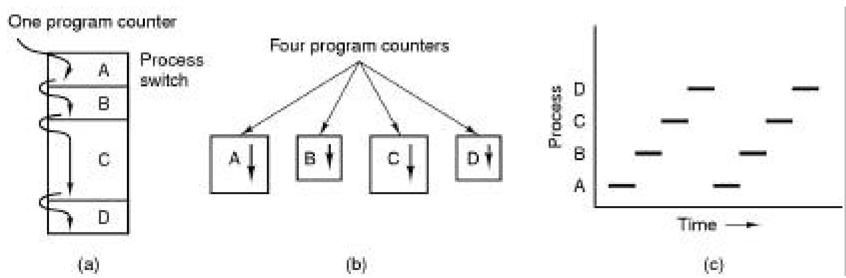
Lecture Overview

- Introduction to processes and threads
  - Process model
  - Process representation in the operating system
  - Thread overview
  - Thread implementations

What is a Process?

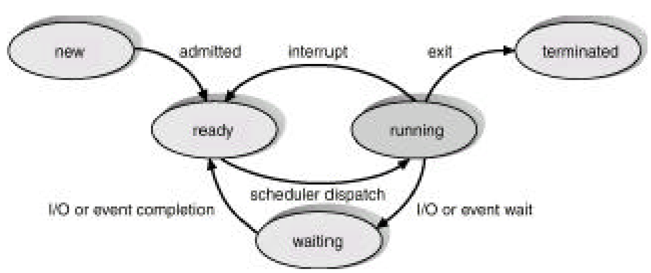
- A process is a unit of work in the operating system
- A process is an executing program, it has a sequential execution flow
- A process has associated resources (e.g., stack, registers, memory, open files, etc.)
- A process is conceptual, it does not exist at the hardware level

Process Model



a) Multiprogramming of four programs
b) Conceptual model of 4 independent, sequential processes
c) Only one program active at any instant, but each make progress

Process States & Transitions



- Process states
  - *new* = the process is being created
  - *running* = instructions are being executed
  - *waiting* = the process is waiting for some event to occur
  - *ready* = the process is waiting to be assigned to a process
  - *terminated* = the process has finished execution

Process Creation & Termination

- Principal events that cause process creation
  - System initialization
  - Execution of a process creation system
  - User request to create a new process
  - Initiation of a batch job
- Conditions that terminate processes
  - Normal exit (voluntary)
  - Error exit (voluntary)
  - Fatal error (involuntary)
  - Killed by another process (involuntary)

Process Hierarchies

- UNIX
  - All processes have a parent process that created it
    - Initial processes are children of the OS *init* process
  - A process may create its own child process and its children may create their own children and so on
    - This forms a *process group*
- Windows
  - There is no notion of parent/child processes

Process Descriptors

- Each process is represented in the operating system with a data structure called a *process descriptor* or *process control block*
- Generally, the OS stores all process descriptors in a data structure, such as a *process table*

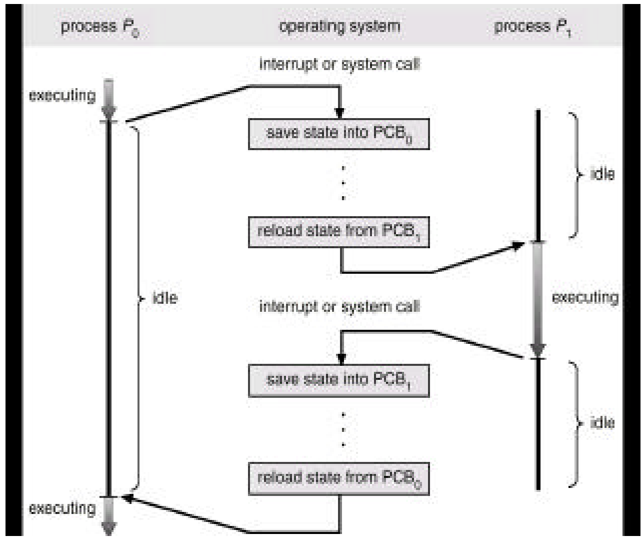
Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Examples of the type of information stored in a process descriptor

Process Switching

- When the OS switches to another process, it must save the state of the old process and load the saved state for the new process
  - This is called a *context switch*
- Context switch time is overhead; the system does no useful work while switching
- Context switching is time dependent on hardware support

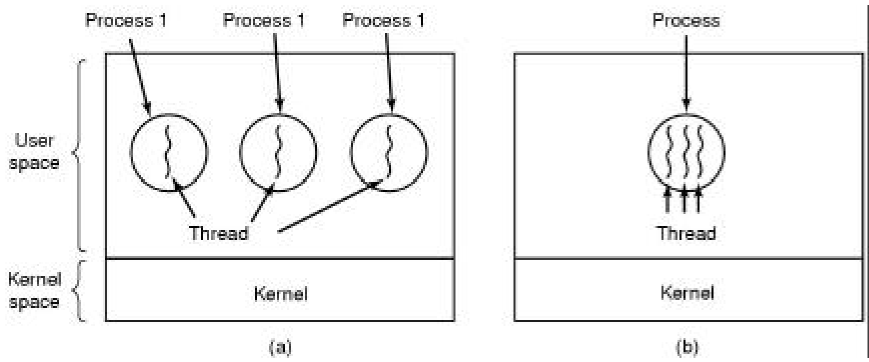
Process Switching



Threads

- We defined a process as having resources and a sequential execution flow
  - It is possible to separate these two notions
- A *thread* is a unit of sequential execution (another for thread is a *lightweight process*)
- With the concept of a thread, a *process* is merely a grouping mechanism or container for resources
- This distinction enables a process to have multiple threads of control, i.e., to be *multithreaded*

Threads



a) Three processes each with one thread (original process notion)
b) One process with three threads (multithreaded)

Threads

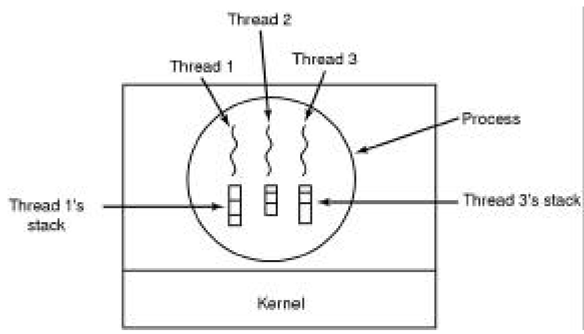
Unlike multiple process, multiple thread do not have all of their own resources, instead they *share* the resources of their parent process

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Threads

Each thread does have its own stack, because each has its own execution history

- As we saw in the x86 Assembler example, the stack is used for local variable storage as well as procedure call chains

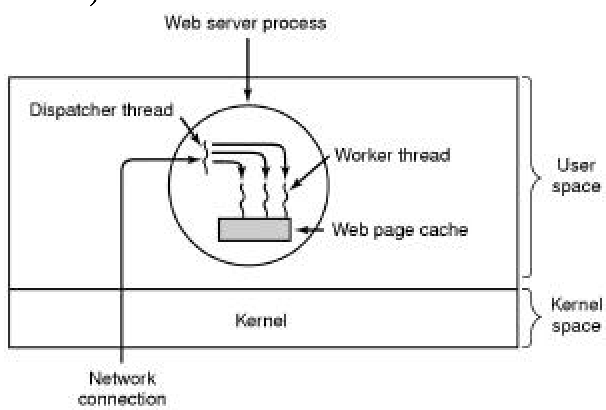


Why Do We Want Threads?

- Performance - multi-processor machines
- Responsiveness - user interfaces
- Efficiency - blocking calls
- Naturalness - related, but separate activity streams

Multithreaded Example

A Web server can service multiple requests at the same time with multiple threads of control, instead of having to block requests until the first one finishes (similar approaches are possible with multiple processes)



Multithreaded Example

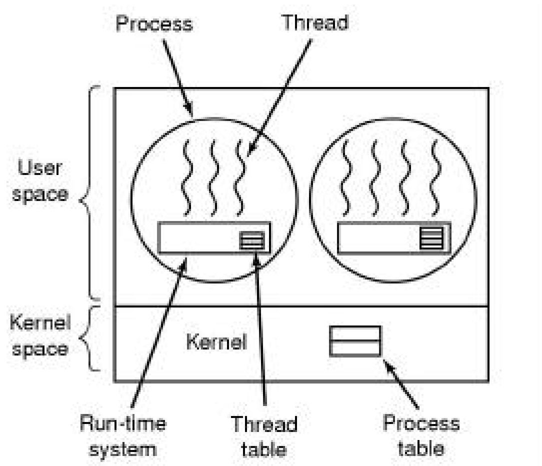
Using a single process, there are three ways to construct the example Web server, the most natural and effective is with multiple threads

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

A finite-state machine is actually a way to simulate multiple threads as we know from concurrent programming, but it is no longer a sequential process model and is more complicated to implement

Implementing Threads

Threads implemented in user space

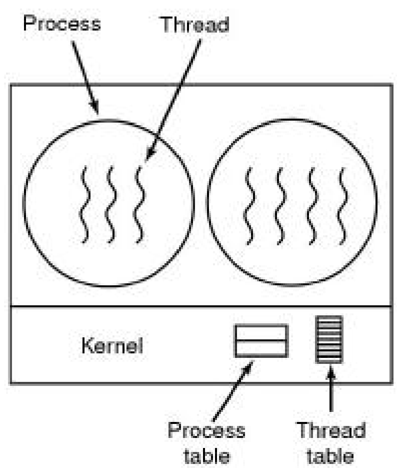


Implementing Threads

- Advantages of user space threads
  - Can be implemented on any [sufficient] operating system
  - Thread switching is fast
  - Thread scheduling is fast
  - Enable custom scheduling algorithms
  - Scale better
- Disadvantages of user space threads
  - Use a cooperative approach to sharing the CPU
  - Blocking system calls are problematic since one blocked thread will block all threads; the most useful place for threads is when using blocking systems calls

Implementing Threads

Threads implemented in kernel space



Implementing Threads

- Advantages of kernel space threads
  - No runtime system in each process
  - Kernel can easily reschedule threads when they block
- Disadvantages of kernel space threads
  - Thread switching is slower than user space threads
  - Thread scheduling is slower than user space threads

Difficulties with Multithreaded Programs

- Globally scoped data may cause conflicts if accessed by multiple threads at the same time
- Any libraries used by multithreaded programs must be reentrant, i.e., designed to allow multiple threads of control
- Complicates signal handling