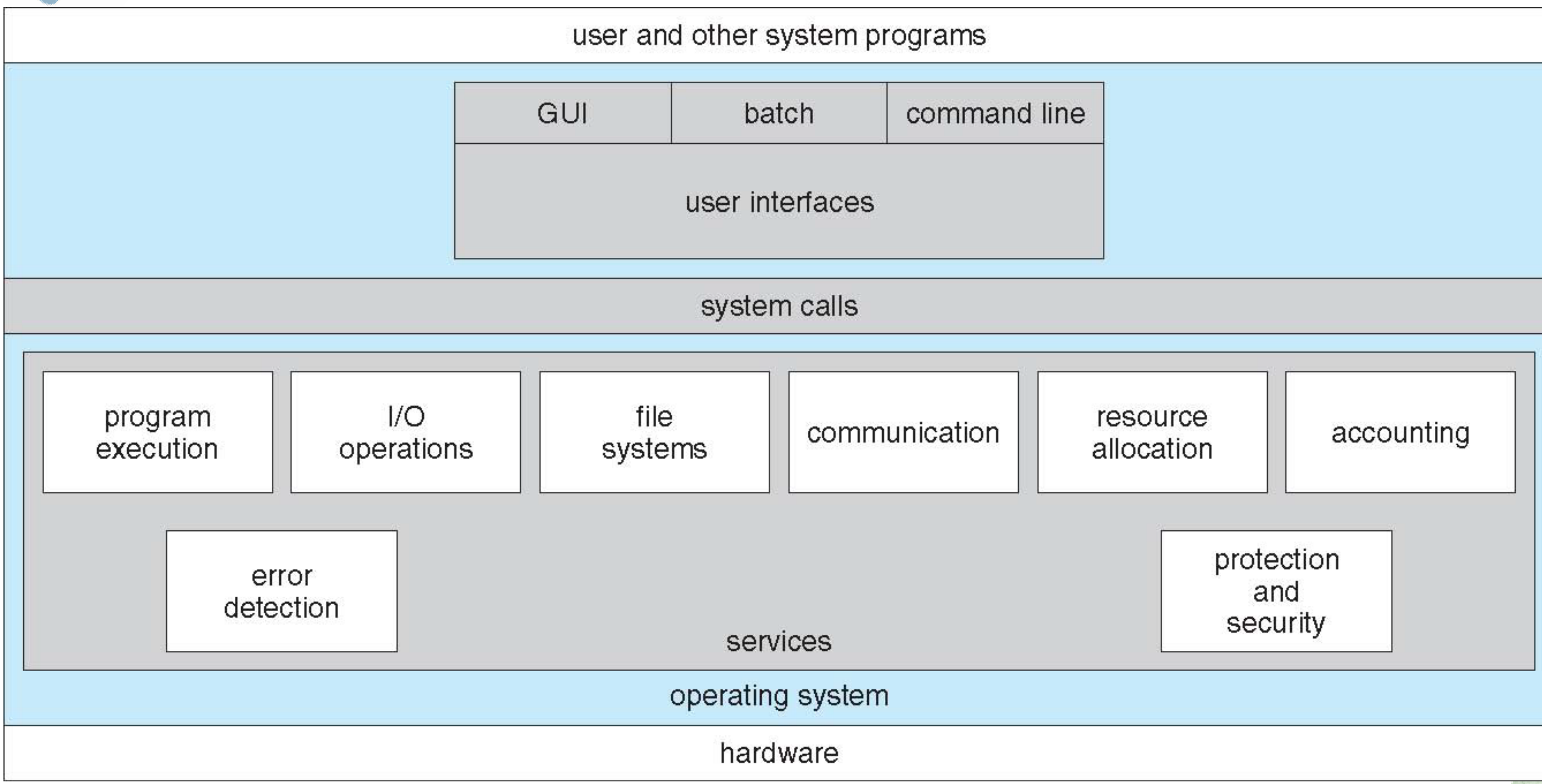
Abstraction: Process (cont.)

Objectives

- How are processes represented and controlled by the OS.
- ***Process states*** which characterize the behaviour of processes.
- ***Data structures*** used to manage processes.
- Ways in which the OS uses these data structures to control process execution.

Operating System Services



Types of System Calls

- Process control
  - load, execute; create process, terminate process; get process attributes, set process attributes; wait for time, wait for events; allocate and free memory; end, abort;
- File management
  - …
- Device management
  - …
- Information maintenance
  - …
- Communications
  - …

Dual Mode Operation

- If every program had unfettered access to system resources, a program could look at all memory locations, including that of other programs, as well as read all the data on all of the attached disks
- Dual-mode operation is then designed to provide a layer of protection and stability to computer systems by separating user programs and the operating system into two modes: user mode and kernel mode
- User mode restricts access to privileged resources, while kernel mode has full access to these resources

Dual Mode Operation

User Mode

- user program executes in user mode
- certain areas of memory are protected from user access
- certain instructions may not be executed

Kernel Mode

- OS executes in kernel mode
- privileged instructions may be executed
- protected areas of memory may be accessed

Dual Mode Operation

- The advantages of dual mode operation include
  - provides a layer of protection between user programs and OS to protect the system from unauthorized and malicious access
  - Reduce the risk of malware attacks or other security threats as user programs in user mode cannot modify system data or perform privileged operations
  - Ensure system-level operations are performed correctly and reliably
  - Prevent user programs from accidentally or maliciously causing system crashes or other errors

System Calls

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

System Calls

- It should be obvious that we don't want user-mode programs to easily switch to kernel mode
  - that would make the two privilege levels useless
- However, a user-mode program does need to cross into kernel mode
  - E.g., when a program wants to read from disk
- We need a mechanism whereby
  - a user-mode program can switch into kernel mode
  - but have no control over the instructions which will be performed in kernel mode

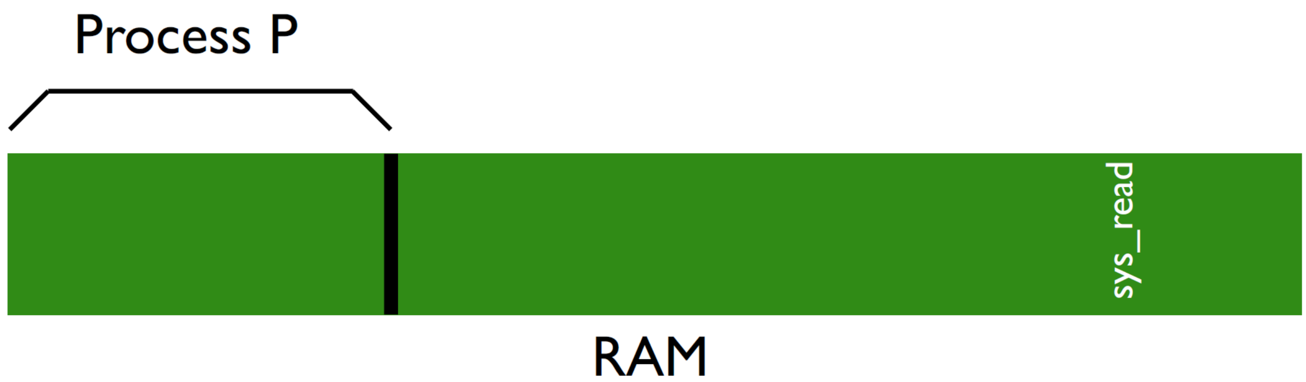
System Calls

- The user program
  - sets up the parameters
  - traps to the kernel
    - Different architectures have different trap instructions, e.g., int on x86
- System call enters the kernel at a fixed location
  - Switches the stack pointer SP to the kernel stack
  - Saves the user SP value
  - Saves the user PC value (= return address)
  - Saves the user privilege mode
  - Sets the kernel privilege mode
  - Sets the new PC to the system call handler

System Calls

- Kernel system call handler carries out the desired system call
  - Saves user registers
  - Examines the system call number
  - Checks arguments
  - Performs operation
  - Stores result
  - Performs a “return from system call” instruction
    - Restores user privilege mode, SP and PC and registers if returning directly to the user process which made the call
    - Or go through the dispatcher to select the next process to run
- All memory used by the system call handler is in kernel space, so it is protected from interference by the user program

System Calls



P wants to call read()

System Calls

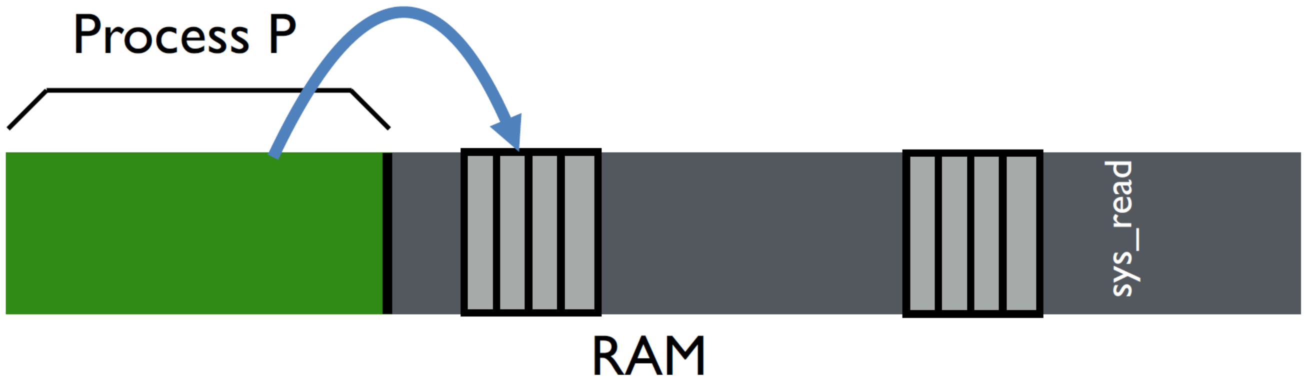
P can only see its own memory because of **user mode** (other areas, including kernel, are hidden)

P wants to call read() but no way to call it directly

System Calls

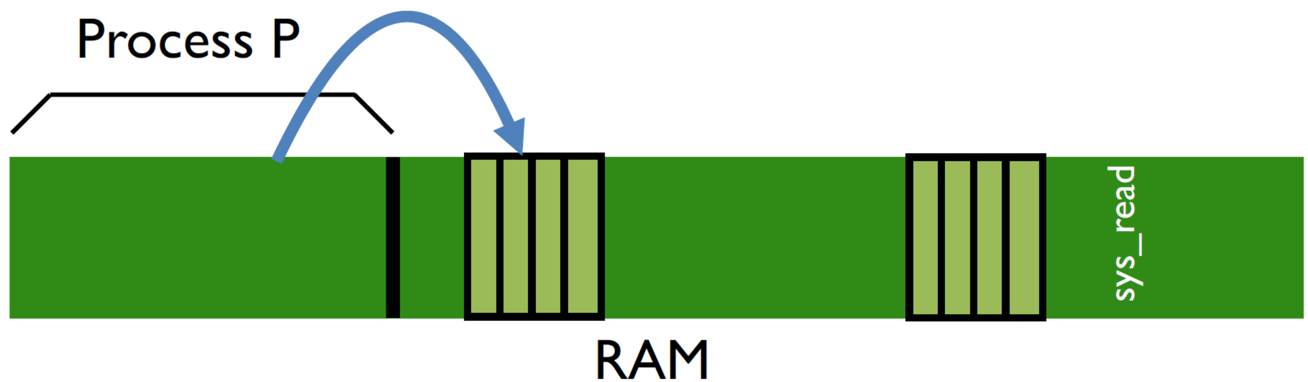

```
movl $6, %eax;    int $64
```

System Calls




```
movl $6, %eax;    int $64
```

System Calls

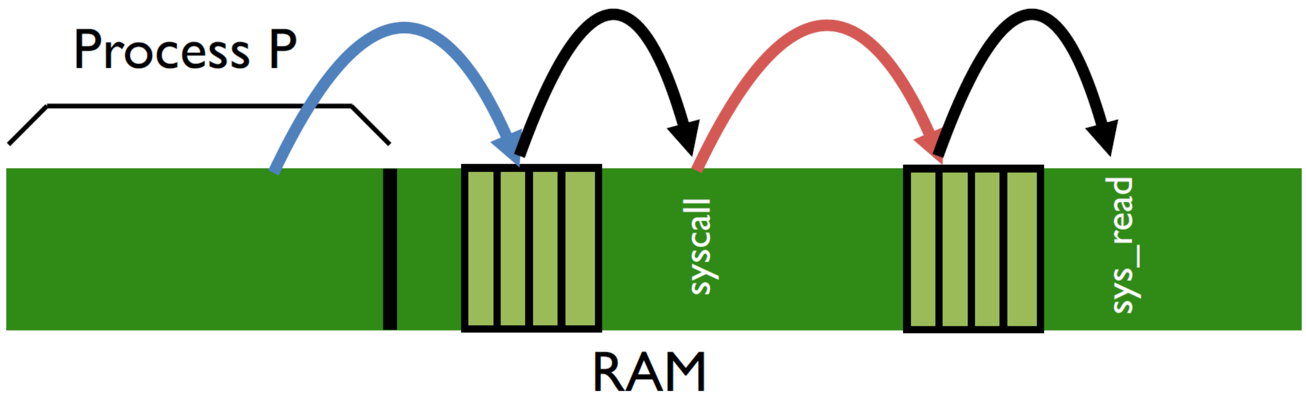


Syscall table index

```
movl $6, %eax;    int $64
```

Trap table index

System Calls



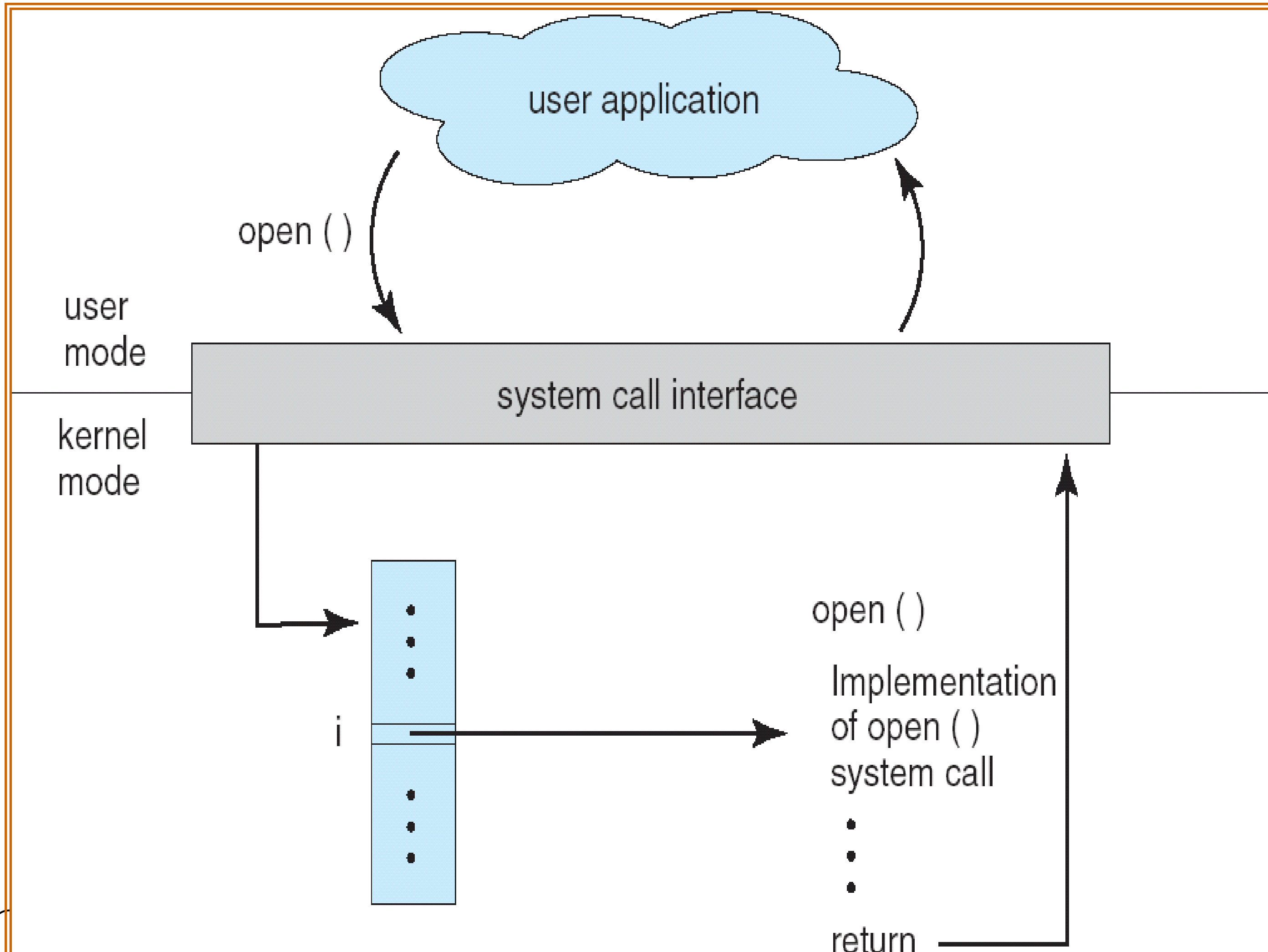

```
movl $6, %eax;    int $64
```

Follow entries to correct system call code

System Calls

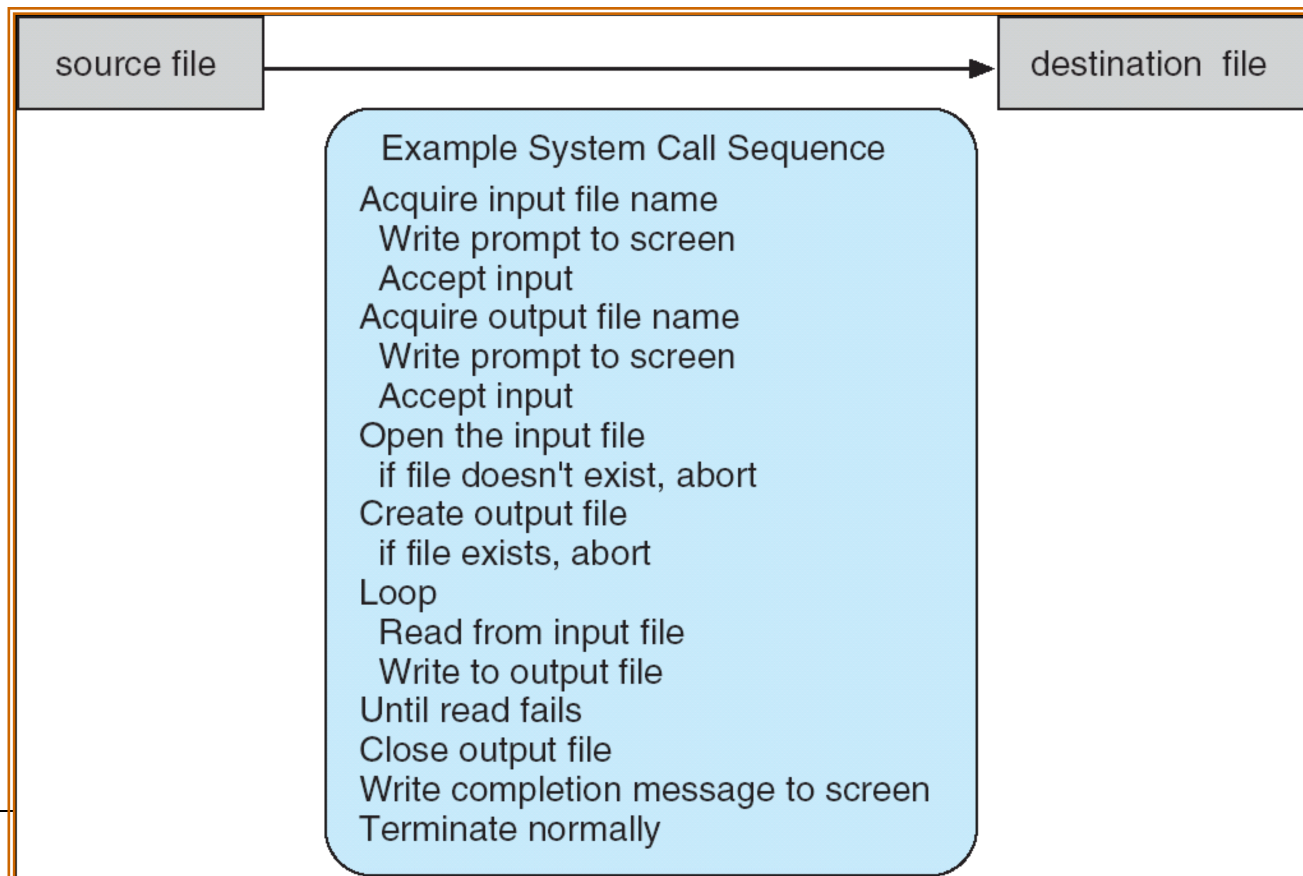
- System calls are mostly accessed by user programs via a high-level Application Programming Interface (API) rather than direct system call use
  - E.g., C library implements standard API from kernel system call interface

API – System Call – OS Relationship



Example of System Calls

- System call sequence to copy the contents of one file to another file



Processes and Threads

- Processes have two characteristics:
  - **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 scheduled and dispatched by the OS
- These two characteristics are treated independently by modern operating systems:
  - 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***

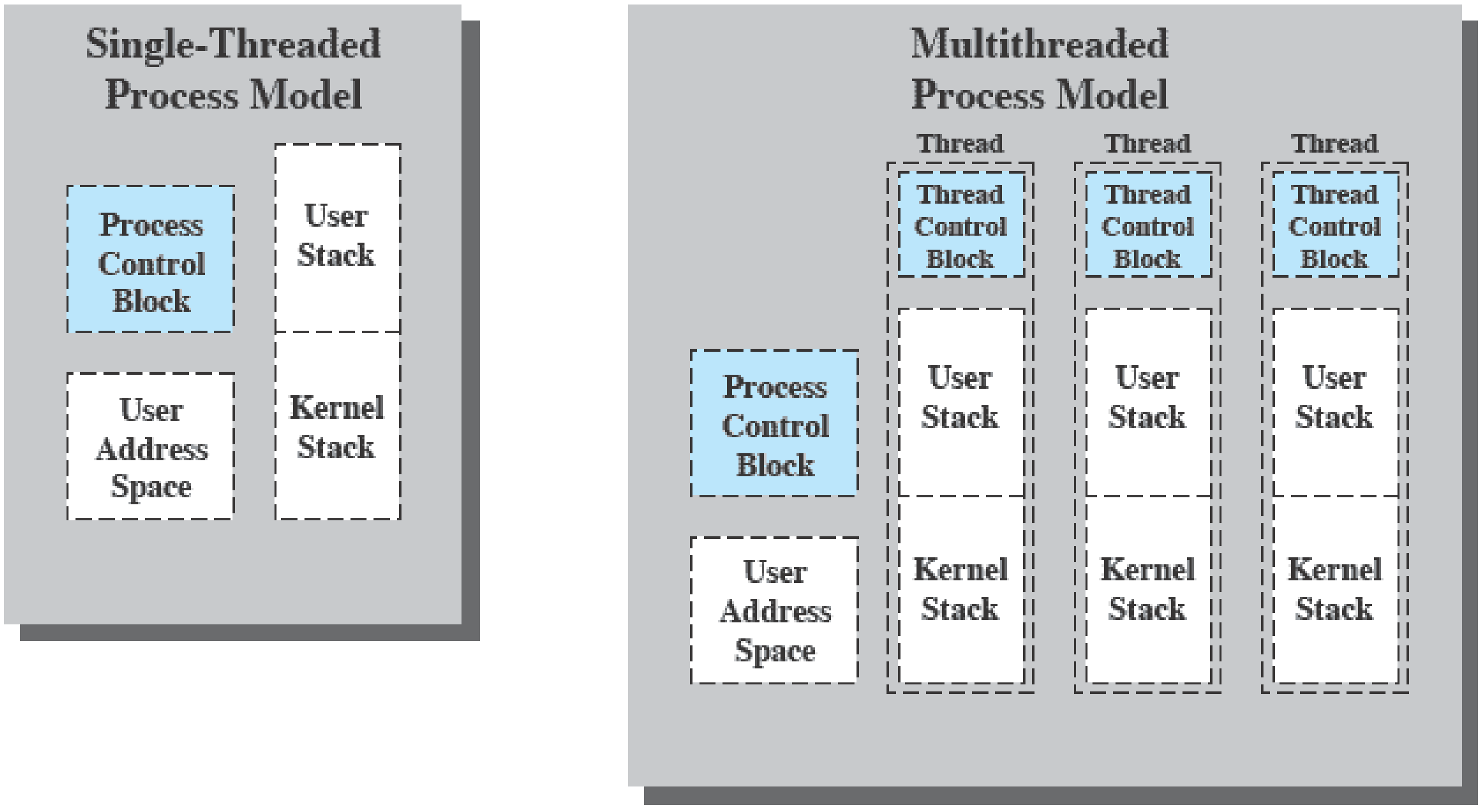
Process

- The unit of resource allocation and a unit of protection
- A process is associated with
  - A virtual address space which holds the process image
  - Protected access to
    - Processors,
    - Other processes,
    - Files,
    - I/O resources

Multiple Threads in Process

- Each thread has
  - Access to the memory and resources of its process (all threads of a process share this)
  - An execution state (running, ready, etc.)
  - Saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables

Threads vs. processes



Thread Synchronization

- It is necessary to synchronize the activities of the various threads in a process so that they do not interfere with each other
  - 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
  - Synchronization needed to coordinate execution of multiple threads

Linux Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process (or task) that happens to share the same address space as its parent
- A distinction is only made when a new thread is created by the `clone` system call
  - `fork` creates a new process with its own entirely new process context
  - `clone` creates a new process with its own identity, but that is allowed to share the data structures of its parent

Linux Threads

- Using `clone` gives an application fine-grained control over exactly what is shared between two threads

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Windows Threads

- Windows makes use of two types of process-related objects:
- Processes
  - an entity corresponding to a user job or application that owns resources
- Threads
  - a dispatchable unit of work that executes sequentially and is interruptible

Summary

- Operating System services
  - System calls
  - Dual mode to separate user-mode from kernel mode for security
  - Syscall: call kernel mode functions
    - Transfer from user-mode to kernel-mode (trap)
    - Return from kernel-mode to user-mode (return-from-trap)
- Threads
  - Process related to resource ownership
  - Thread related to program execution
  - Threads of a process share the same address space and resources
  - Any alteration of a resource by one thread affects the other threads in the same process