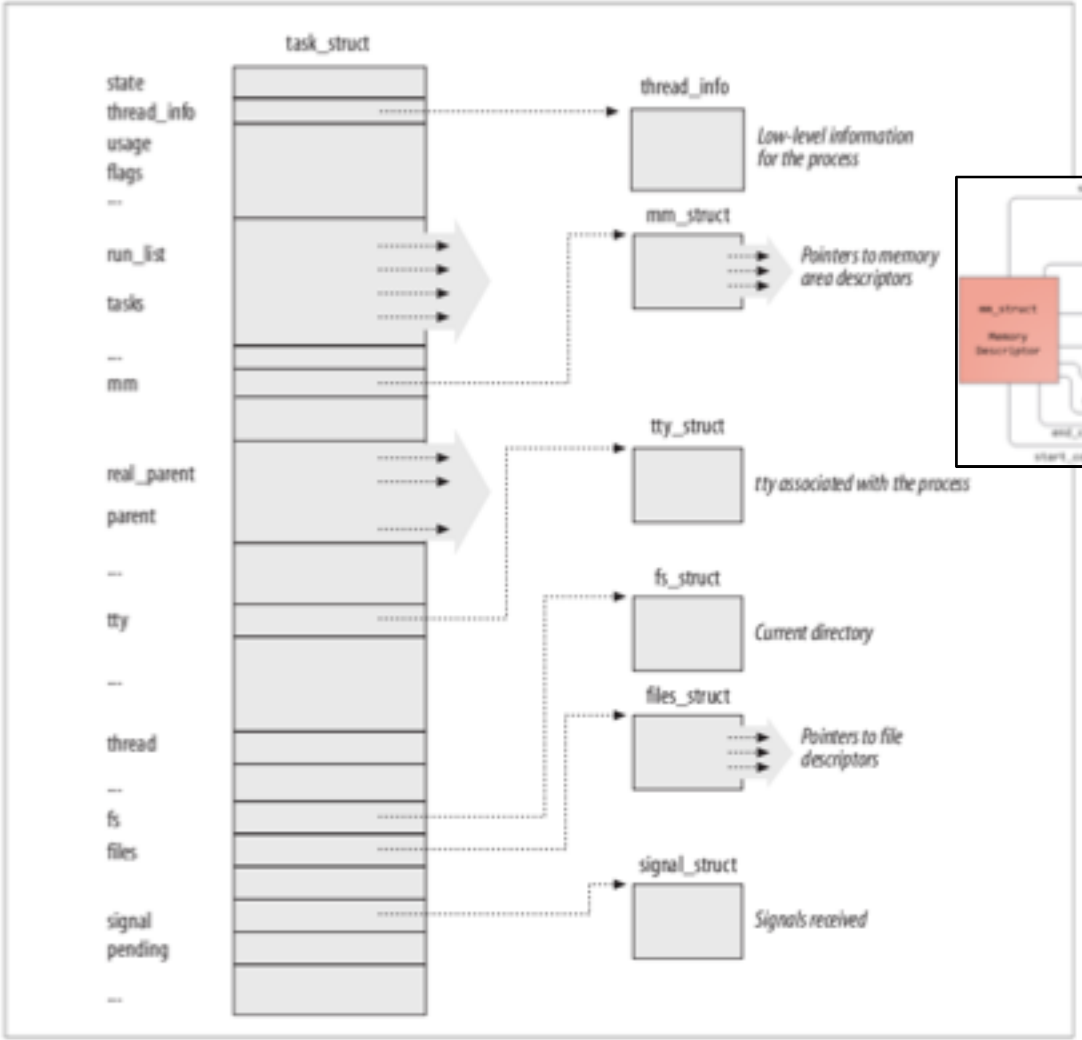
Accounting Info: Amount of CPU and real time used, time limits, account numbers, job or process numbers.

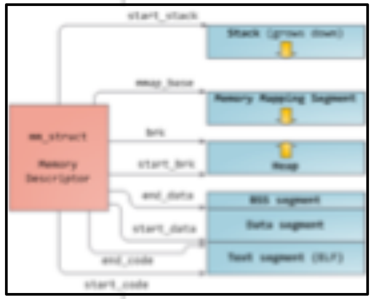
I/O Status Info: The list of I/O devices allocated to process, list of open files

Operating Systems *differ* in Process Representation

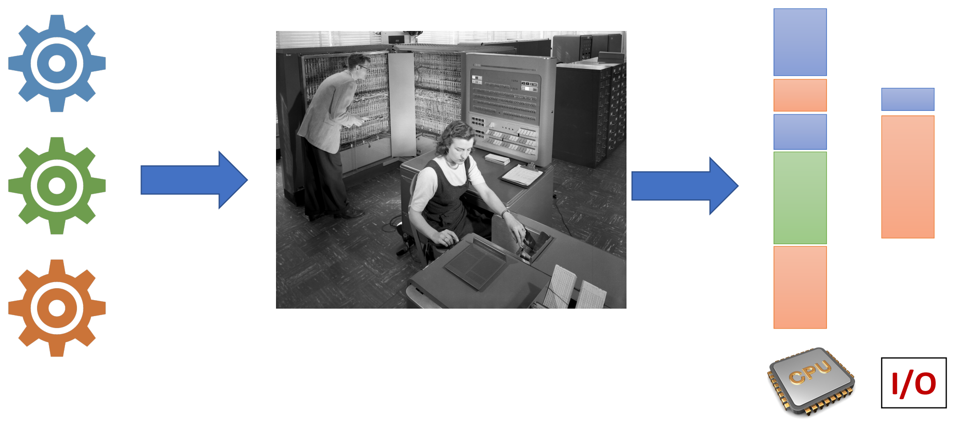


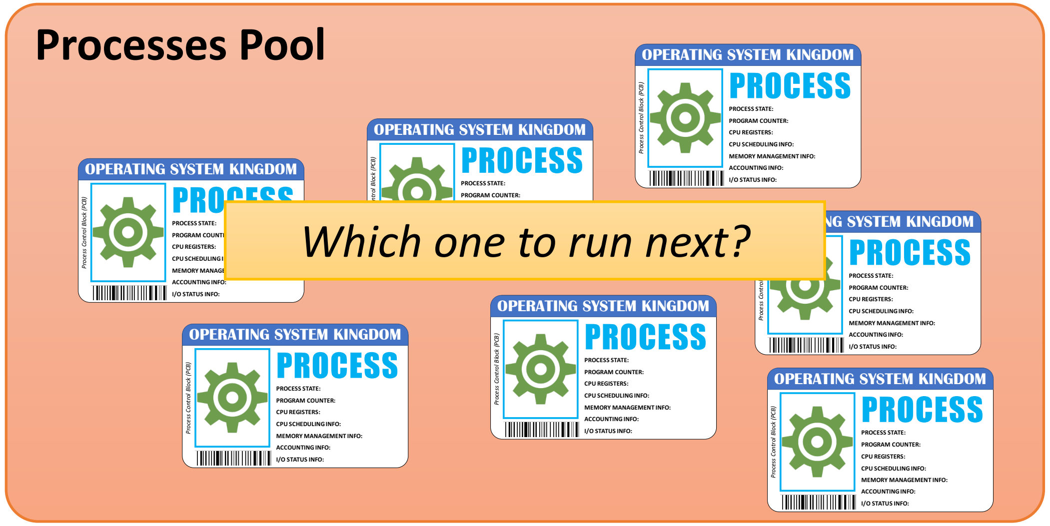
https://github.com/torvalds/linux/blob/master/include/linux/sched.h#L1501
http://www.tldp.org/LDP/tlk/ds/ds.html





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

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

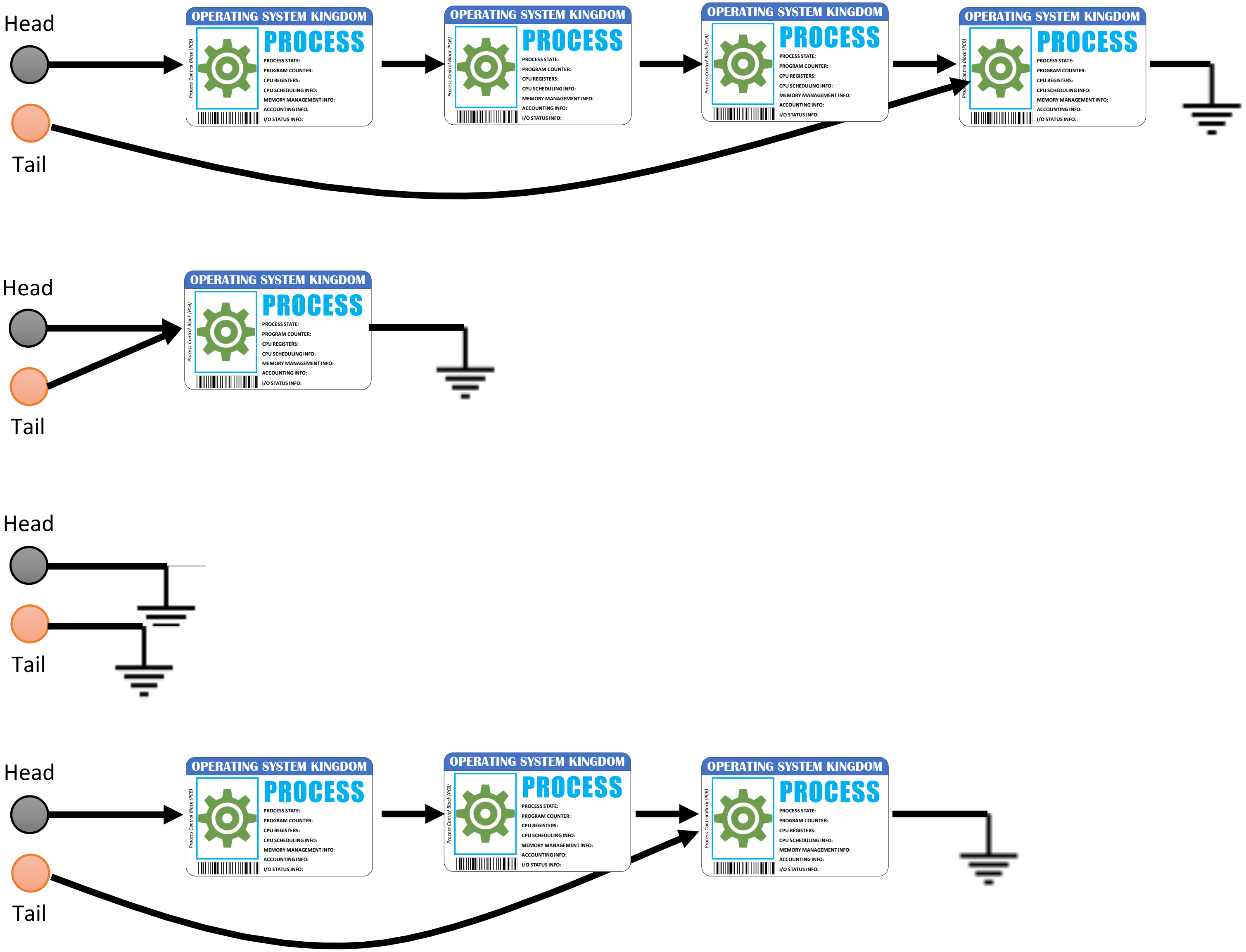
Set of all processes residing in main memory, ready and waiting to execute

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

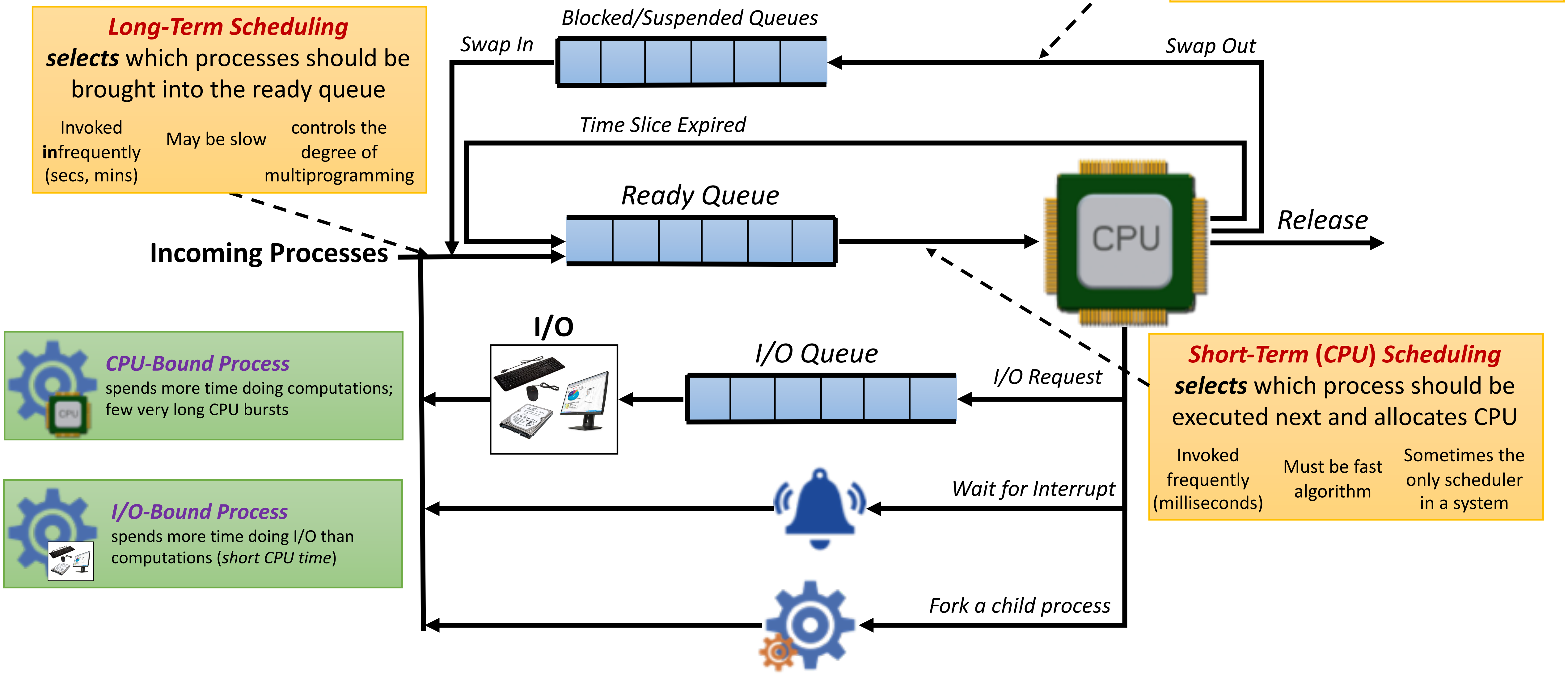


Representation of Process Scheduling

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

process swapping scheduler

Reduces the degree of multiprogramming



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

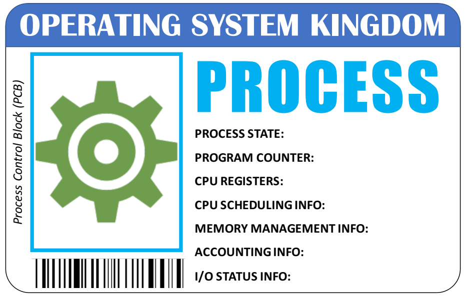
Context Switching

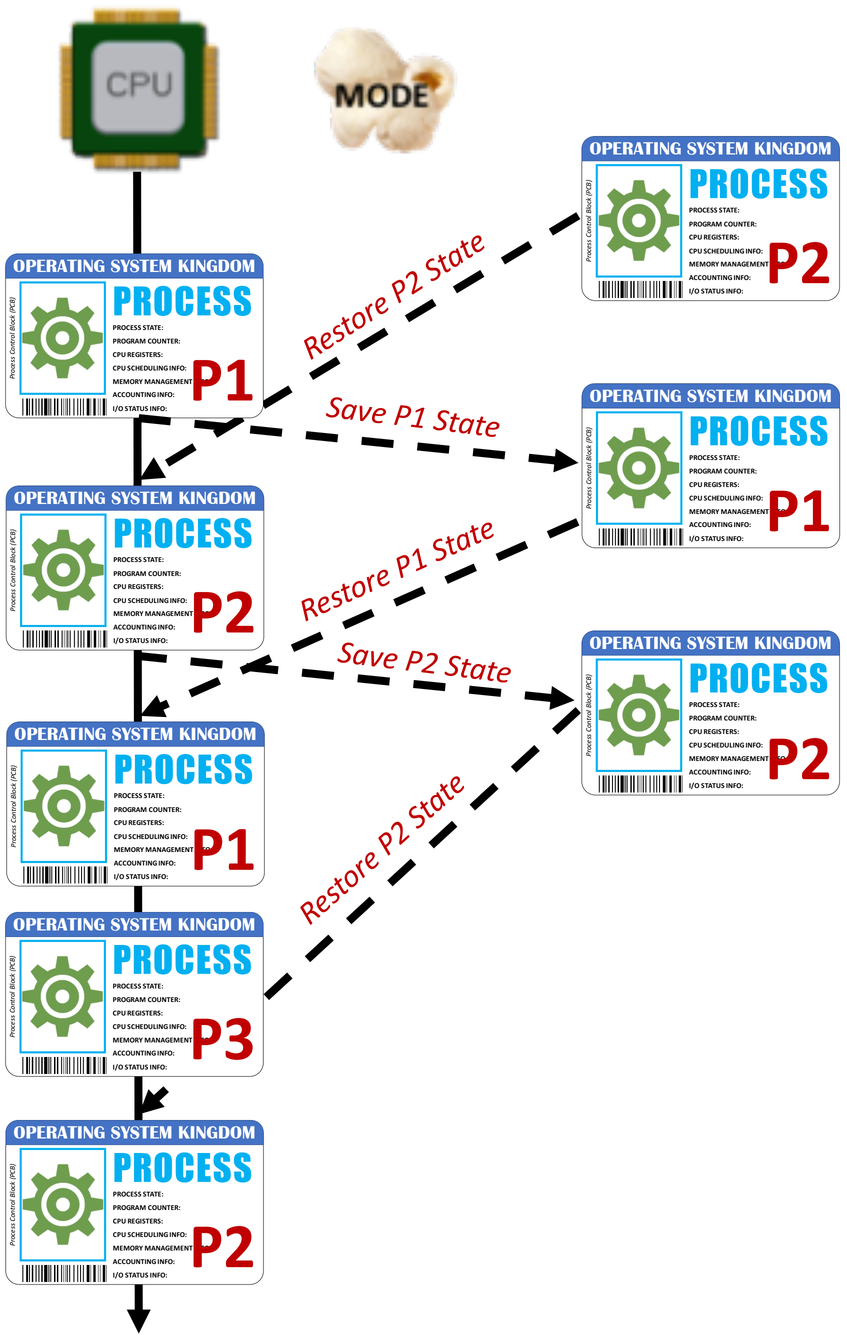
enables multiple processes to share a single CPU

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





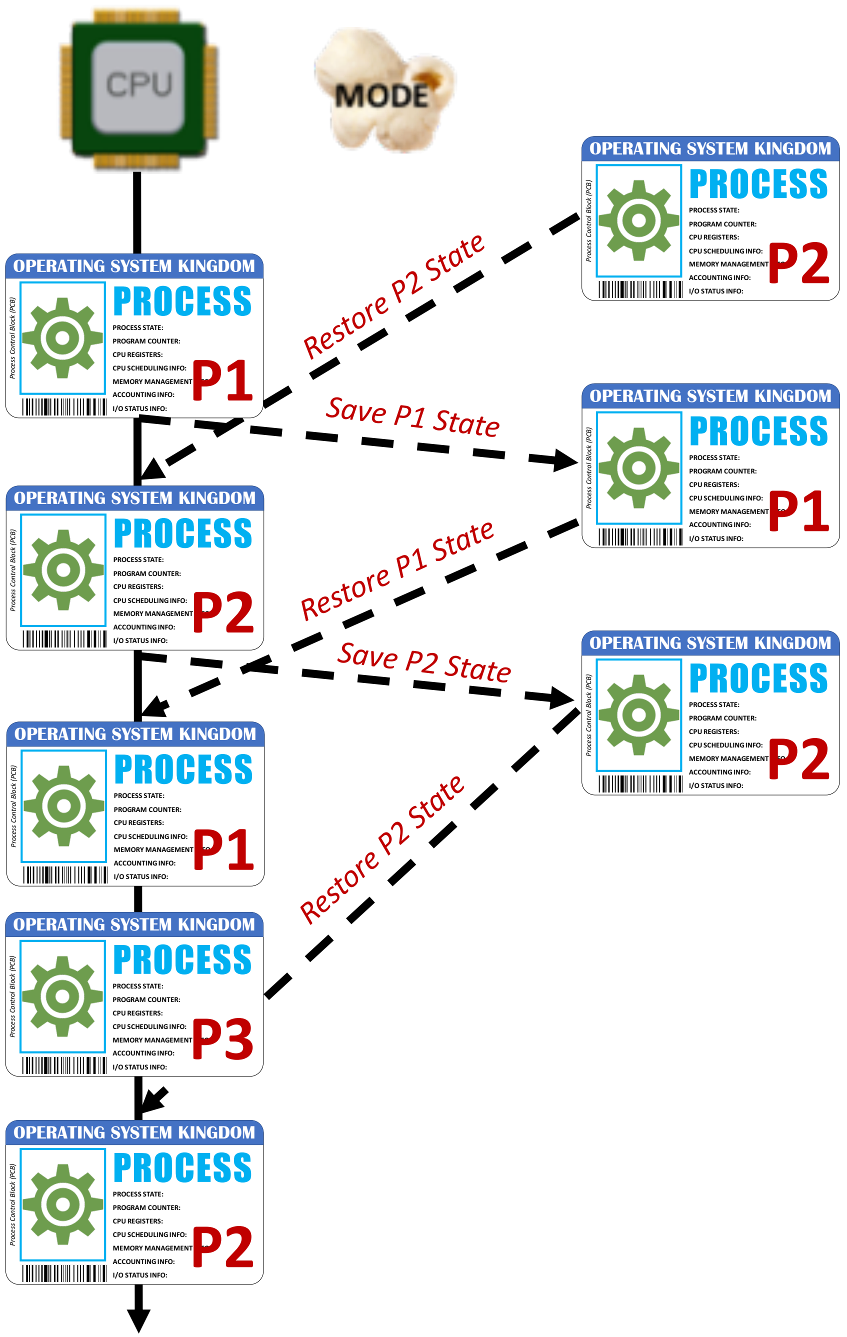
Context Switching

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

System Call

Provide Abstractions

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

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

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

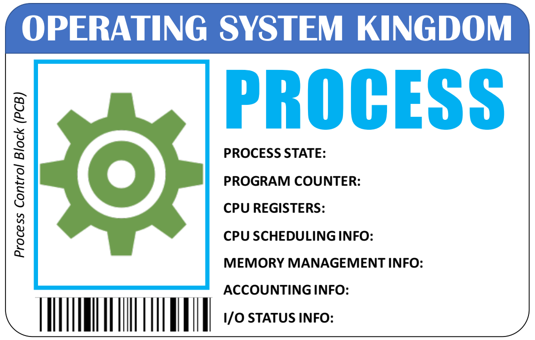
Get/Set Time or Date
Get/Set System Data

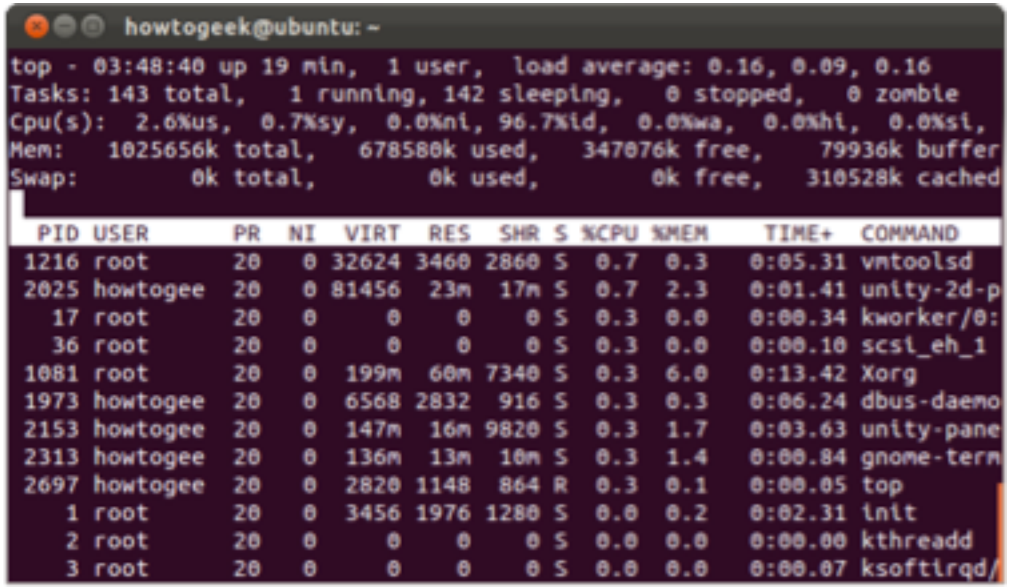
Request/Release/Read/Write Device
Get/Set Device Attributes
Logically Attach/Detach devices

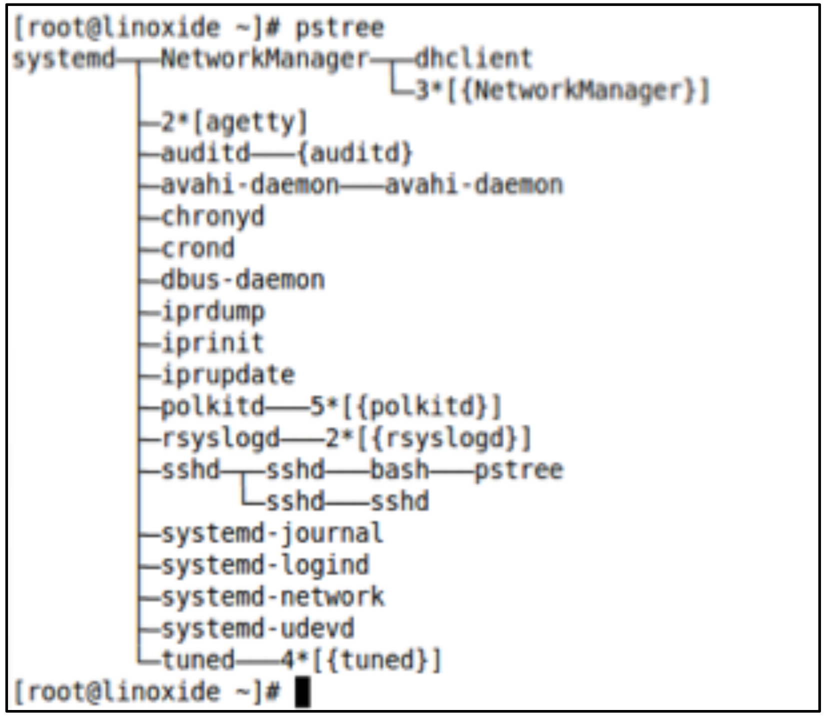
Process Creation

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**







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

Parent and children execute concurrently

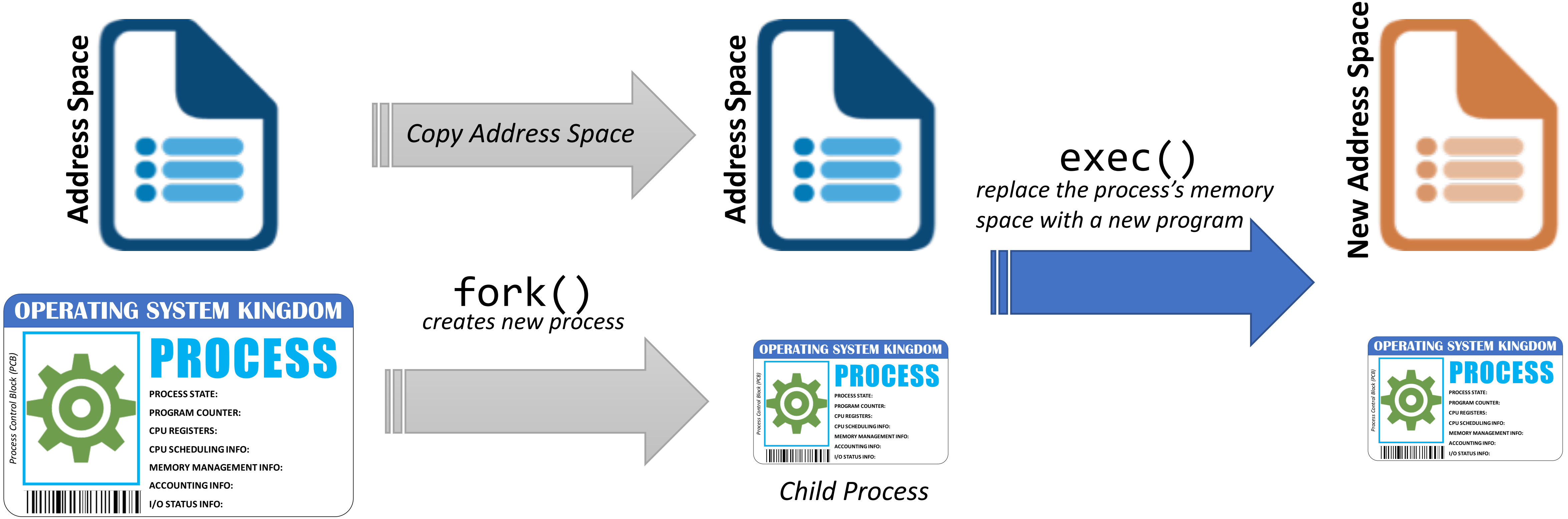
Parent waits until children terminate

Address Space

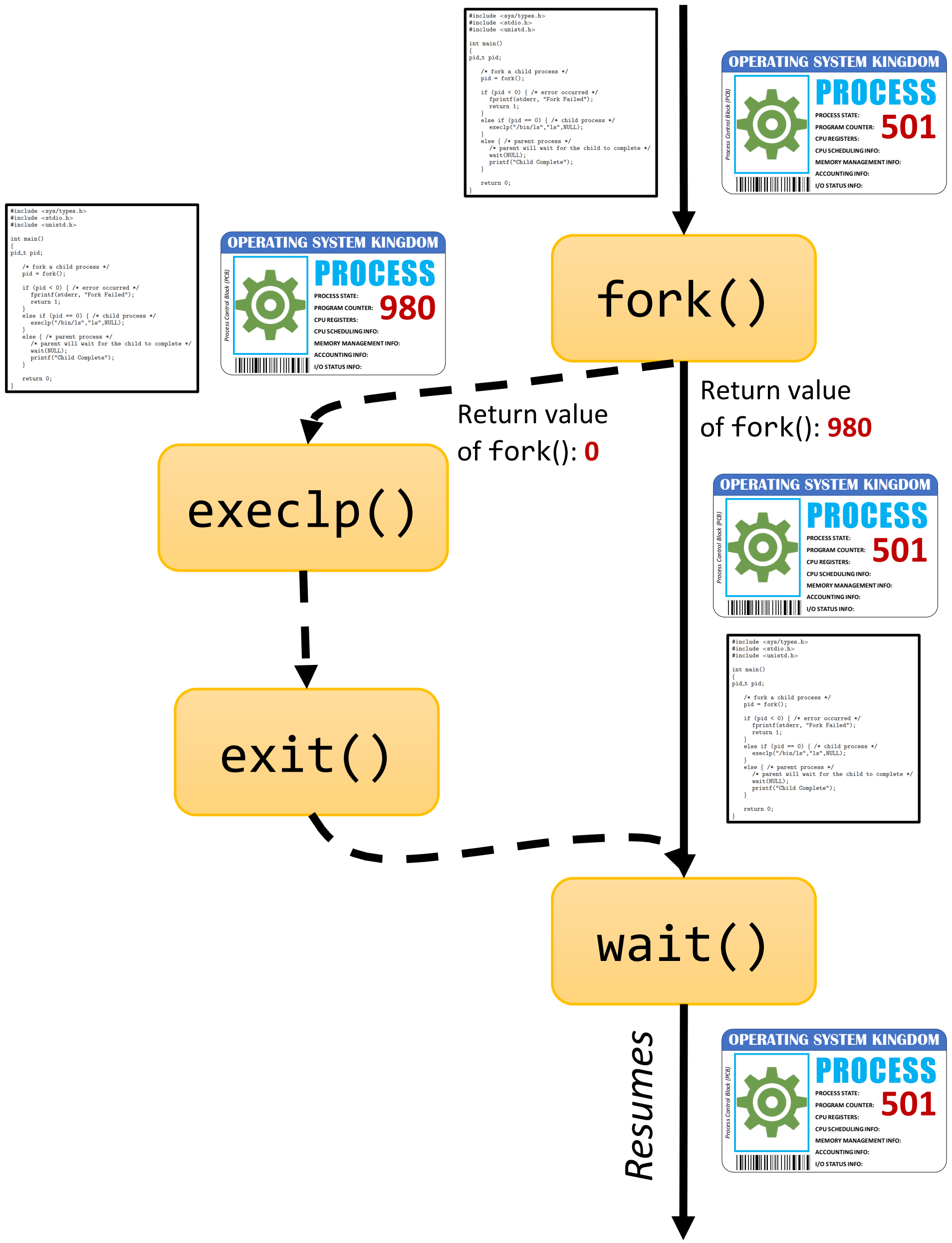
Child duplicate of parent

Child has a program loaded into it

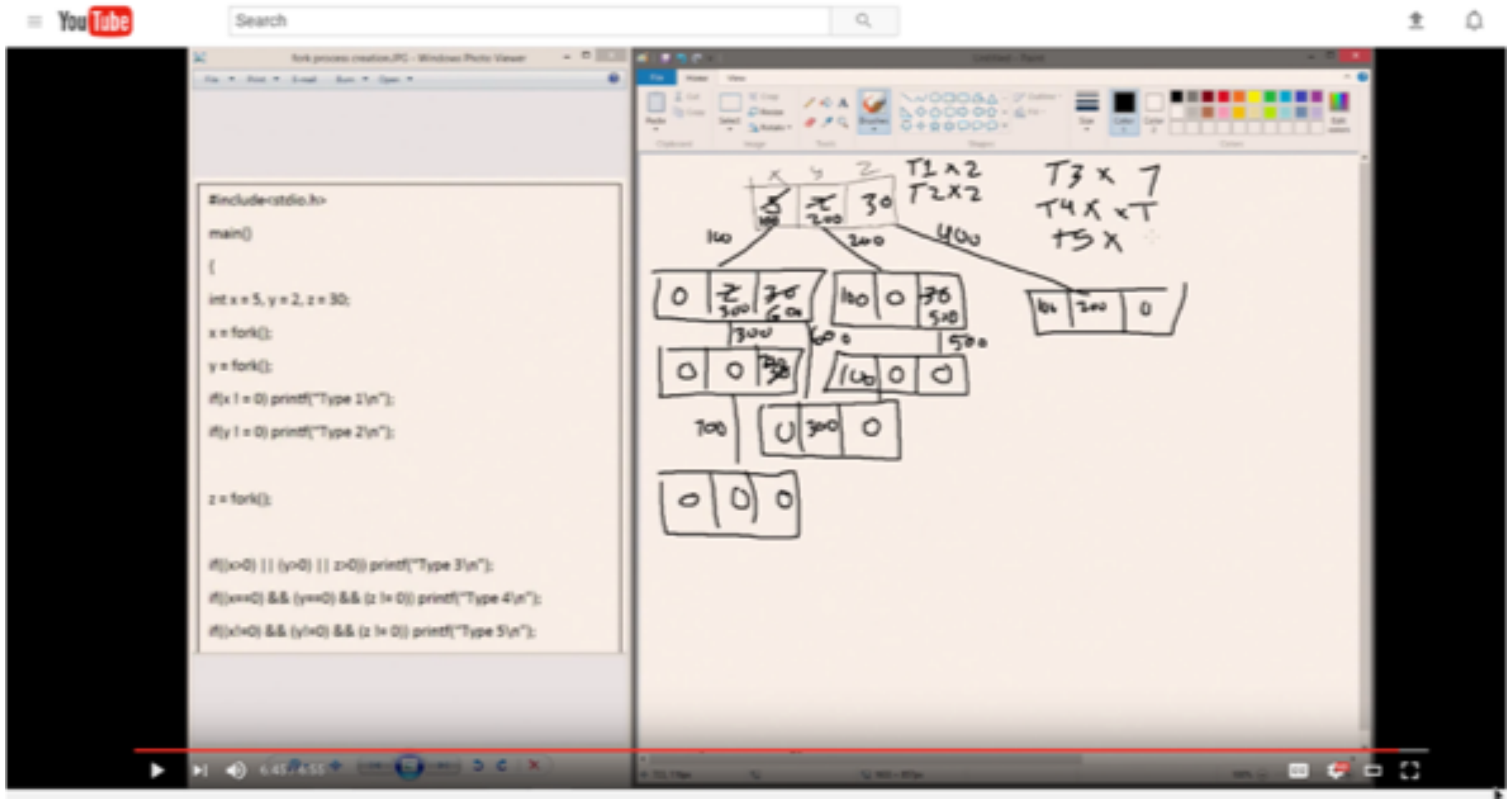
Process Creation



Process Creation

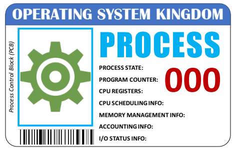



```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

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;
}
```



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

```c
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

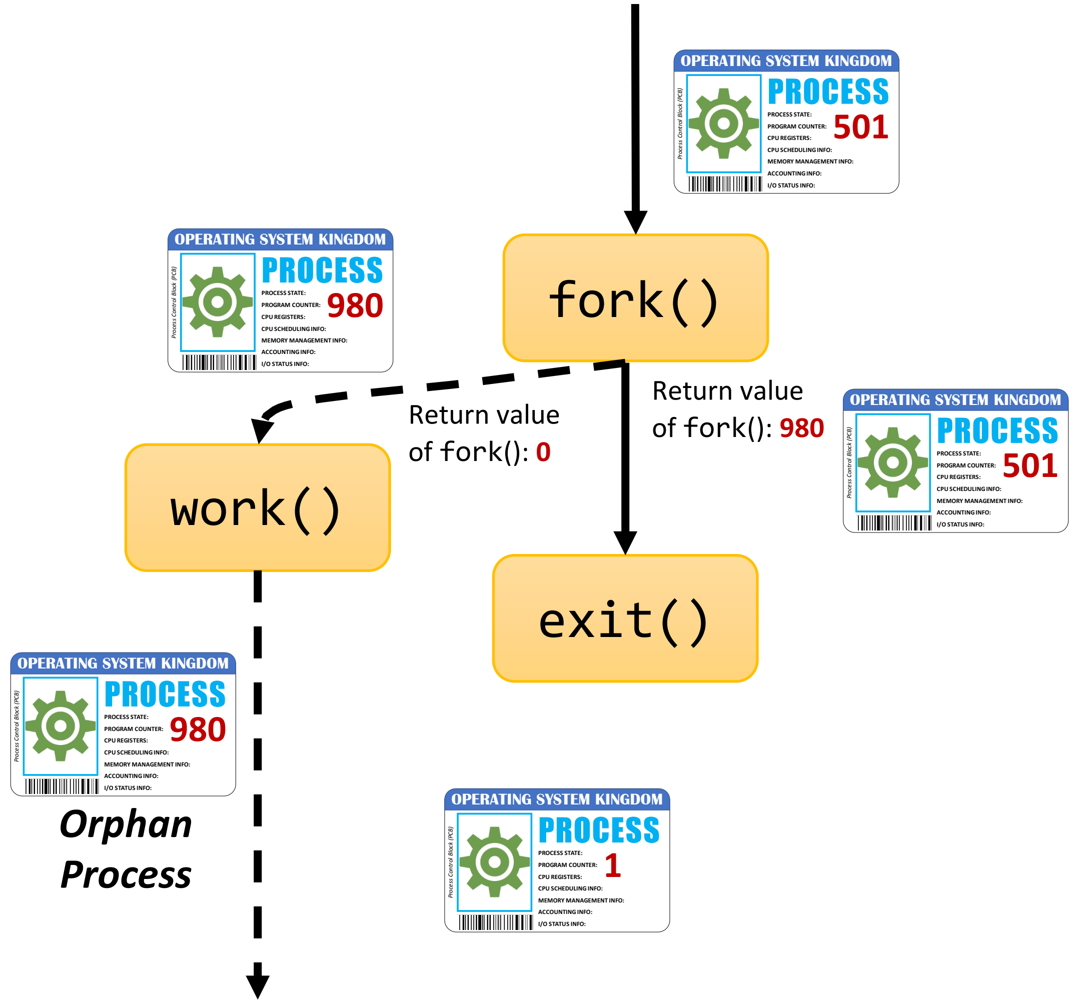
The parent process may wait for termination of a child process by using **`wait`**()

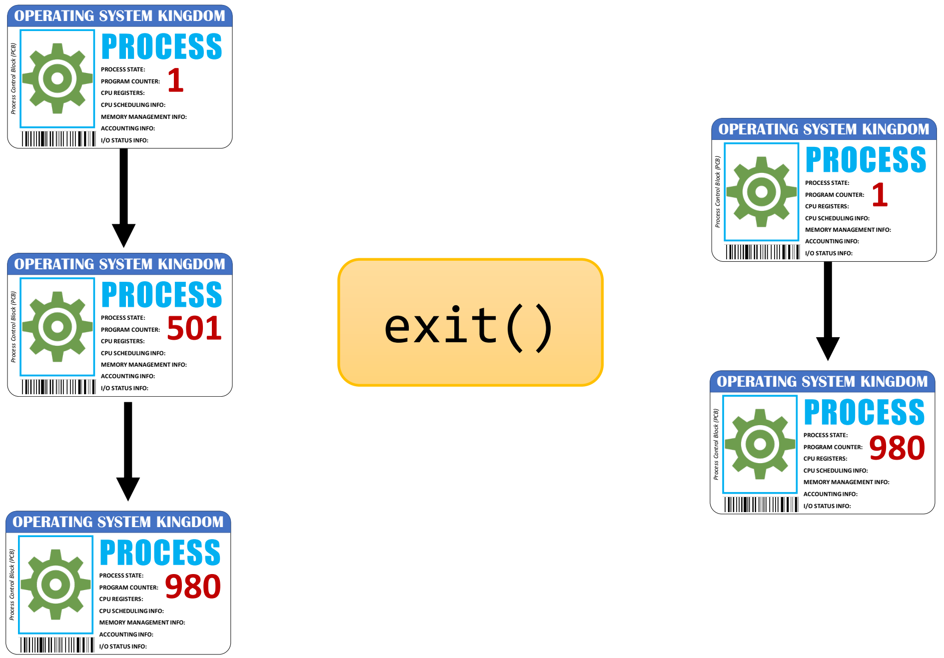
Returns status data from child to parent via **`pid = wait(&status);`**

Process' resources are deallocated by OS

Orphan Process

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



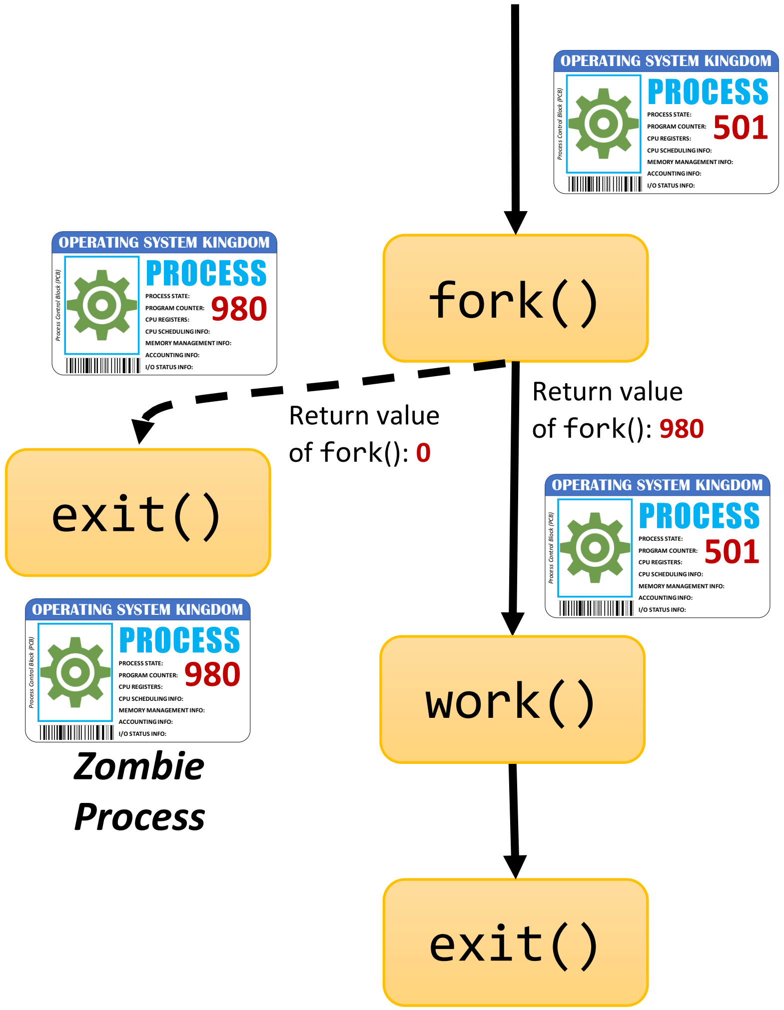


Some operating systems do not allow ”**Orphan**” processes to exists. If a process terminates, then all its children must also be terminated

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



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


أي حد يسأل عني
أنا رايح أنام -_-

つづく