# Operating Systems
## CS-C3140, Lecture 4

Alexandru Paler

# Announcements

- Exam: 13 December 2022, 17:00, U2
  - was 7 December 2022
  - due to overlapping project meetings
  - details about exact form will be announced asap
- Assignments will be opened each Sunday 23:59 -> deadline 7(14) days

# Processor State Information

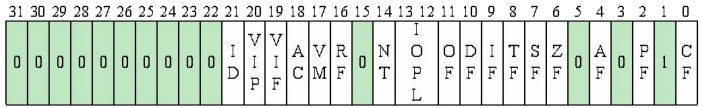
Consists of the contents of processor registers

- User-visible registers
- Control and status registers
- Stack pointers

**Program status word (PSW)**

- Contains condition codes plus other status information
- EFLAGS register is an example of a PSW used by any OS running on an x86 processor

# x86 EFLAGS Register Bits




X ID = Identification flag
X VIP = Virtual interrupt pending
X VIF = Virtual interrupt flag
X AC = Alignment check
X VM = Virtual 8086 mode
X RF = Resume flag
X NT = Nested task flag
X IOPL = I/O privilege level
S OF = Overflow flag

C DF = Direction flag
X IF = Interrupt enable flag
X TF = Trap flag
S SF = Sign flag
S ZF = Zero flag
S AF = Auxiliary carry flag
S PF = Parity flag
S CF = Carry flag

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Shaded bits are reserved

**Figure 3.12 x86 EFLAGS Register**

**Status Flags (condition codes)**

**AF (Auxiliary carry flag)**
Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.

**CF (Carry flag)**
Indicates carrying out or borrowing into the leftmost bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.

**OF (Overflow flag)**
Indicates an arithmetic overflow after an addition or subtraction.

**PF (Parity flag)**
Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.

**SF (Sign flag)**
Indicates the sign of the result of an arithmetic or logic operation.

**ZF (Zero flag)**
Indicates that the result of an arithmetic or logic operation is 0.

**Control Flag**

**DF (Direction flag)**
Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).

**System Flags (should not be modified by application programs)**

**AC (Alignment check)**
Set if a word or doubleword is addressed on a nonword or nondoubleword boundary.

**ID (Identification flag)**
If this bit can be set and cleared, this processor supports the CPUID instruction. This instruction provides information about the vendor, family, and model.

**RF (Resume flag)**
Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.

**IOPL (I/O privilege level)**
When set, causes the processor to generate an exception on all accesses to I/O devices during protected mode operation.

**IF (Interrupt enable flag)**
When set, the processor will recognize external interrupts.

**TF (Trap flag)**
When set, causes an interrupt after the execution of each instruction. This is used for debugging.

**NT (Nested task flag)**
Indicates that the current task is nested within another task in protected mode operation.

**VM (Virtual 8086 mode)**
Allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.

**VIP (Virtual interrupt pending)**
Used in virtual 8086 mode to indicate that one or more interrupts are awaiting service.

**VIF (Virtual interrupt flag)**
Used in virtual 8086 mode instead of IF.

# Process Control Information

- The most important data structure in an OS
  - Contains all of the information about a process that is needed by the OS
  - Blocks are read and/or modified by virtually every module in the OS
  - Defines the state of the OS
- **Difficulty is not access, but protection**
  - A bug in a single routine could damage process control blocks, which could destroy the system's ability to manage the affected processes
  - A design change in the structure or semantics of the process control block could affect a number of modules in the OS

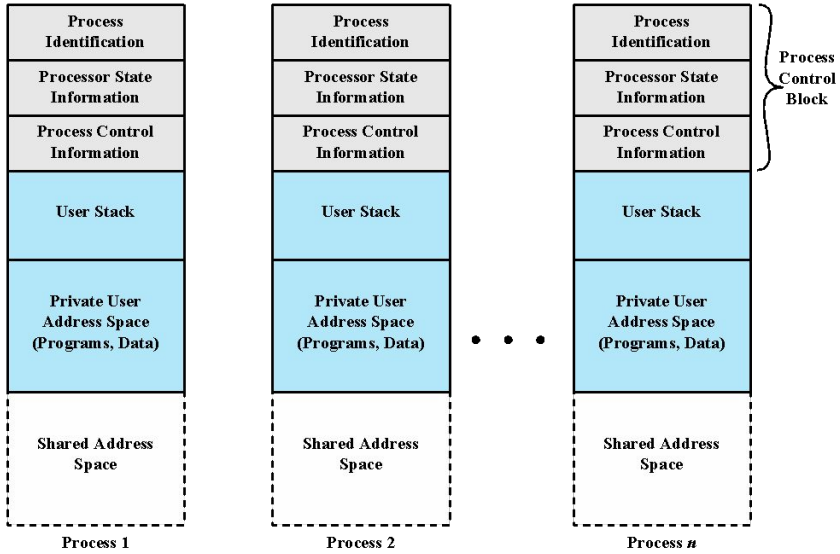
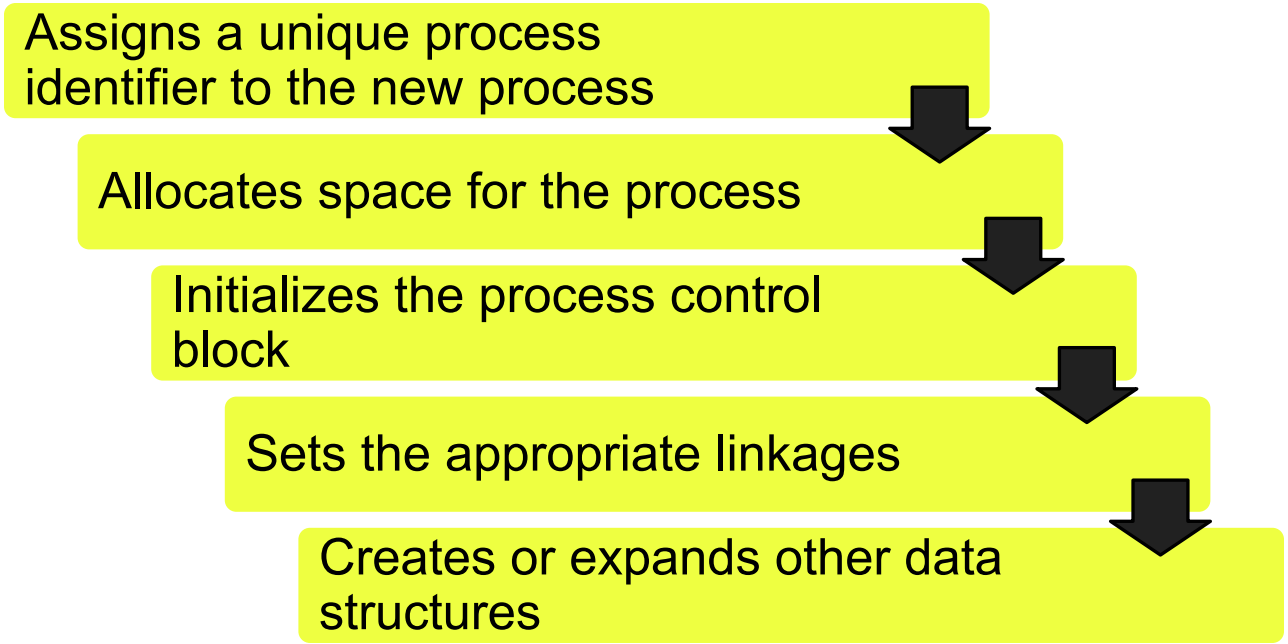



**Figure 3.13 User Processes in Virtual Memory**

# Process Creation

- Once the OS decides to create a new process it:



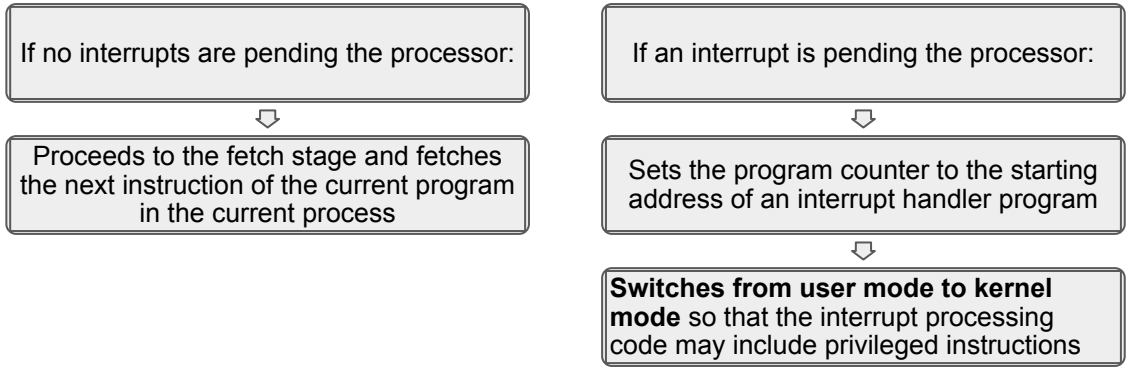
# Interrupting the Execution of a Process

- Interrupt
  - Due to some sort of event that is external to and independent of the currently running process
    - Clock interrupt
    - I/O interrupt
    - Memory fault
  - Time slice
    - The maximum amount of time that a process can execute before being interrupted
- Trap
  - An error or exception condition generated within the currently running process
  - OS determines if the condition is fatal
    - Moved to the Exit state and a process switch occurs
    - Action will depend on the nature of the error the design of the OS

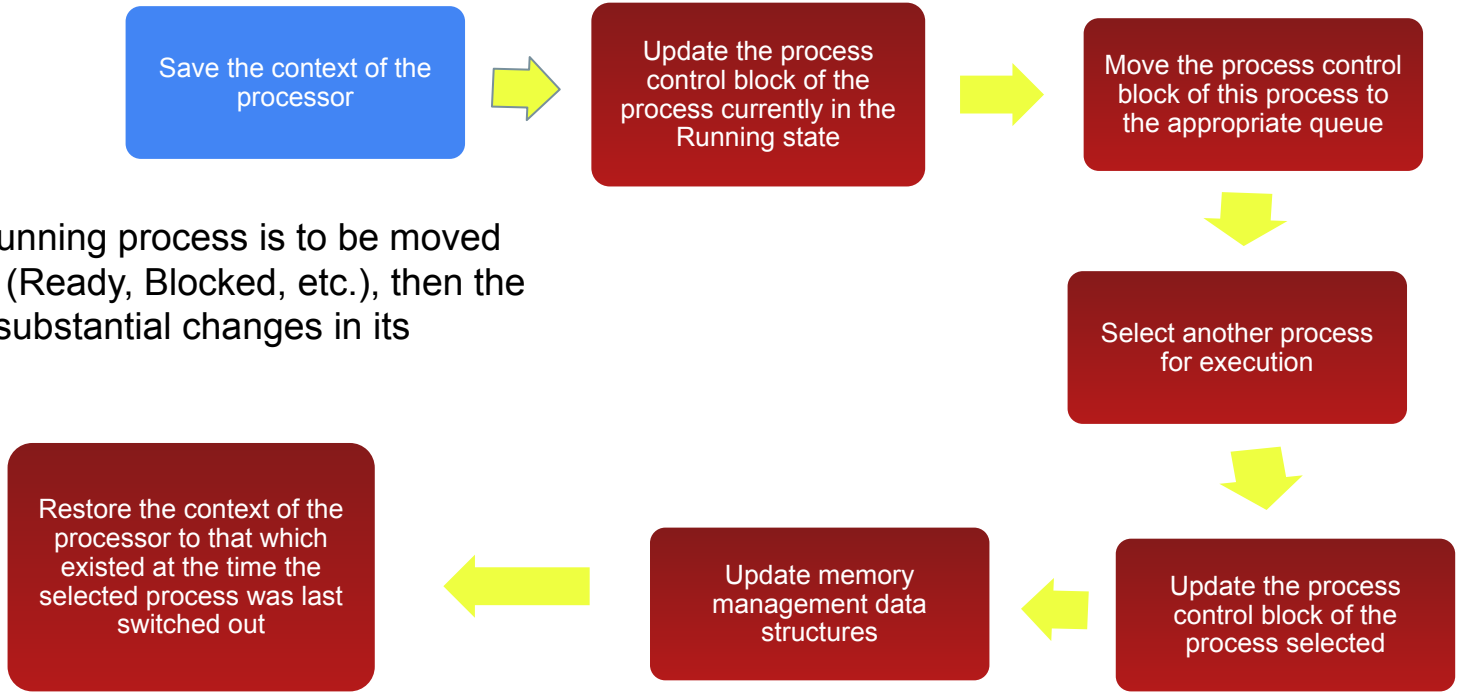
| Mechanism | Cause | Use |
|---|---|---|
| Interrupt | External to the execution of the current instruction | Reaction to an asynchronous external event |
| Trap | Associated with the execution of the current instruction | Handling of an error or an exception condition |
| Supervisor call | Explicit request | Call to an operating system function |

# Modes of Execution and Switching

- User Mode
  - Less-privileged mode
  - User programs typically execute in this mode
- System Mode
  - More-privileged mode
  - Also referred to as control mode or kernel mode
  - Kernel of the operating system



# Change of Process State




If the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment

# Execution of the Operating System

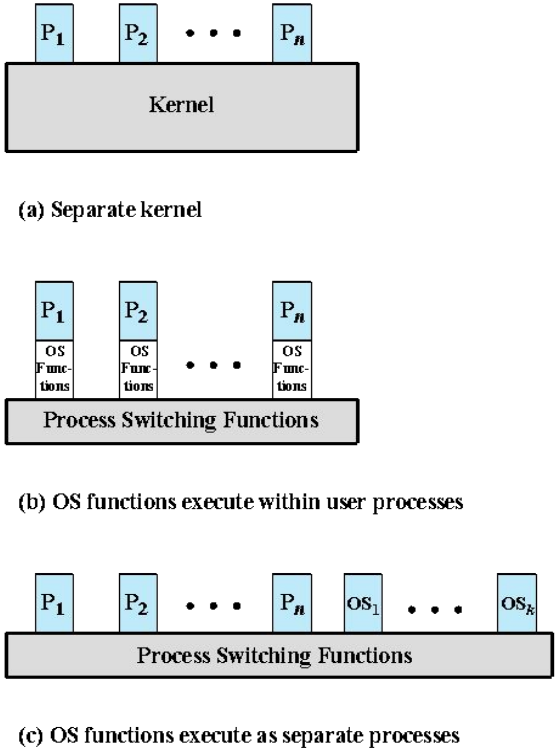



**Figure 3.15 Relationship Between Operating System and User Processes**

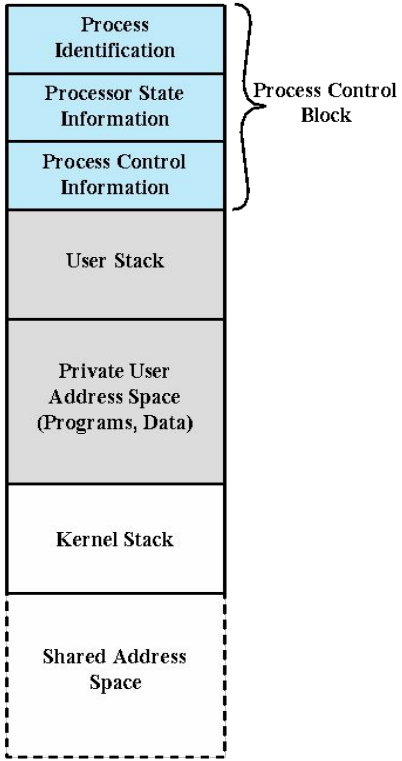



**Figure 3.16 Process Image: Operating System Executes Within User Space**

# Summary

- What is a process?
  - Background
  - Processes and process control blocks
- Process states
  - Two-state process model
  - Creation and termination
  - Five-state model
  - Suspended processes
- Process description
  - Operating system control structures
  - Process control structures
- Process control
  - Modes of execution
  - Process creation
  - Process switching
- Execution of the operating system
  - Nonprocess kernel
  - Execution within user processes
  - Process-based operating system

# Processes vs. Threads

# Processes and Threads

- The unit of dispatching is referred to as a **thread or lightweight process**
- The unit of resource ownership is referred to as a **process or task**
- **Multithreading** - The ability of an OS to support multiple, concurrent paths of execution within a single process

Resource Ownership

- Process includes a virtual address space to hold the process image
- The OS performs a protection function to prevent unwanted interference between processes with respect to resources

Scheduling/Execution

- Follows an execution path that may be interleaved with other processes
- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

# Single- and Multi-Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS: single process and single thread

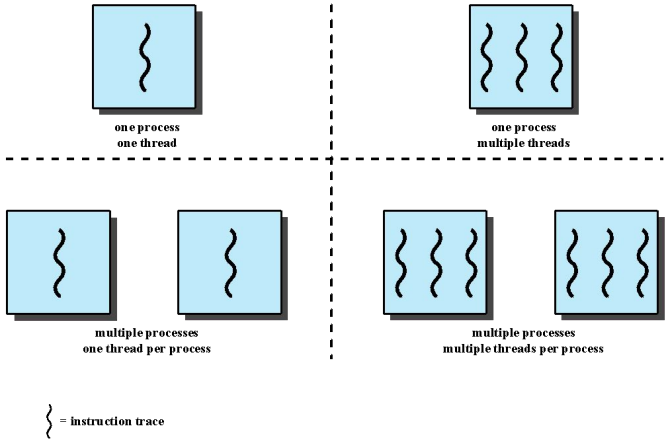



Figure 4.1 Threads and Processes

- A Java run-time environment is an example of a system of one process with multiple threads

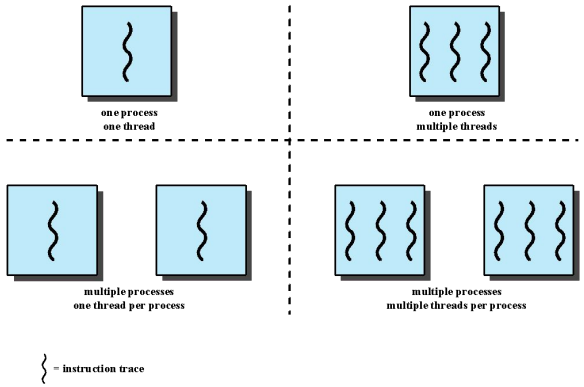



Figure 4.1 Threads and Processes

# One or More Threads in a Process

Process - Defined in a multithreaded environment as “the unit of resource allocation and a unit of protection”:

- Associated with processes:
- A virtual address space that holds the process image
- Protected access to:
- Processors
- Other processes (for interprocess communication)
- Files
- I/O resources (devices and channels)

Each thread has:

- An execution state (Running, Ready, etc.)
- A saved thread context when not running
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its processes, shared with all other threads in that process

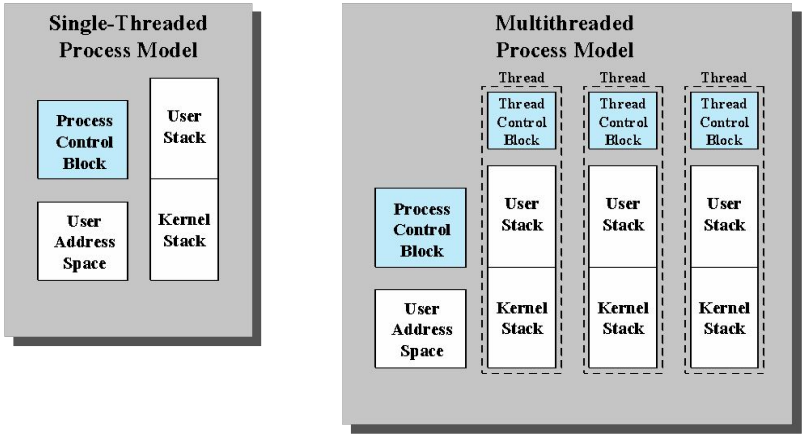
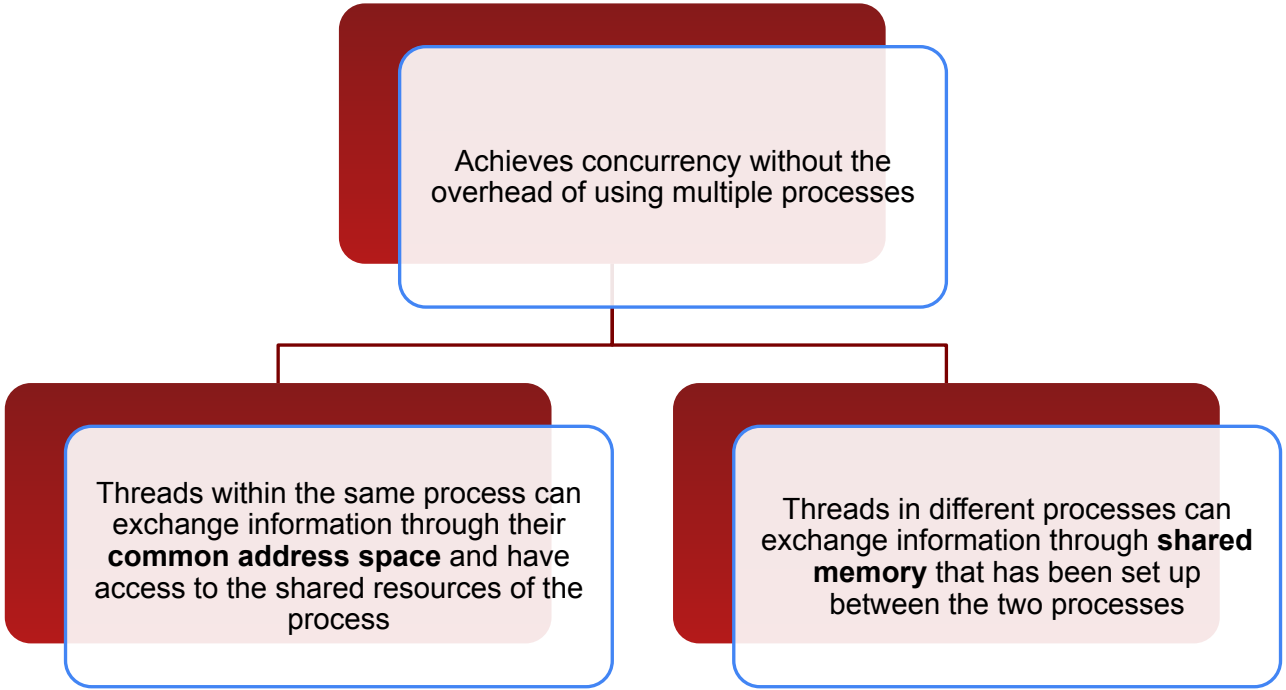
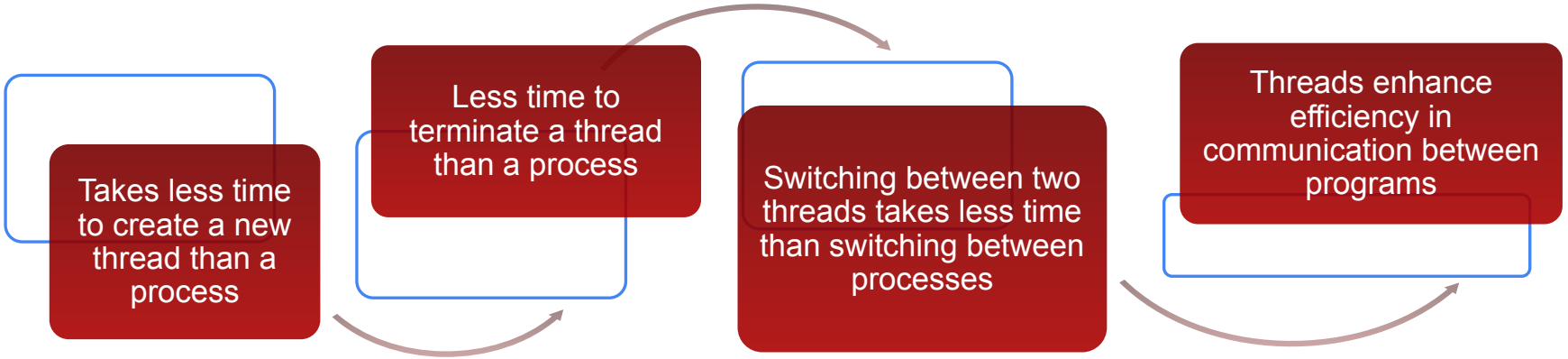



**Figure 4.2 Single Threaded and Multithreaded Process Models**

# Multithreading

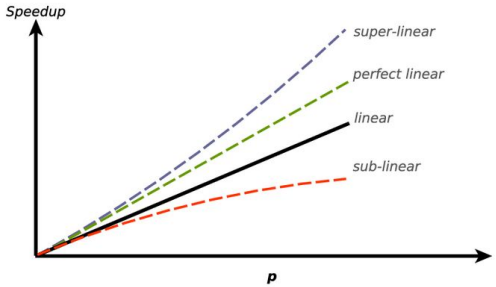


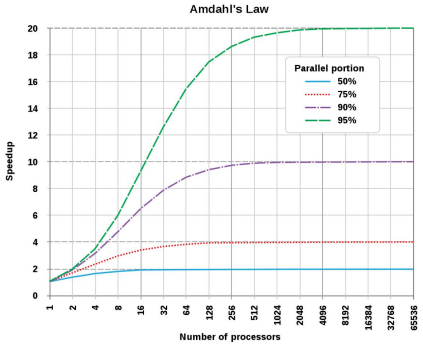
# Key Benefits of Threads

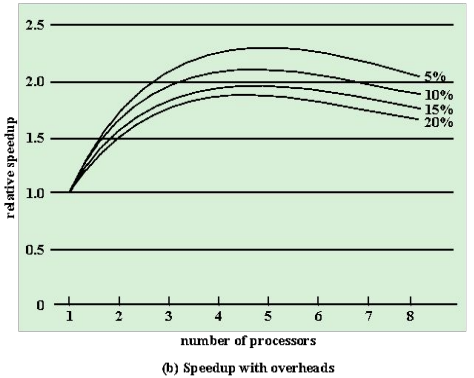


# Applications That Benefit from Multi-Threading

- Multithreaded native applications - small number of highly threaded processes
- Multiprocess applications - have many single-threaded processes
- Examples:
  - Java applications benefit from multicore technology
  - Multi-instance applications: Multiple instances of the application in parallel

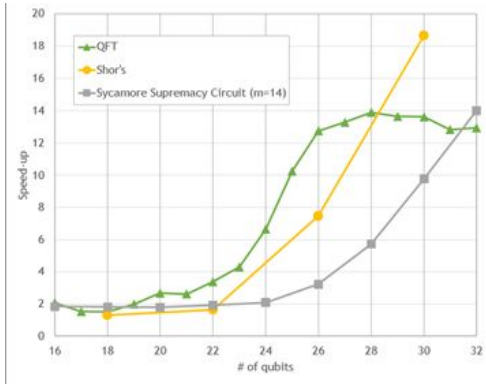


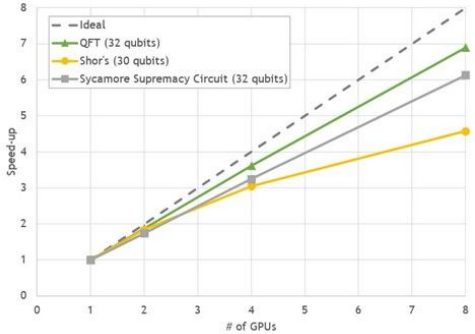



(b) Speedup with overheads

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$




https://developer.nvidia.com/blog/accelerating-quantum-circuit-simulation-with-nvidia-custatevec/

# Relationship between Threads and Processes

| Threads:Processes | Description | Example Systems |
|---|---|---|
| **1:1** | Each thread of execution is a unique process with its own address space and resources. | Traditional UNIX implementations |
| **M:1** | A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process. | Windows NT, Solaris, Linux, OS/2, OS/390, MACH |
| **1:M** | A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems. | Ra (Clouds), Emerald |
| **M:N** | Combines attributes of M:1 and 1:M cases. | TRIX |

# Threads and Their Use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

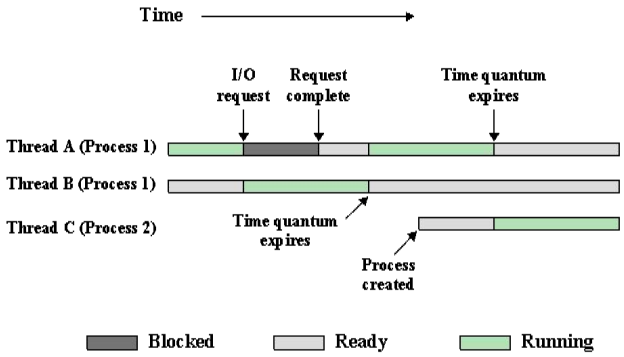
- Thread synchronization
  - It is necessary to synchronize the activities of the various threads
  - All threads of a process share the same address space and other resources
  - Any alteration of a resource by one thread affects the other threads in the same process

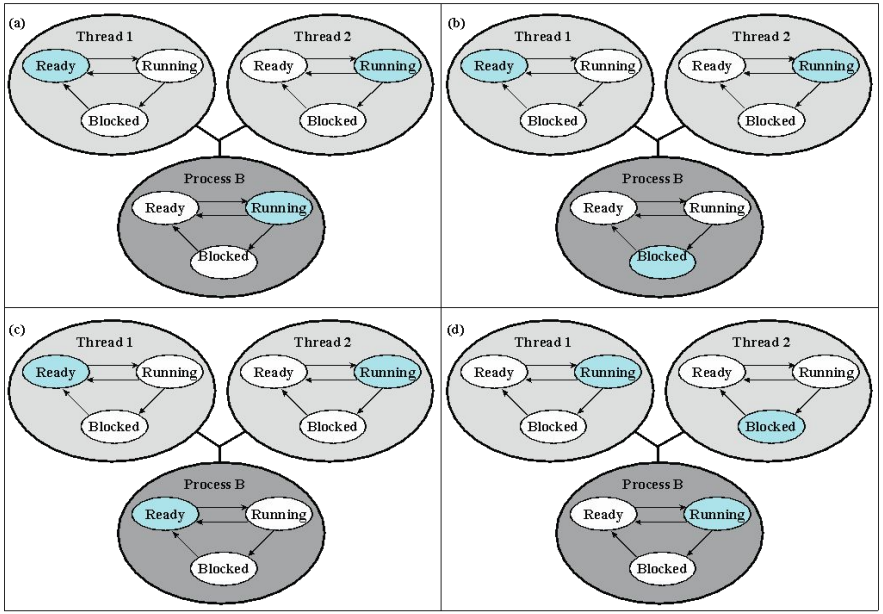
- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
- Suspending a process involves suspending all threads of the process
- Termination of a process terminates all threads within the process

# Thread Execution States

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

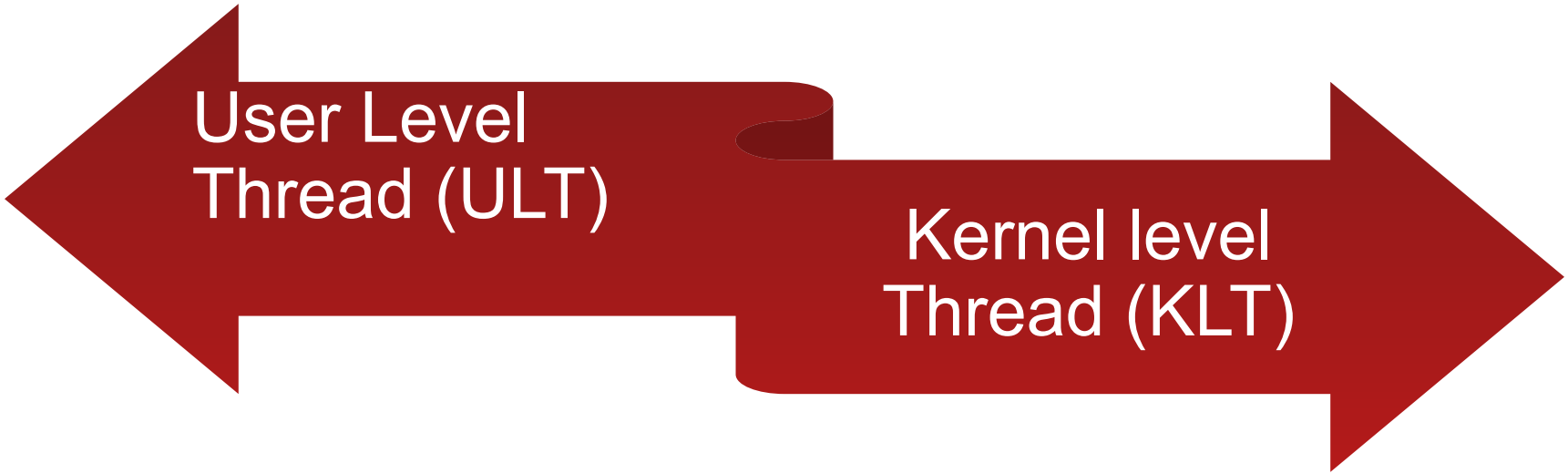



Colored state is current state

**Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States**

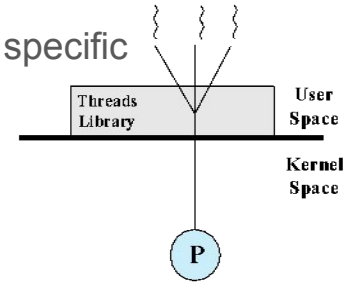
# Types of Threads



# ULT: Advantages and Disadvantages

- All thread management is done by the application
- The kernel is not aware of the existence of threads

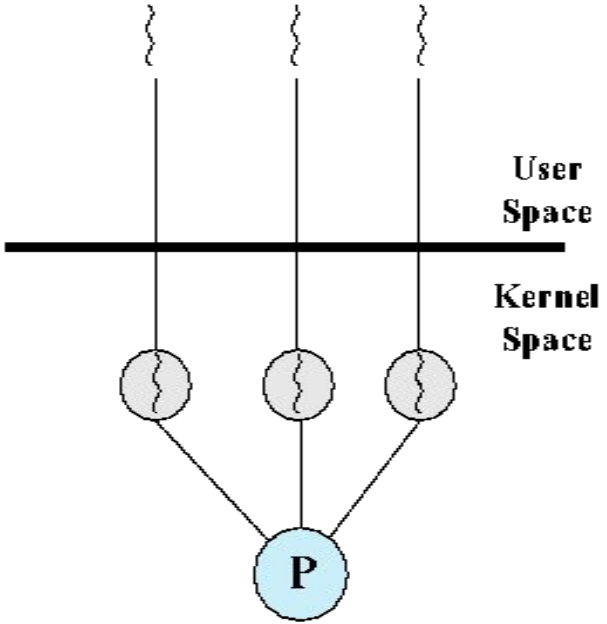
- Thread switching does not require kernel mode privileges
- Scheduling can be application specific
- ULTs can run on any OS




(a) Pure user-level

- In a typical OS many system calls are blocking
  - when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
  - Jacketing: convert a blocking system call into a non-blocking system call
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

# Kernel-Level Threads (KLTs)




(b) Pure kernel-level

- Thread management is done by the kernel
- There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
- **Windows is an example of this approach**

# Advantages and Disadvantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded
- Combined Approach: Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application

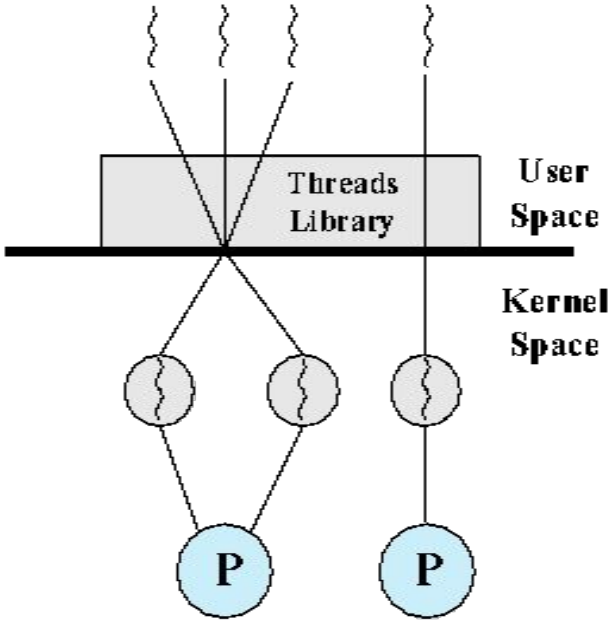
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

| Operation | User-Level Threads | Kernel-Level Threads | Processes |
|---|---|---|---|
| Null Fork | 34 | 948 | 11,300 |
| Signal Wait | 37 | 441 | 1,840 |

Example: Thread and Process Operation Latencies (μs).
Switching to Kernel-Level is more costly.

# Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application




(c) Combined

# Linux: Processes and Threads

# Linux Namespaces

- A namespace enables a process to have a different view of the system than other processes that have other associated namespaces
  - Widely by Linux Containers (LXC) projects
  - Different from Docker containers
- Namespaces
  - created by the **clone()** system call
  - gets as a parameter one of the six namespaces clone flags (CLONE_NEWNS, CLONE_NEWPID, CLONE_NEWNET, CLONE_NEWIPC, CLONE_NEWUTS, and CLONE_NEWUSER).
- A process can also create a namespace with the **unshare()** system call with the same flags;
  - as opposed to clone(), a new process is not created in such a case
  - a new namespace is created, and is attached to the calling process

Six Namespaces

- UTS Namespace
  - information about the current kernel, including nodename
- Mount Namespace
  - a specific view of the filesystem hierarchy
  - two processes with different mount namespaces see different filesystem hierarchies
- IPC Namespace
  - isolates certain interprocess communication (IPC) resources
  - semaphores, POSIX message queues etc
  - enable IPC among processes that share the same IPC namespace
- PID Namespace
  - processes can have the same PID
- Network Namespace
  - isolation of the system resources associated with networking
  - own network devices, IP addresses, IP routing tables etc.
  - allows processes that belong to the namespace to have the needed network access
- The Linux cgroup Subsystem
  - the basis of lightweight process virtualization
  - forms the basis of Linux containers

# Linux Tasks

- Older versions of Linux offered no support for multithreaded applications.
- Execution flows of a multithreaded application
  - created, handled and scheduled in user mode
  - POSIX pthread library
- Linux uses lightweight processes (see later)
  - better support for multithreaded applications
  - two LWP may share resources
- **Thread group** is a set of LWPs that implement an application and act as a whole

A process, or task, in Linux is represented by a *task_struct* data structure

This structure contains information in a number of categories

- A process can query the kernel to obtain:
  - PID of its parent process
  - the execution state of children
- Unix is not allowed to discard data included in a process descriptor field right after the process terminates
- Descriptor must be saved until the parent process is notified

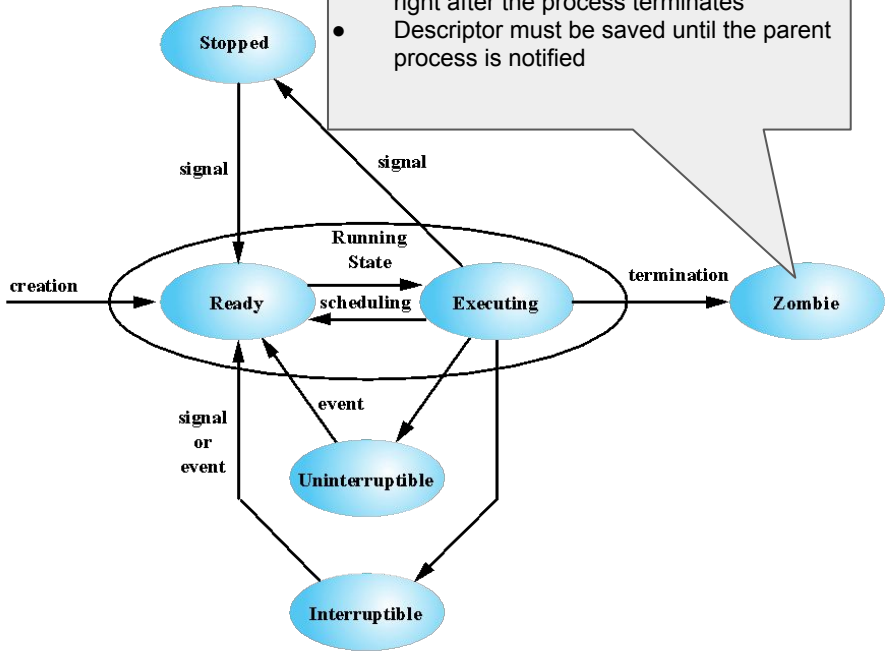
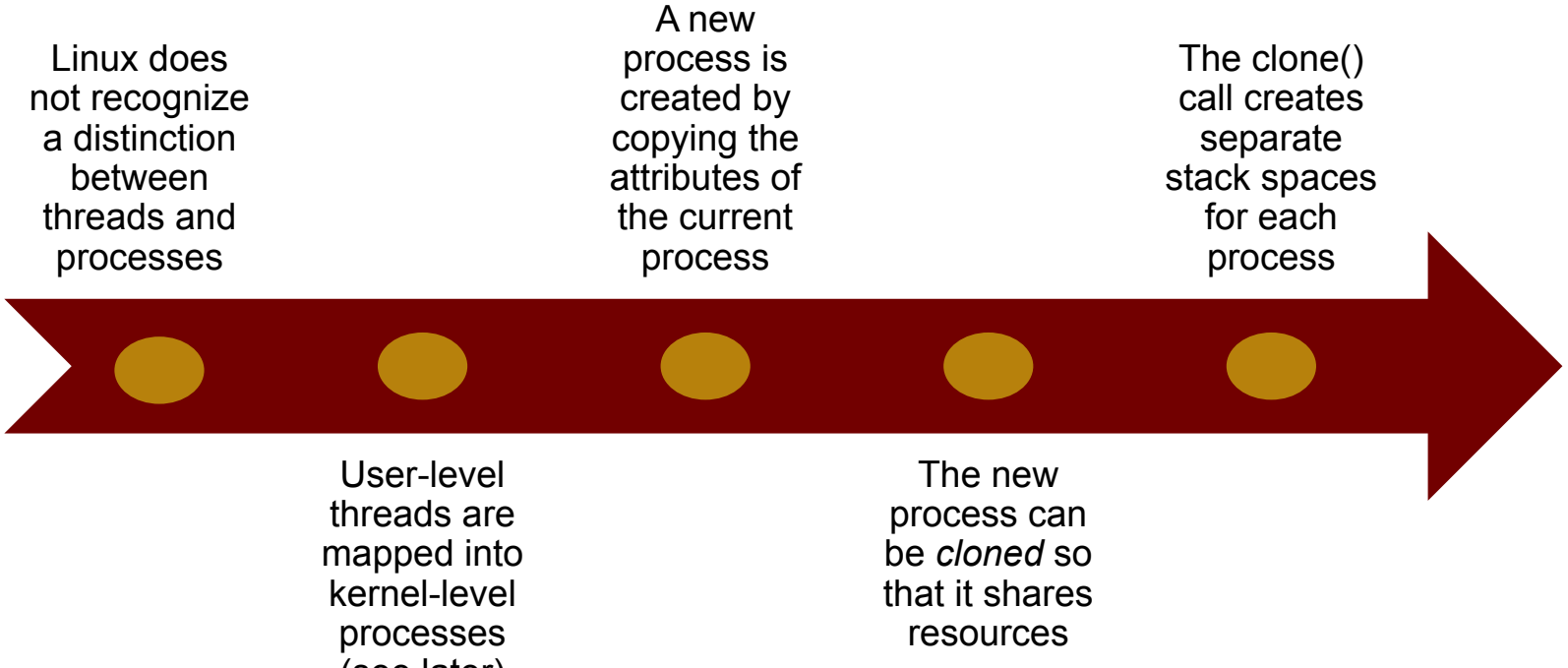



**Figure 4.15 Linux Process/Thread Model**

# Linux Threads



# Lightweight Process (LWP) Data Structure

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it
- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers
- The kernel stack for this LWP
  - includes system call arguments, results,
  - error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure
- https://yajin.org/os2018fall/04_thread_b.pdf

# Identifying a process

- Each execution context that can be independently scheduled must have its own process descriptor
  - 32-bit address of the **task_struct** structure
  - useful for the kernel to identify processes
- PIDs are numbered sequentially
  - getpid() and /proc/sys/kernel/pid_max

```
(base) alexandru@DESKTOP-TKPML3T:~/os$ cat /proc/sys/kernel/pid_max
32768
```

  - recycling: pidmap_array bitmap denotes free and used PIDs
  - all threads of a multithreaded app have the same PID (POSIX 1003.1c standard)
- Process 1 init is the ancestor of all other processes

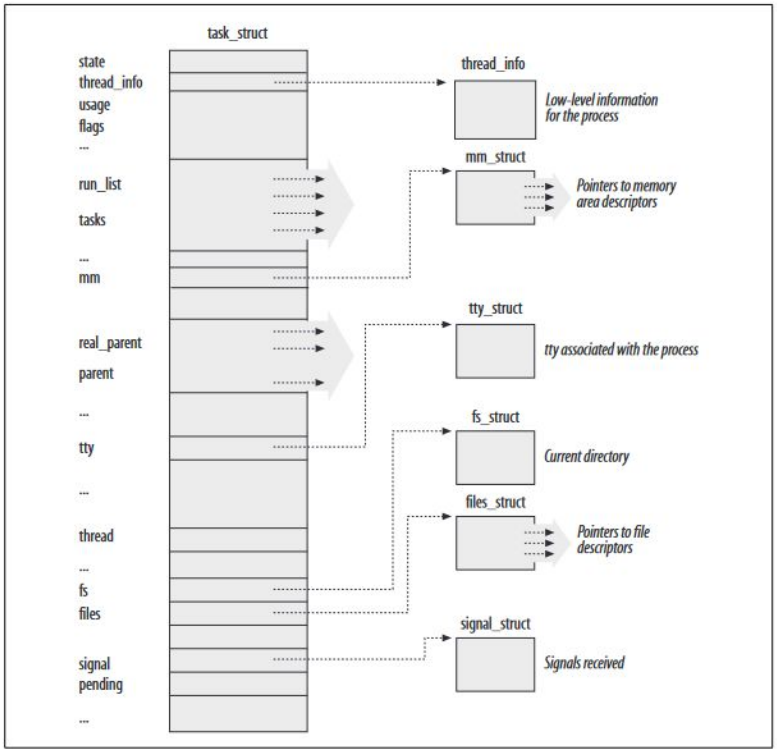



*Figure 3-1. The Linux process descriptor*

# Process organization - Wait queues

Wait queues:

- have several uses in the kernel, particularly for interrupt handling, process synchronization, and timing.
- implement conditional waits on events: a process wishing to wait for a specific event places itself in the proper wait queue and relinquishes control
- implemented as doubly linked lists
    - elements include pointers to process descriptors
    - identified by a wait queue head, a data structure of type wait_queue_head_t

Sleeping processes:

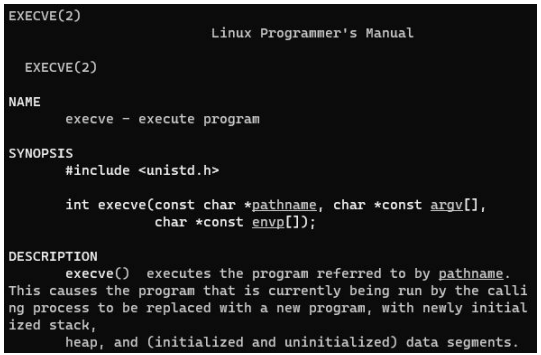
- waiting for some event to occur
- e**xclusive processes** (denoted by the value 1 in the flags field of the corresponding wait queue element) are selectively woken up by the kernel
- **nonexclusive processes** (denoted by the value 0 in the flags field) are always woken up by the kernel when the event occurs

The kernel awakens processes in the wait queues, putting them in the TASK_RUNNING state, by means of one of kernel code macros.

# Creating Processes

Traditional Unix systems:

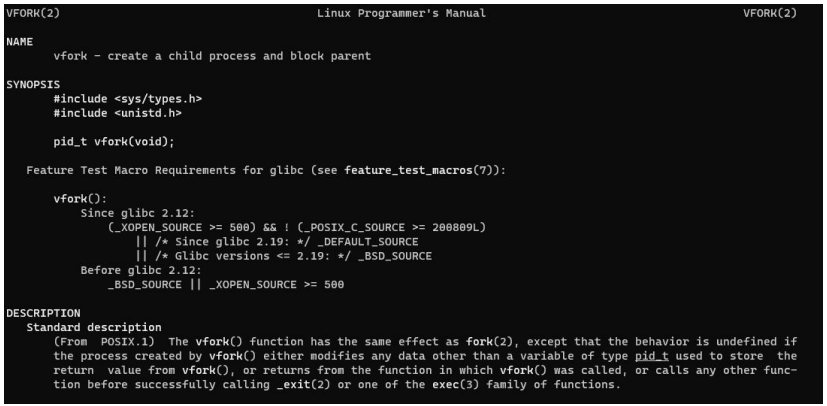
- treat all processes in the same way
- resources owned by the parent process are duplicated in the child process
  - process creation very slow and inefficient,
  - requires copying the entire address space of the parent process
- child process rarely needs to read or modify inherited resources
  - it issues an immediate **execve( )**
  - wipes out the address space that was so carefully copied



```
EXECVE(2)                    Linux Programmer's Manual

  EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *pathname, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve()  executes the program referred to by pathname.
This causes the program that is currently being run by the calli
ng process to be replaced with a new program, with newly initial
ized stack,
       heap, and (initialized and uninitialized) data segments.
```

Modern Unix kernels:

- The Copy On Write technique allows both the parent and the child to read the same physical pages
  - When one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process
- LWP allow both the parent and the child to share many per process kernel data structures
- The **vfork( )** system call creates a process that shares the memory address space of its parent
  - prevents parent from overwriting data of child
  - the parent's execution is blocked until the child exits or executes a new program.



```
VFORK(2)                     Linux Programmer's Manual                     VFORK(2)

NAME
       vfork - create a child process and block parent

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t vfork(void);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       vfork():
           Since glibc 2.12:
               (_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
                   || /* Since glibc 2.19: */ _DEFAULT_SOURCE
                   || /* Glibc versions <= 2.19: */ _BSD_SOURCE
           Before glibc 2.12:
               _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION
   Standard description
       (From  POSIX.1)  The vfork() function has the same effect as fork(2), except that the behavior is undefined if
       the process created by vfork() either modifies any data other than a variable of type pid_t used to store  the
       return  value from vfork(), or returns from the function in which vfork() was called, or calls any other func-
       tion before successfully calling _exit(2) or one of the exec(3) family of functions.
```

# Kernel Threads

- Delegate critical tasks to intermittently running processes
  - flushing disk caches, swapping out unused pages, servicing network connections, and so on
  - not efficient to perform these tasks in strict linear fashion
  - better response if scheduled in the background.
- In Linux, kernel threads differ from regular processes in the following ways:
  - Kernel threads run only in Kernel Mode, while regular processes run alternatively in Kernel Mode and in User Mode.
  - kernel threads use only linear addresses greater than PAGE_OFFSET (Virtual start address of the first bank of RAM)

```
ps --ppid 2 -p 2 -o uname,pid,ppid,cmd,cls
```

Examples of kernel threads (parent pid is 2) are:

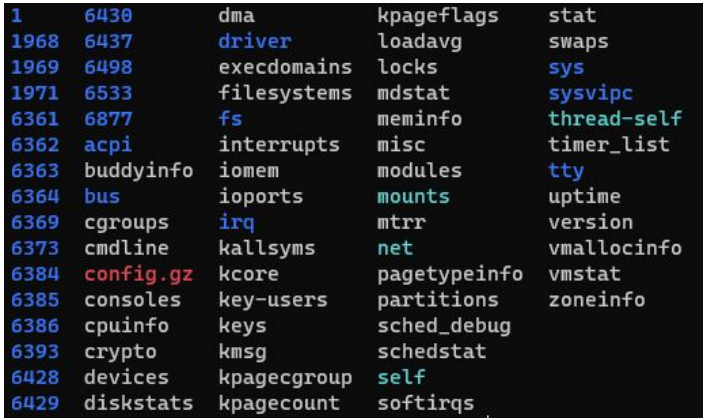
- **keventd** (also called events) Executes the functions in the keventd_wq workqueue
- **kapmd** Handles the events related to the Advanced Power Management (APM).
- **kswapd** Reclaims memory
- **pdflush** Flushes “dirty” buffers to disk to reclaim memory
- **kblockd** periodically activates the block device drivers
- **ksoftirqd** Runs the tasklets; one for each CPU in the system

https://unix.stackexchange.com/questions/411159/linux-is-it-possible-to-see-only-kernel-space-threads-process

# /proc - process information pseudo-file system

/proc is a virtual filesystem:

- a control and information centre for the kernel
- read/change kernel parameters (sysctl) while the system is running
- files in this directory is the fact that all of them have a file size of 0, with the exception of kcore, mtrr and self
- sometimes referred to as a process information pseudo-file system
- Doesn't contain 'real' files but runtime system information




A lot of system utilities are simply calls to files in /proc:

- 'lsmod' is the same as 'cat /proc/modules'
- 'lspci' is a synonym for 'cat /proc/pci'

https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html

# Information about processes

- /proc/PID/cmdline Command line arguments.
- /proc/PID/cwd Link to the current working directory.
- /proc/PID/environ Values of environment variables.
- /proc/PID/exe Link to the executable of this process.
- /proc/PID/fd Directory, which contains all file descriptors.
- /proc/PID/maps Memory maps to executables and libraries.
- /proc/PID/mem Memory held by this process.
- /proc/PID/root Link to the root directory of this process.
- /proc/PID/stat Process status.
- /proc/PID/statm Process memory status information.
- /proc/PID/status Process status in human readable form.

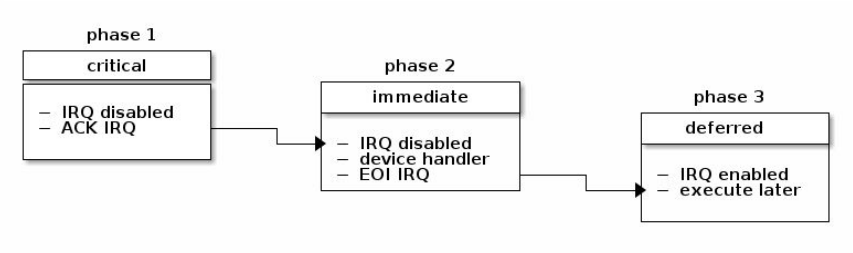
```
(base) alexandru@DESKTOP-TKPML3T:~/os$ sudo cat /proc/6373/cmdli
ne
/home/alexandru/.vscode-server/bin/74b1f979648cc44d385a2286793c2
26e611f59e7/node/home/alexandru/.vscode-server/bin/74b1f979648cc
44d385a2286793c226e611f59e7/out/server-main.js--host=127.0.0.1--
port=0--connection-token=1891230501-3244511119-3775249901-366097
3238--use-host-proxy--without-browser-env-var--disable-websocket
-compression--accept-server-license-terms--telemetry-level=all(b
ase) alexandru@DESKTOP-TKPML3T:~/os$
```

```
Name:   sh
Umask:  0022
State:  S (sleeping)
Tgid:   6389
Ngid:   0
Pid:    6389
PPid:   6364
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 64
Groups: 4 20 24 25 27 29 30 44 46 117 119 1000
NStgid: 6369
NSpid:  6369
NSpgid: 6363
NSsid:  6363
VmPeak:     2612 kB
VmSize:     2612 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:       592 kB
VmRSS:       592 kB
RssAnon:             68 kB
RssFile:            524 kB
RssShmem:             0 kB
VmData:      184 kB
VmStk:       132 kB
VmExe:        76 kB
VmLib:      1648 kB
VmPTE:        44 kB
VmSwap:        0 kB
HugetlbPages:          0 kB
CoreDumping:    0
THP_enabled:    1
Threads:        1
SigQ:   0/24789
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000200000000
SigCgt: 0000000000010002
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 000001ffffffffff
CapAmb: 0000000000000000
NoNewPrivs:     0
Seccomp:        0
Seccomp_filters:        0
Speculation_Store_Bypass:       thread vulnerable
Cpus_allowed:   ff
```

# Interrupt handling in Linux

Three phases: critical, immediate and deferred.

- critical
  - the kernel will run the generic interrupt handler that determines the interrupt number, the interrupt handler for this particular interrupt and the interrupt controller
  - critical actions will also be performed (e.g. acknowledge the interrupt at the interrupt controller level).
  - Local processor interrupts are disabled
- immediate
  - device driver's handlers will be executed
  - the interrupt controller's "end of interrupt" method is called to allow the interrupt controller to reassert this interrupt
  - local processor interrupts are enabled at this point
- deferred
  - avoid doing too much work in the interrupt handler function
  - "bottom half" of the interrupt (the upper half being the part of the interrupt handling that runs with interrupts disabled).
  - interrupts are enabled on the local processor

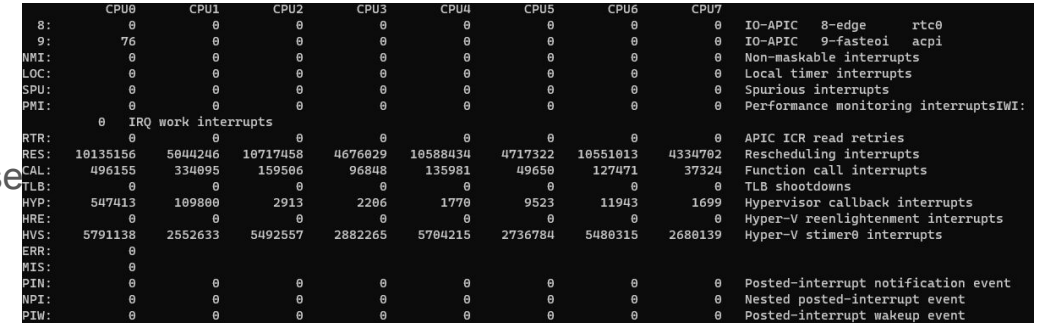



https://linux-kernel-labs.github.io/refs/heads/master/lectures/interrupts.html#interrupt-handling-in-linux

# Information about interrupts

/proc/interrupts

- Shows which interrupts are in use, and how many of each there have been
- Check which interrupts are currently in use and what they are used for
- smp_affinity
  - used to set IRQ to CPU affinity
  - "hook" an IRQ to only one CPU, or to exclude a CPU from handling IRQs.



```
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  8:          0          0          0          0          0          0          0          0   IO-APIC   8-edge      rtc0
  9:         76          0          0          0          0          0          0          0   IO-APIC   9-fasteoi   acpi
NMI:          0          0          0          0          0          0          0          0   Non-maskable interrupts
LOC:          0          0          0          0          0          0          0          0   Local timer interrupts
SPU:          0          0          0          0          0          0          0          0   Spurious interrupts
PMI:          0          0          0          0          0          0          0          0   Performance monitoring interruptsIWI:
          0   IRQ work interrupts
RTR:          0          0          0          0          0          0          0          0   APIC ICR read retries
RES:   10135156    5044246   10717458    4676029   10588434    4717322   10551013    4334702   Rescheduling interrupts
CAL:     496155     334095     159506      96848     135981      49650     127471      37324   Function call interrupts
TLB:          0          0          0          0          0          0          0          0   TLB shootdowns
HYP:     547413     109800       2913       2206       1770       9523      11943       1699   Hypervisor callback interrupts
HRE:          0          0          0          0          0          0          0          0   Hyper-V reenlightenment interrupts
HVS:    5791138    2552633    5492557    2882265    5704215    2736784    5480315    2680139   Hyper-V stimer0 interrupts
ERR:          0
MIS:          0
PIN:          0          0          0          0          0          0          0          0   Posted-interrupt notification event
NPI:          0          0          0          0          0          0          0          0   Nested posted-interrupt event
PIW:          0          0          0          0          0          0          0          0   Posted-interrupt wakeup event
```

```
(base) alexandru@DESKTOP-TKPML3T:~/os$ ls /proc/irq/
0  1  10  11  12  13  14  15  2  3  4  5  6  7  8  9  default_smp_affinity
(base) alexandru@DESKTOP-TKPML3T:~/os$ cat /proc/irq/10/smp_affinity
ff
```