Chapter 2 - (First Part)

Processes and Threads

Slide Credits:

Jonathan Walpole

Andrew Tanenbaum

Lecture overview

Processes

- Process Scheduler
- Process States
- Process Hierarchies
- Relevant Unix System Calls

Threads

- Comparison to Processes
- Examples
- User-Level Thread Package

Processes

A process is a program in execution.

Program

Description of how to perform an activity
Instructions and static data values

Process

A snapshot of a program in execution.

- **Memory**
  (Instructions, Data, Runtime Stack)
- **CPU state (Registers, PC, SP, etc.)**
- **Operating system state**
  (open files, accounting statistics, etc.)

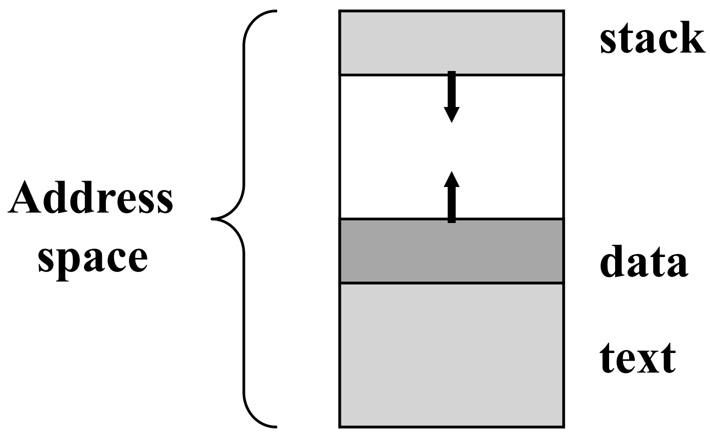
Virtual Address Space

or "Logical" Address Space

Each process runs in its own *virtual memory address space*

Which consists of...

- ***Text*** **– the program code (usually read-only)**
- ***Data space*** **– variables (initialized/uninitialized)**
- ***Stack space*** **– used for function calls**

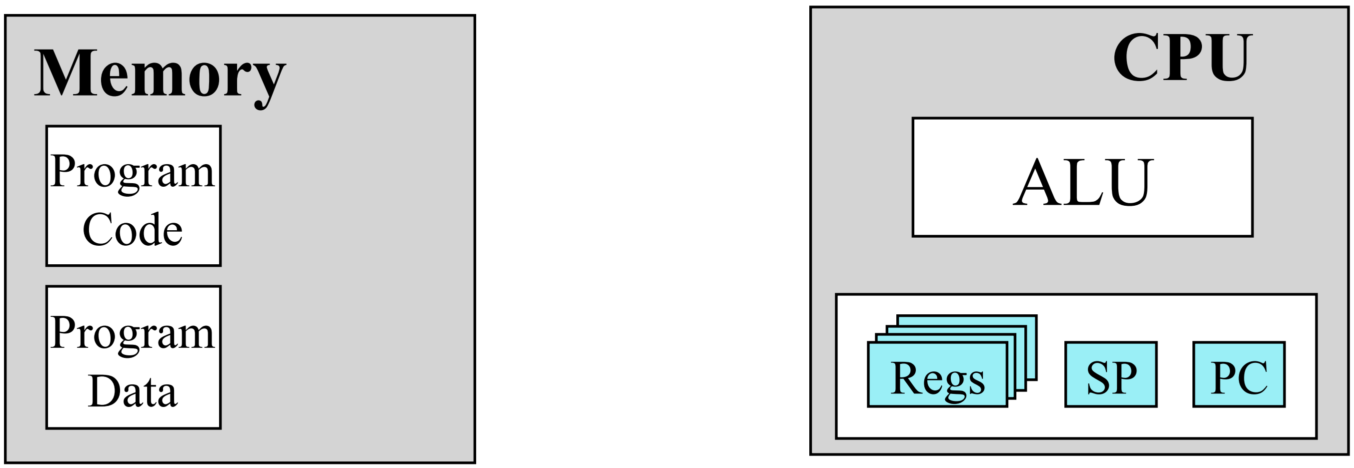


Invoke the same program multiple times?

... Results in the creation of multiple, distinct address spaces.

Process Switching

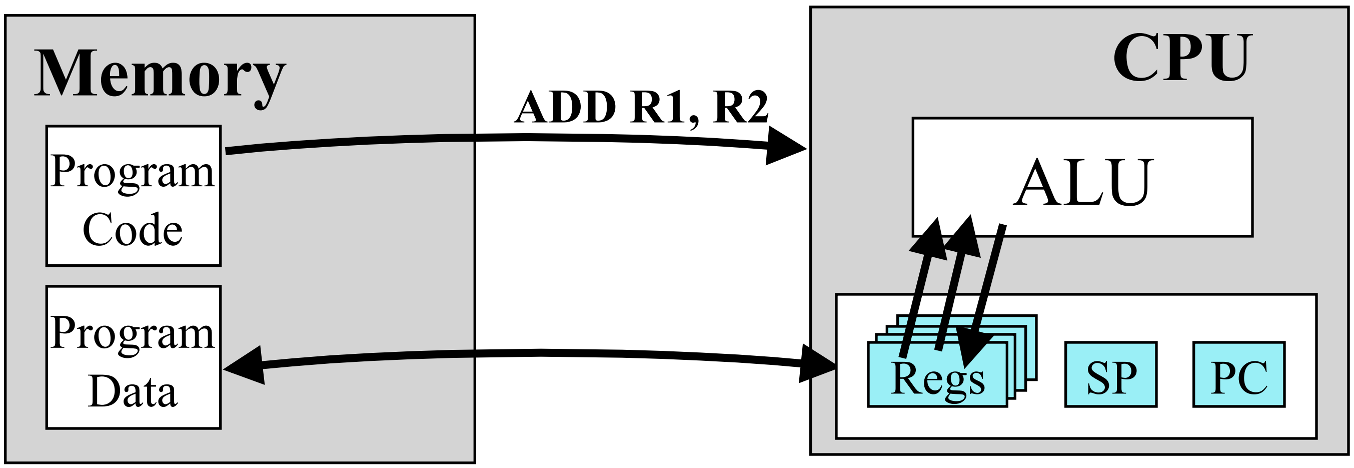
In its simplest form, a computer performs instructions on operands. Registers are used to hold values temporarily to speed things up.



Process Switching

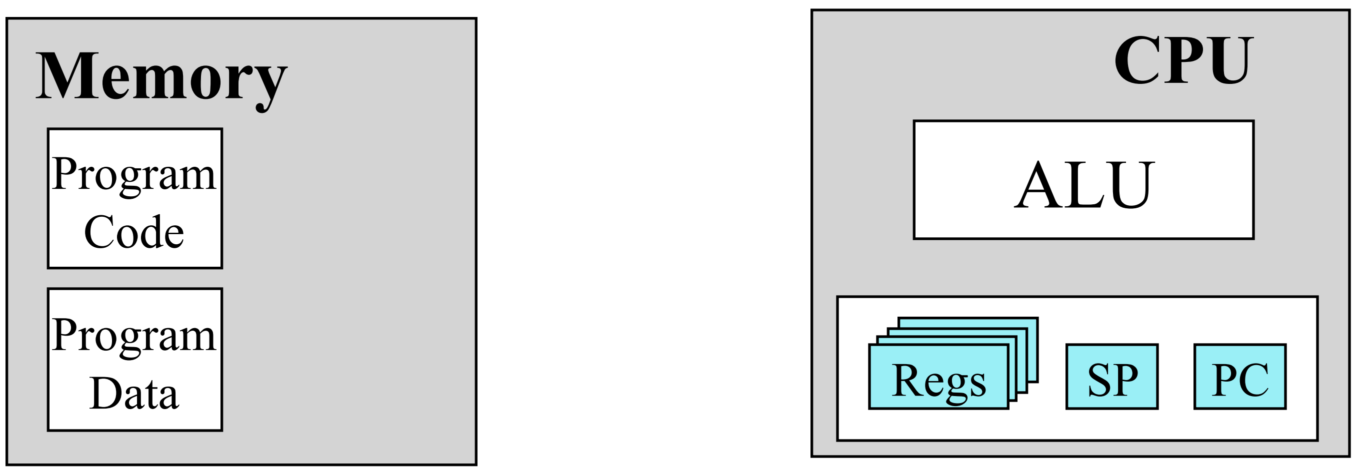
In its simplest form, a computer performs instructions on operands. Registers are used to hold values temporarily to speed things up.

Program 1 is running



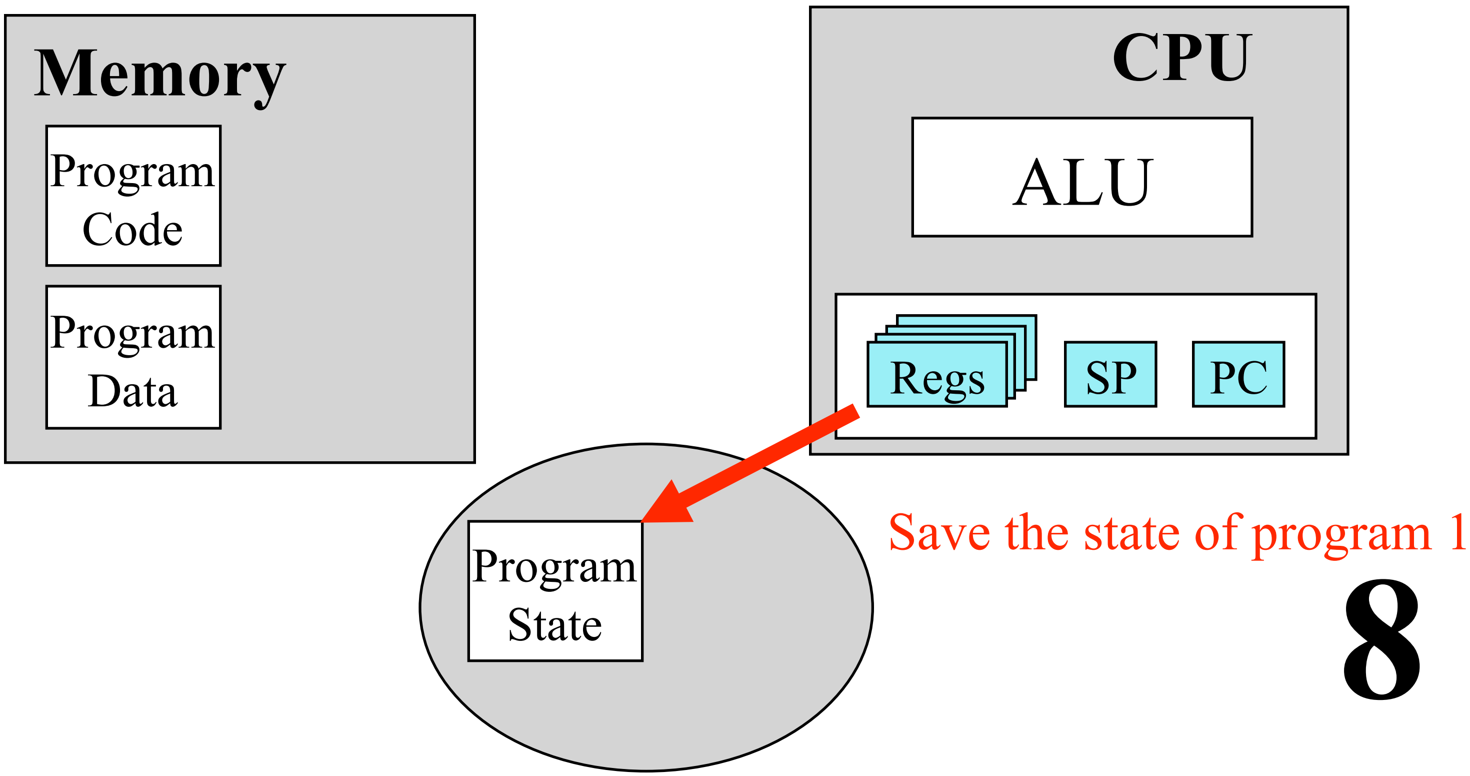
Process Switching

Saving all the information about a process allows a process to be *temporarily suspended.*



Process Switching

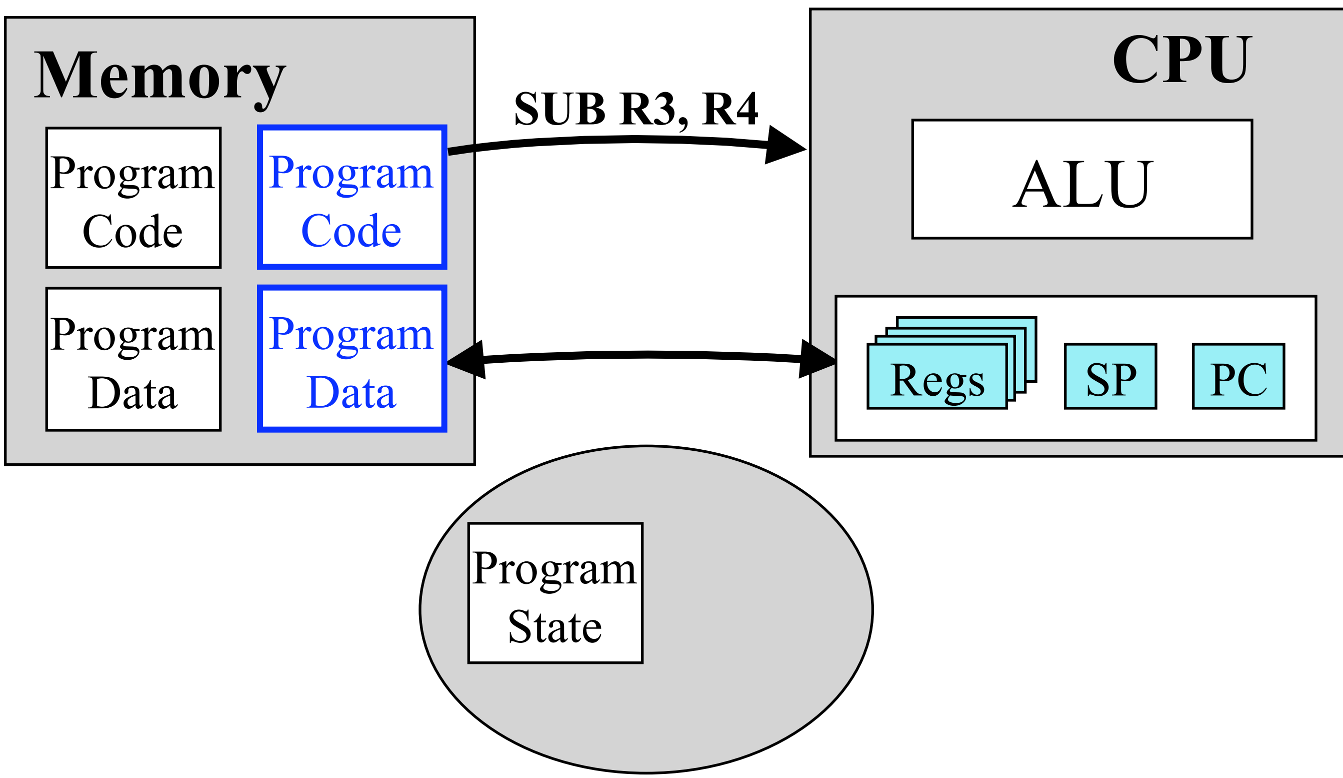
Saving all the information about a process allows a process to be *temporarily suspended.*



Process Switching

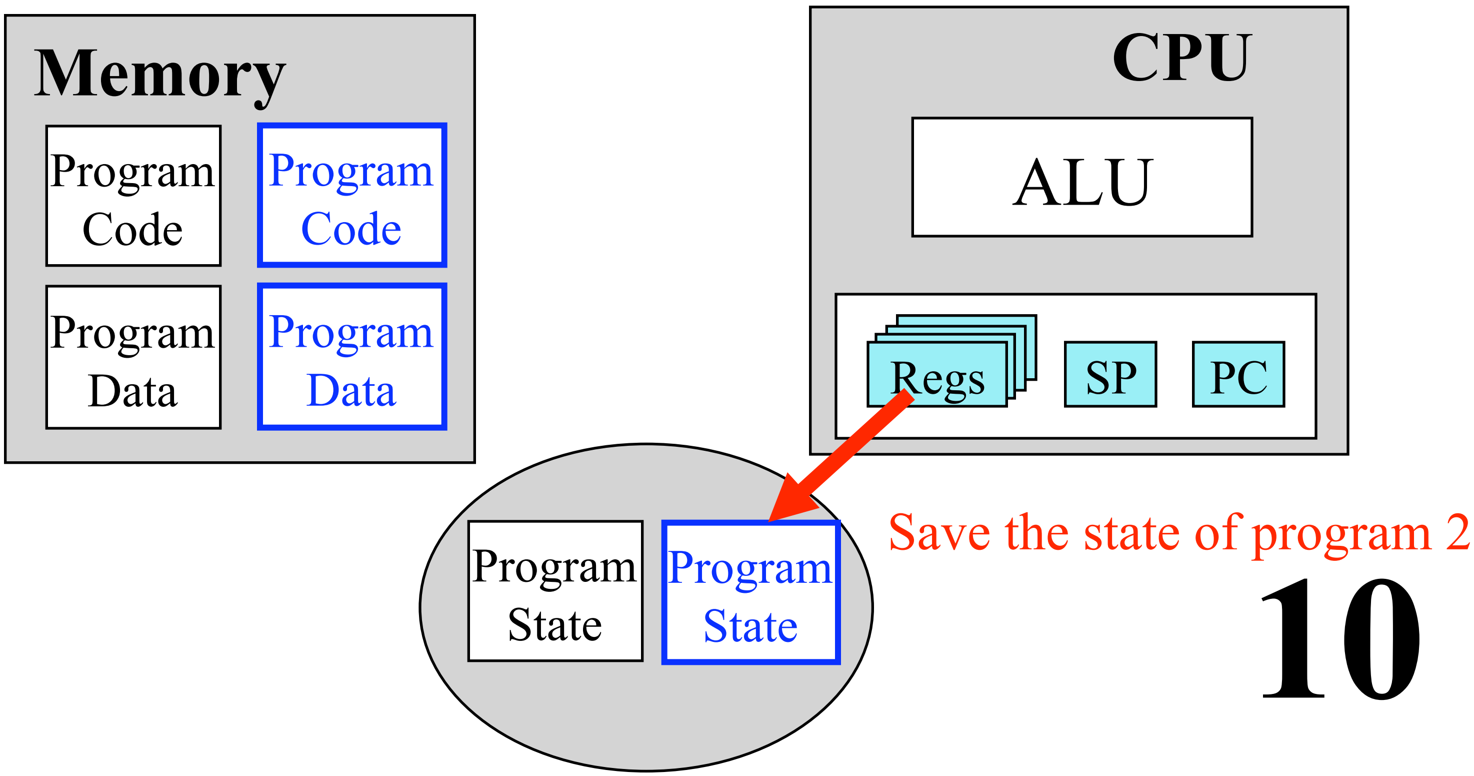
Saving all the information about a process allows a process to be *temporarily suspended.*

Program 2 now has the CPU



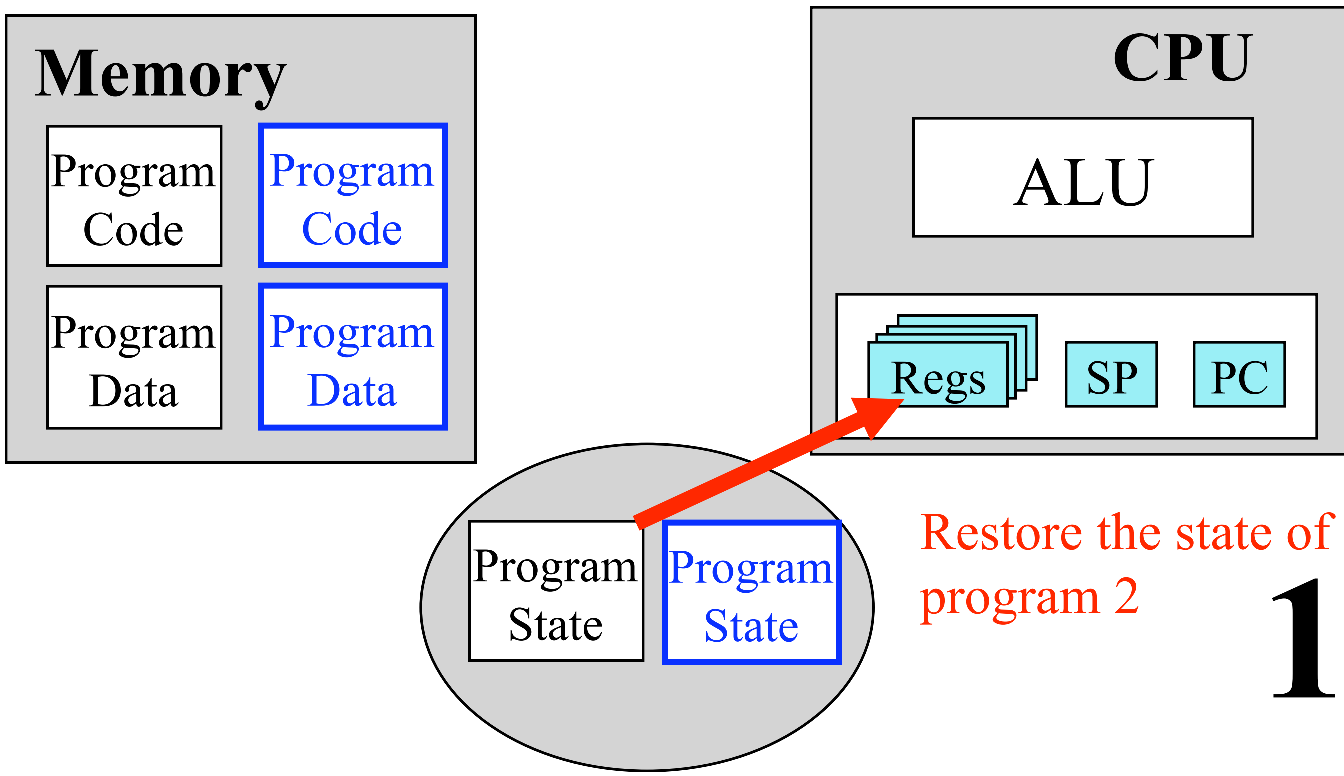
Process Switching

Saving all the information about a process allows a process to be *temporarily suspended.*



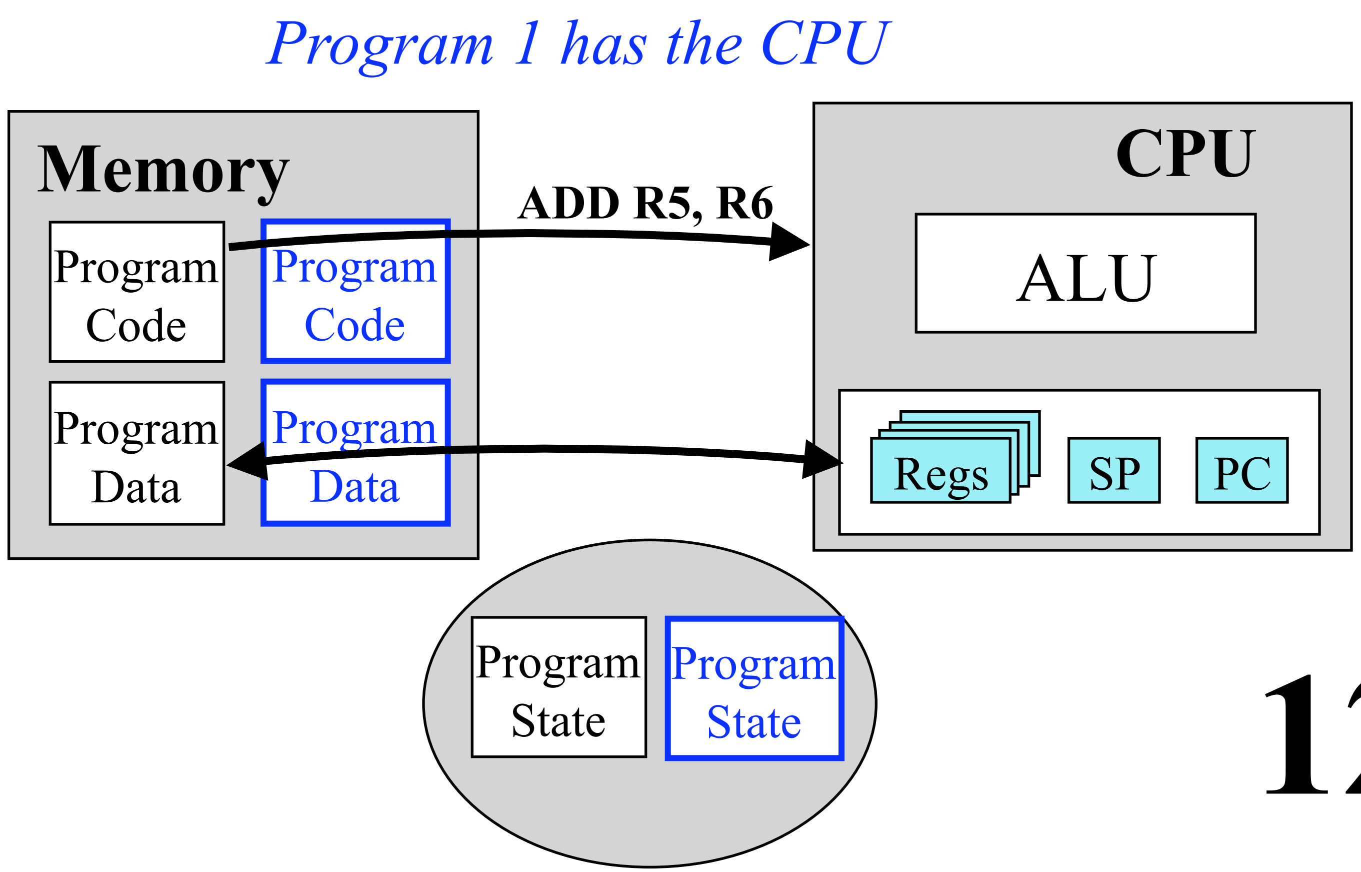
Process Switching

Saving all the information about a process allows a process to be *temporarily suspended.*

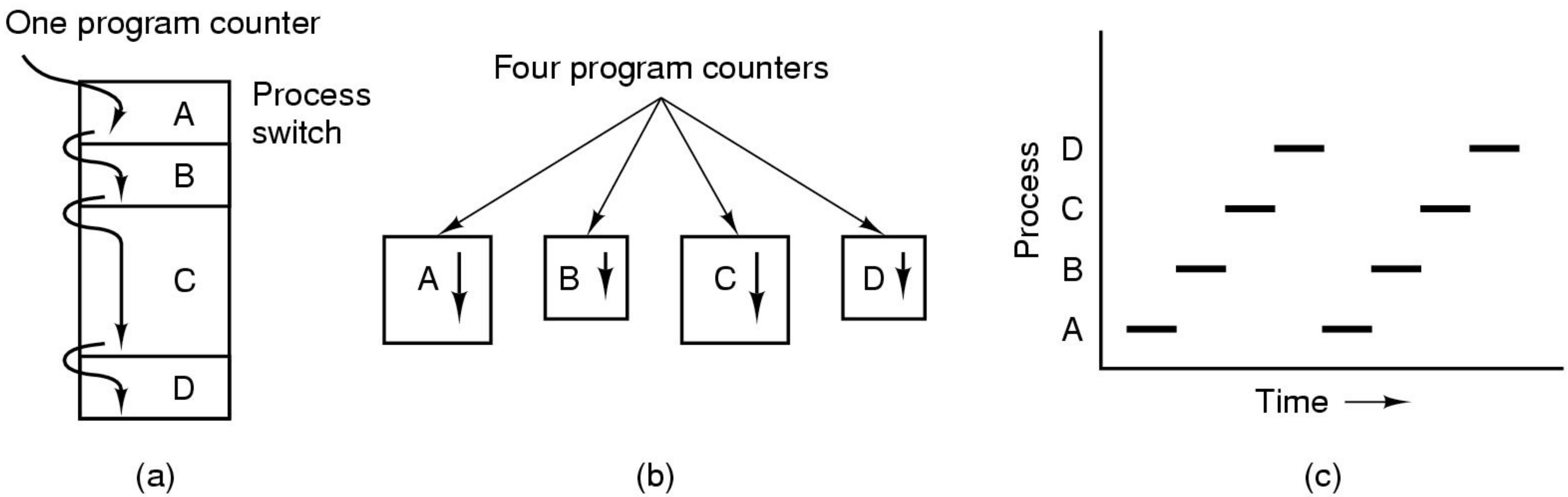


Process Switching

Saving all the information about a process allows a process to be *temporarily suspended.*

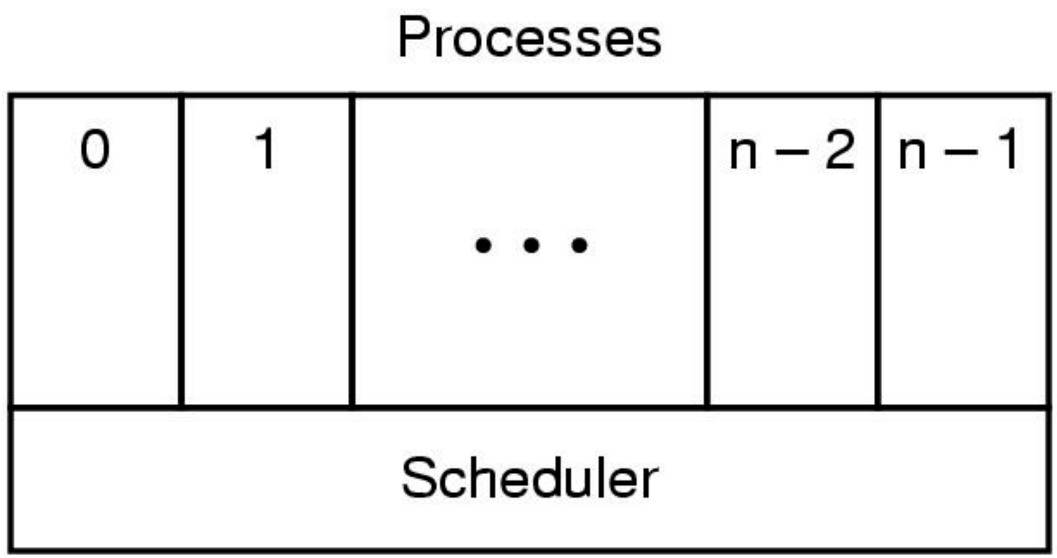


Why use the process abstraction?



- **Multiprogramming of four programs in the same address space**
- **Conceptual model of 4 independent, *sequential processes***
- **Only one program is active at any instant**

The Role of the Scheduler

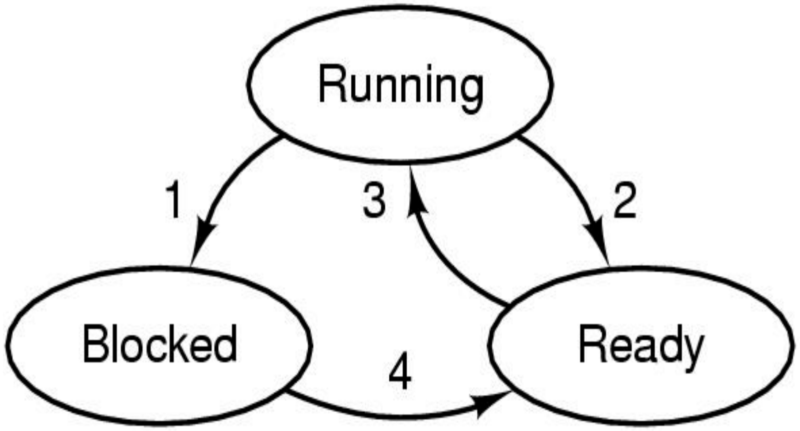


Lowest layer of process-structured OS

handles interrupts & scheduling of processes

Above that layer are sequential processes

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Possible states of a process:

RUNNING

BLOCKED

READY

Implementation of process switching

Skeleton of what the lowest levels of the OS do when an interrupt occurs

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

How Can Processes Be Created?

Events that create processes...

- **System initialization**
- **Initiation of a batch job**
- **Execution of a “process creation” system call**
  (from another process)
- **User request to create a new process**

Process Hierarchies

Parent creates a child process.

Special system calls for communicating with and waiting for child processes

Each process is assigned a unique identifying number

The "Process ID" or "pid".

Child processes can create their own child processes.

Forms a hierarchy

UNIX calls this a "Process Group"

Windows has no concept of process hierarchy.

"All processes are created equal."

How do Processes Terminate?

Conditions which terminate processes...

- **Normal exit (voluntary)**
- **Error exit (voluntary)**
- **Fatal error (involuntary)**
- **Killed by another process (involuntary)**

Process creation in UNIX

All processes have a unique process id

- getpid(), getppid() allow processes to get their information

Process creation

- fork() creates an exact copy of the process
  - identical with exception of the return value of fork()
- exec() replaces an address space with a new program
- system() like CreateProcess()

Process termination, signaling

- signal(), kill() allows a process to be terminated or have specific signals sent to it

Example: Process Creation in UNIX

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Example: Process Creation in UNIX

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
  // child…
  …
  exec();
  }
else {
  // parent
  wait();
  }
…
```

Example: Process Creation in UNIX

csh (pid = 22)

```
…
pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (pid = 24)

```
…
pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

Example: Process Creation in UNIX

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (pid = 24)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

Example: Process Creation in UNIX

csh (pid = 22)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

ls (pid = 24)

```
//ls program

main(){

   //look up dir

   …

}
```

What other process state does the OS manage?

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state 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	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Fields of a process table entry

What about the OS?

Is the OS a process?

It is a program in execution, after all …

Does it need a process control block?

Who manages its state when its not running?

Threads

Processes have the following components:

- **an address space**
- **a collection of operating system state**
- **a CPU context … or *thread of control***

On multiprocessor systems, with several CPUs, it would make sense for a process to have several CPU contexts (threads of control)

Multiple threads of control can run in the same address space on a single CPU system too!

***"thread of control"* and *"address space"* are orthogonal concepts**

Threads

- **Threads share a process address space with zero or more other threads**

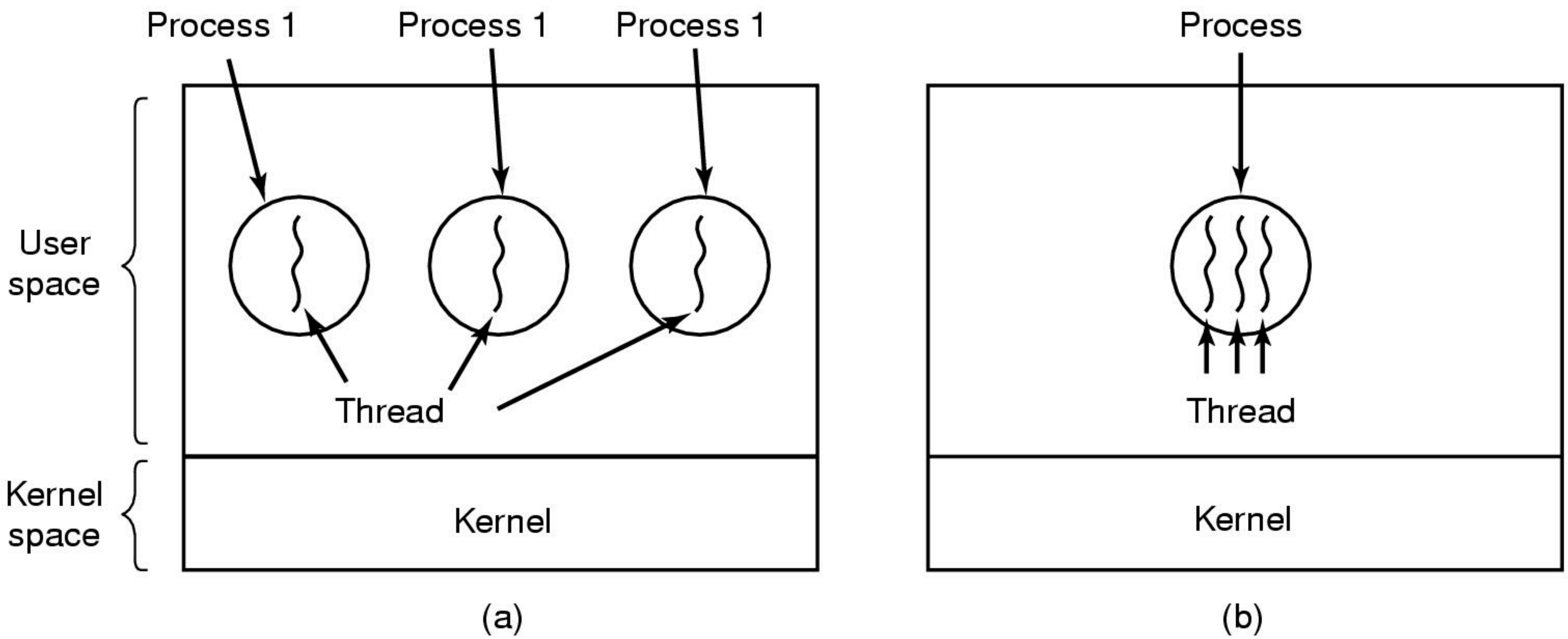
- **Threads have their own**
  CPU State (PC, SP, register values, etc.)
  Stack

- **What other OS state should be private to threads?**

A traditional process can be viewed as:

An address space with a single thread!

Threads vs Processes



(a) Three processes each with one thread

(b) One process with three threads

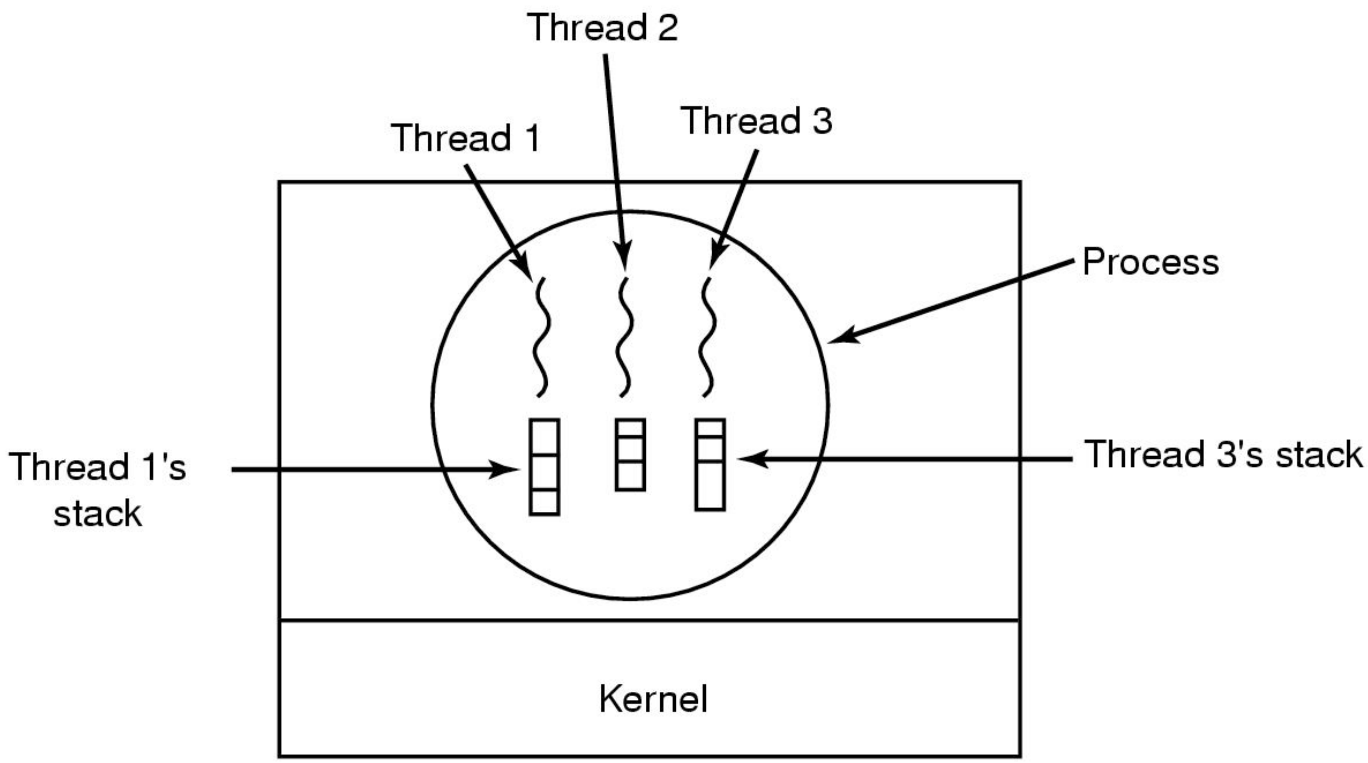
Process State vs Thread State

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	

Items shared by all threads in a process

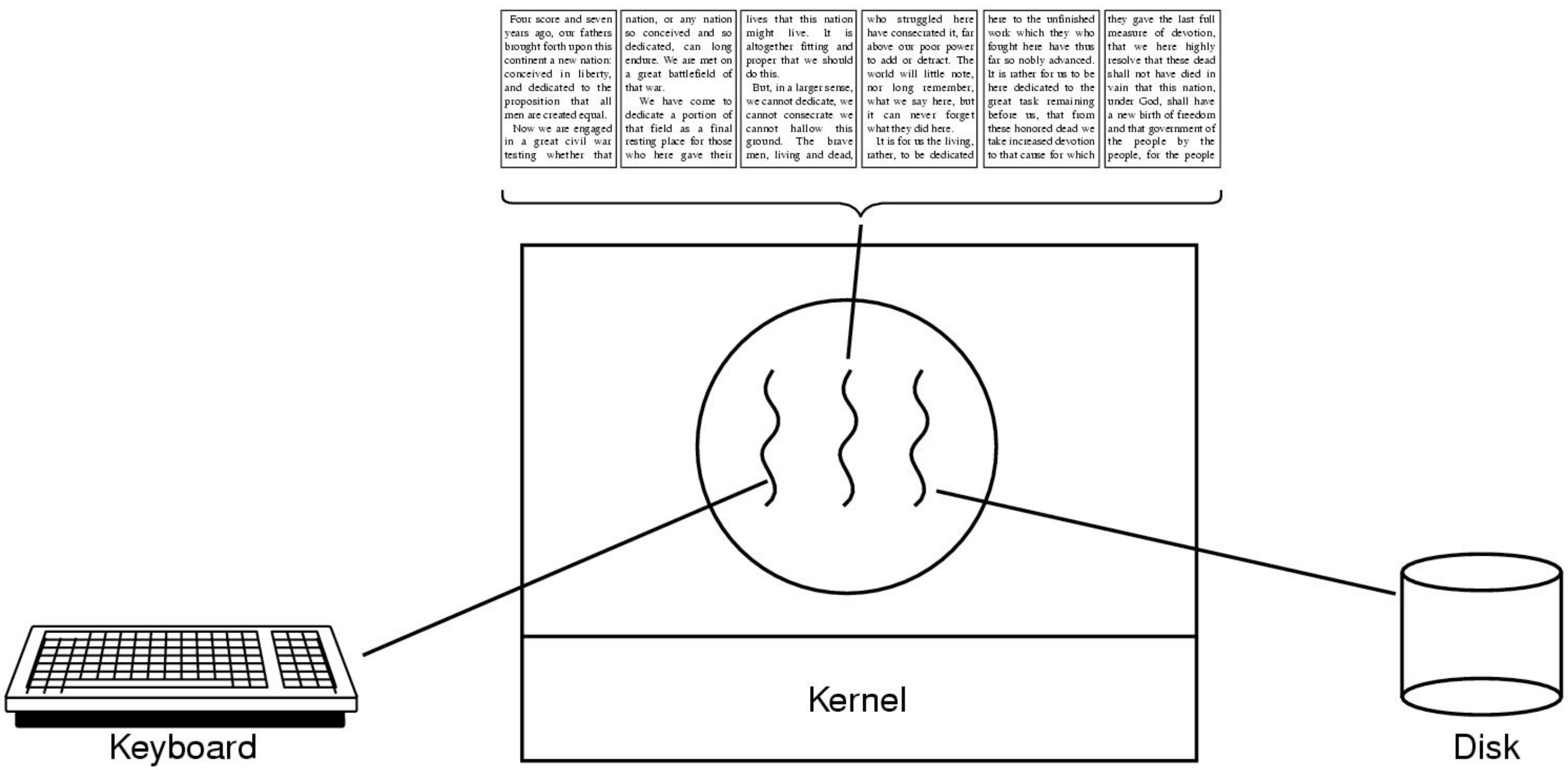
Items private to each thread

Independent execution of threads



Each thread has its own stack

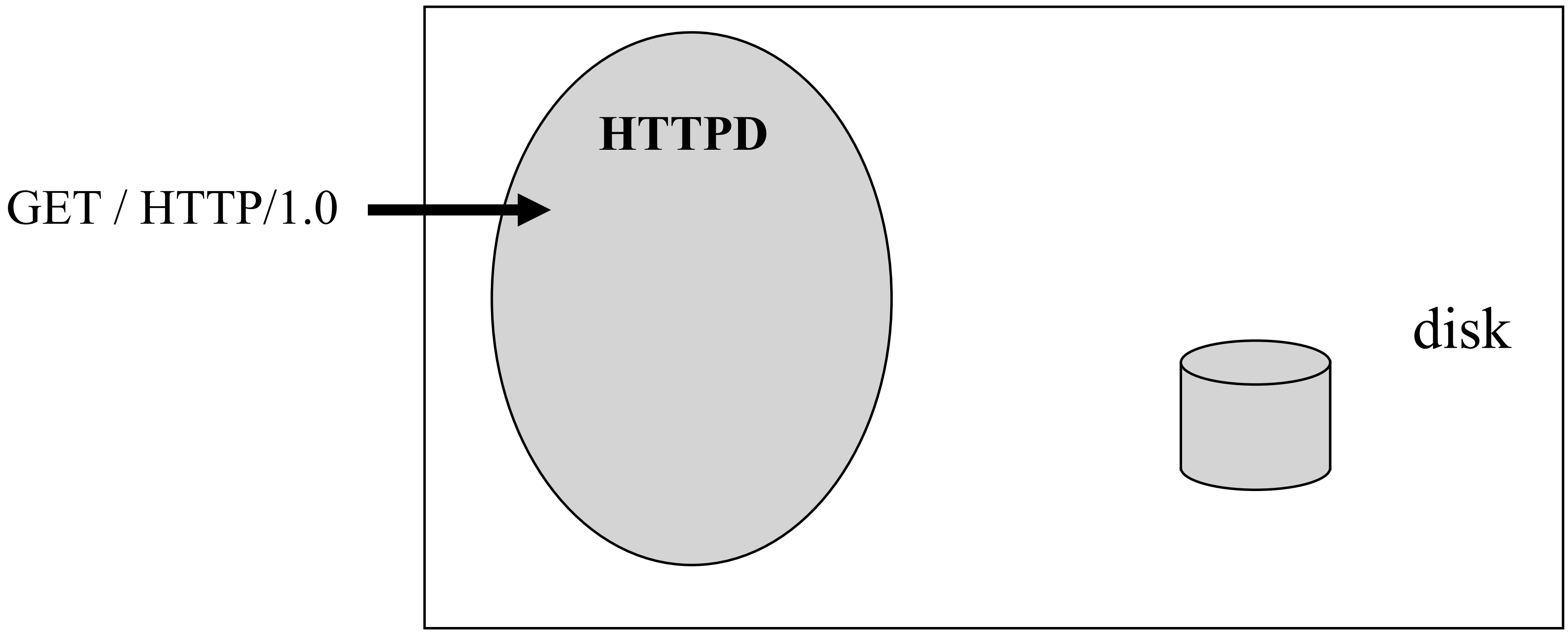
Thread Usage (1)



A word processor with three threads

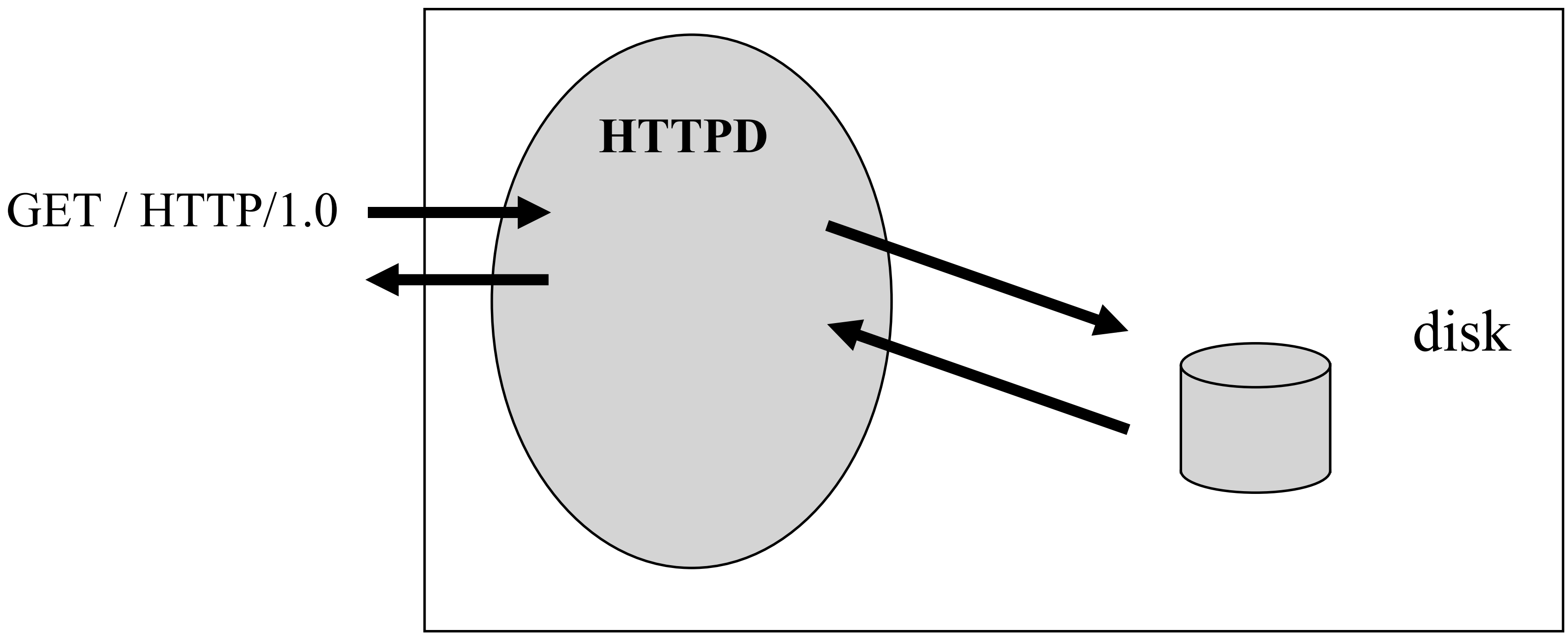
Processes versus threads - example

Web server receives a request for a page...



Processes versus threads - example

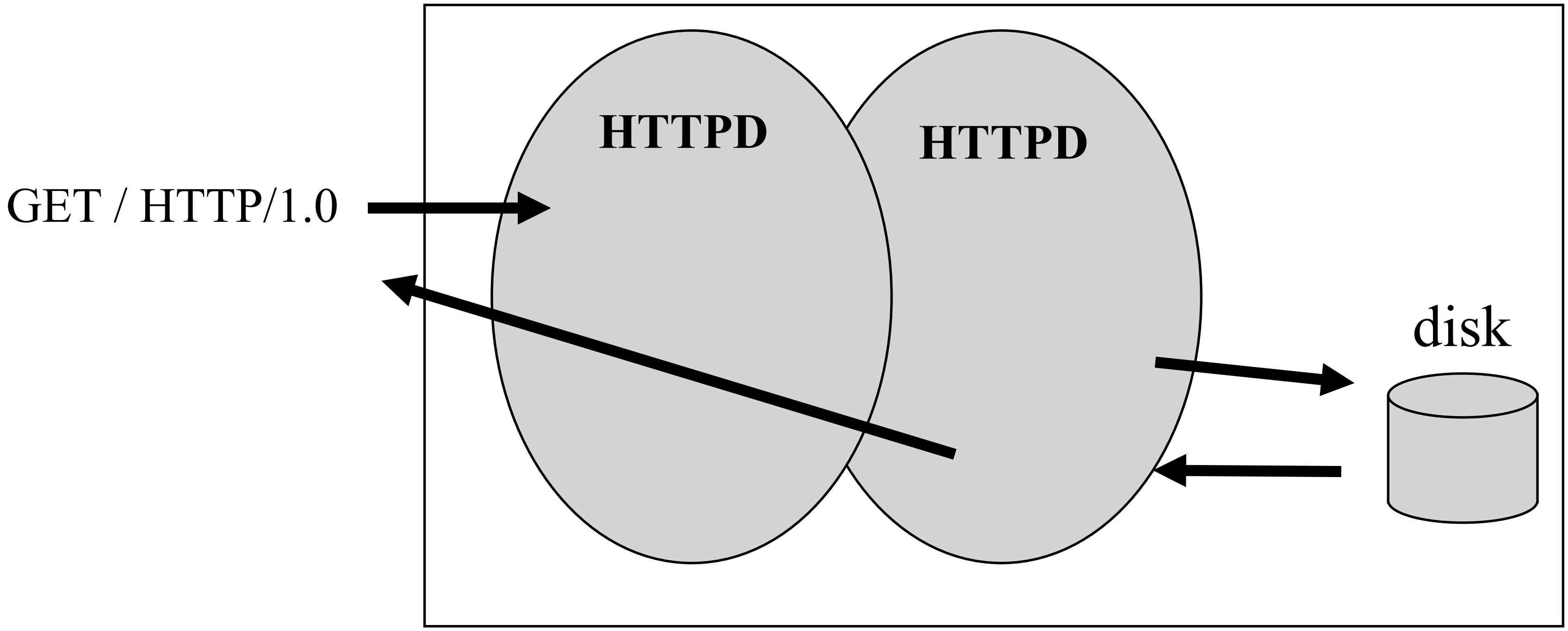
Web server receives a request for a page...



Why is this not a good web server design?

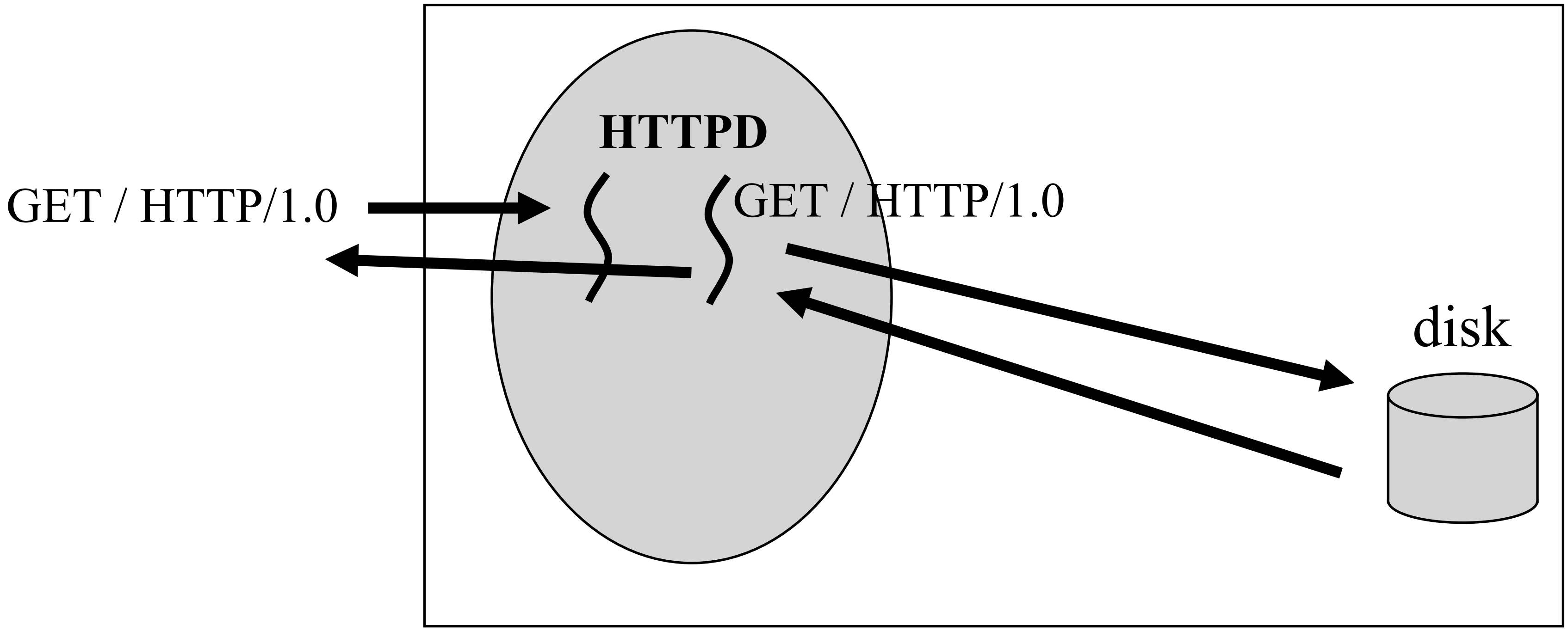
Processes versus threads - example

Web server receives a request for a page...



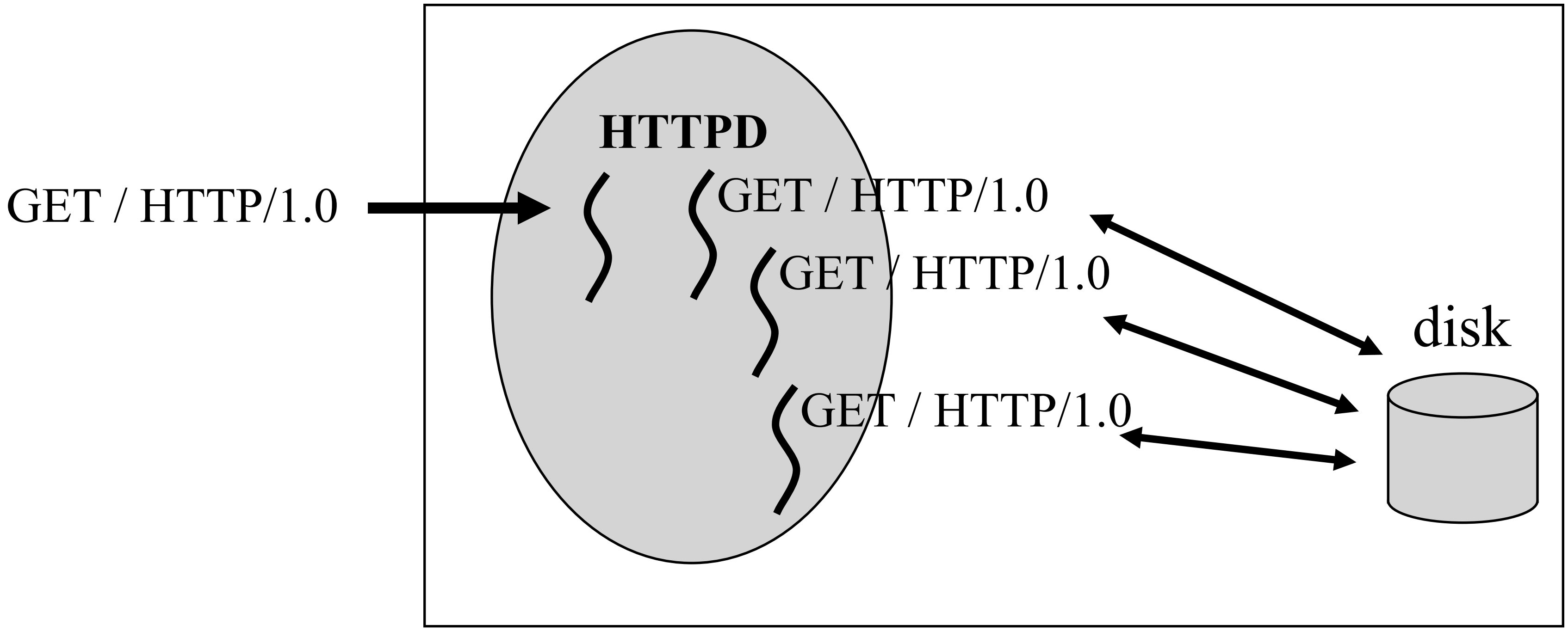
Processes versus threads - example

Web server receives a request for a page...

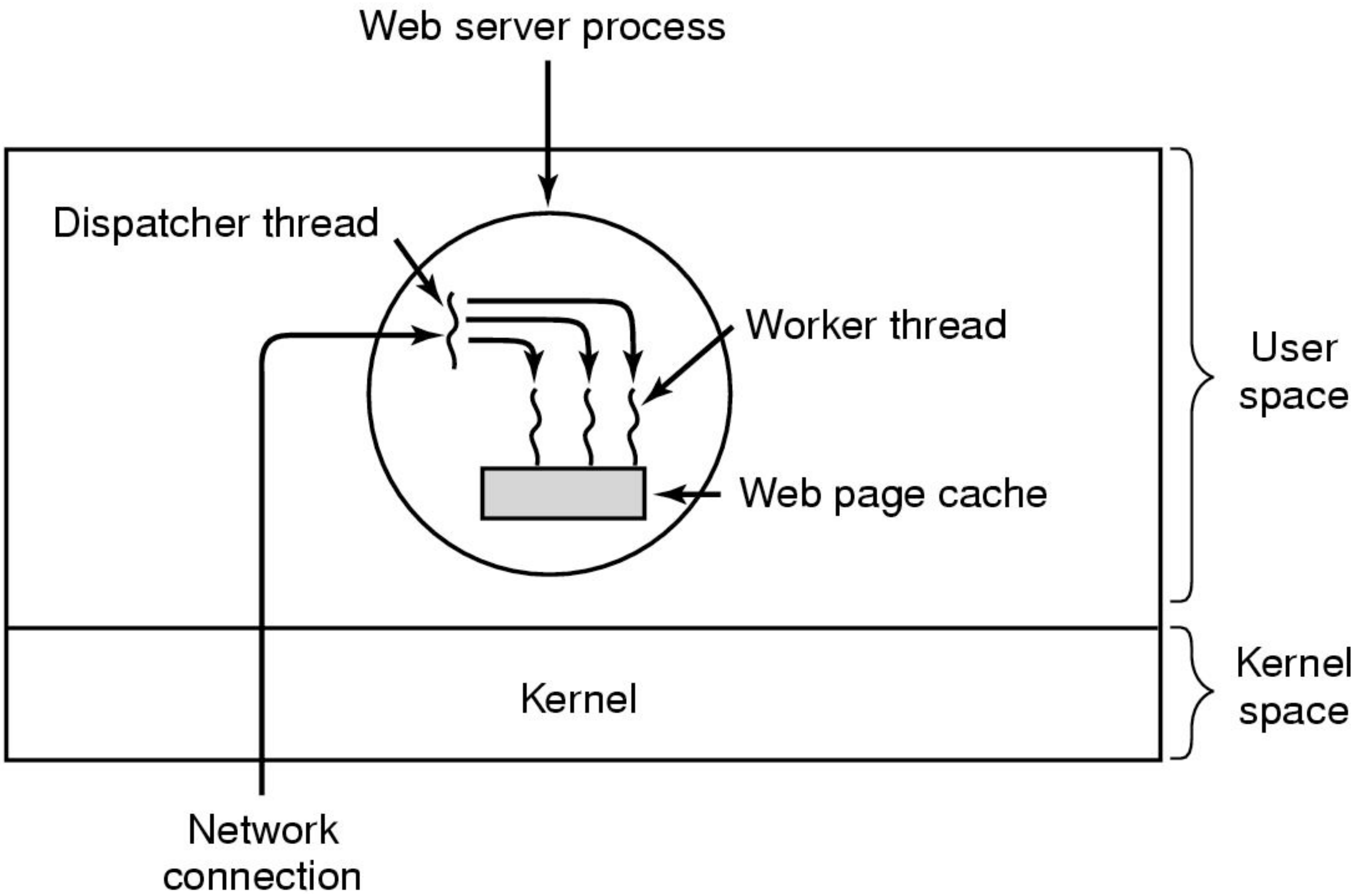


Processes versus threads - example

Web server receives a request for a page...



Threads in a web server



A multithreaded Web server

Threads in a web server

Outline of code for previous slide:

Dispatcher thread

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

Worker thread

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
       read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

System structuring options

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

Three ways to construct a server

Pros & Cons of Threads

Pros

- Overlap I/O with computation!
- Cheaper context switches
- Better mapping to shared memory multiprocessors

Cons

- Potential thread interactions
- Complexity of debugging
- Complexity of multi-threaded programming
- Backwards compatibility with existing code

Making Single-Threaded Code Multithreaded

There is a global variable.

The global variable is modified.

The global variable is then tested.

Making Single-Threaded Code Multithreaded

There is a global variable.
The global variable is modified.
The global variable is then tested.

Typical "C" code...

```
i = read (file, &buff, n);
if (errno) { ...print error message... }
```

Making Single-Threaded Code Multithreaded

There is a global variable.
The global variable is modified.
The global variable is then tested.

Typical "C" code...

```
i = read (file, &buff, n);
if (errno) { ...print error message... }
```

Now imagine that several threads are executing...

Making Single-Threaded Code Multithreaded

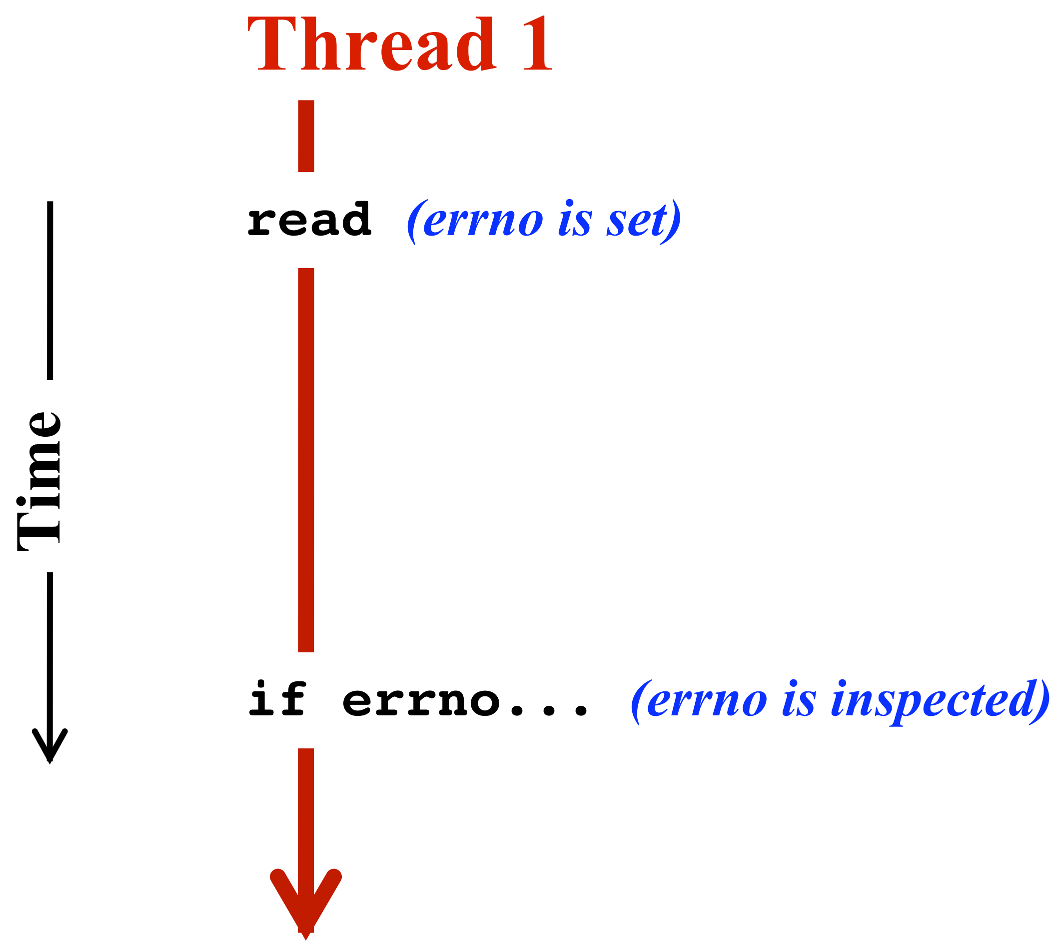
Thread 1

Time

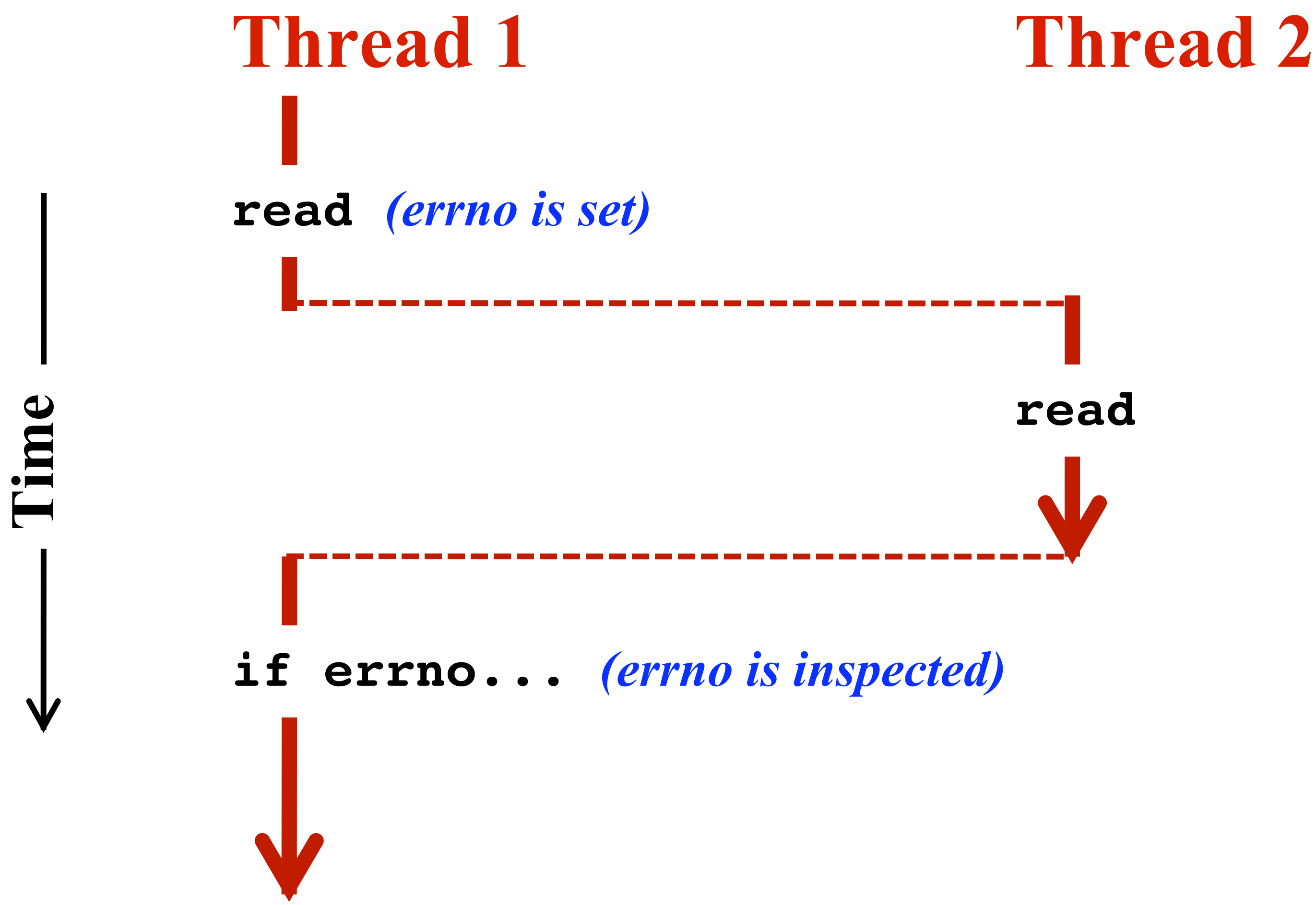
`read` *(errno is set)*

`if errno...` *(errno is inspected)*

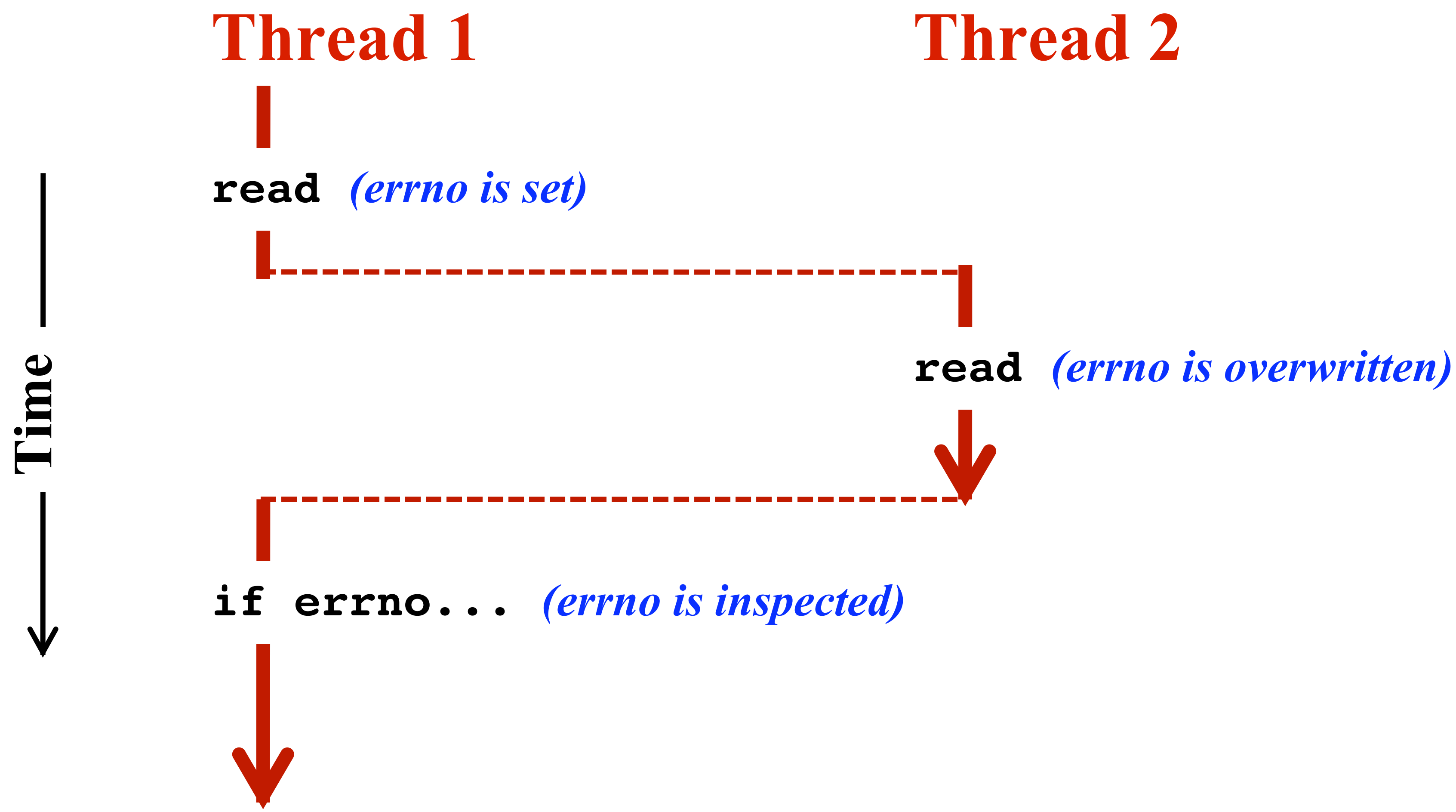
Making Single-Threaded Code Multithreaded



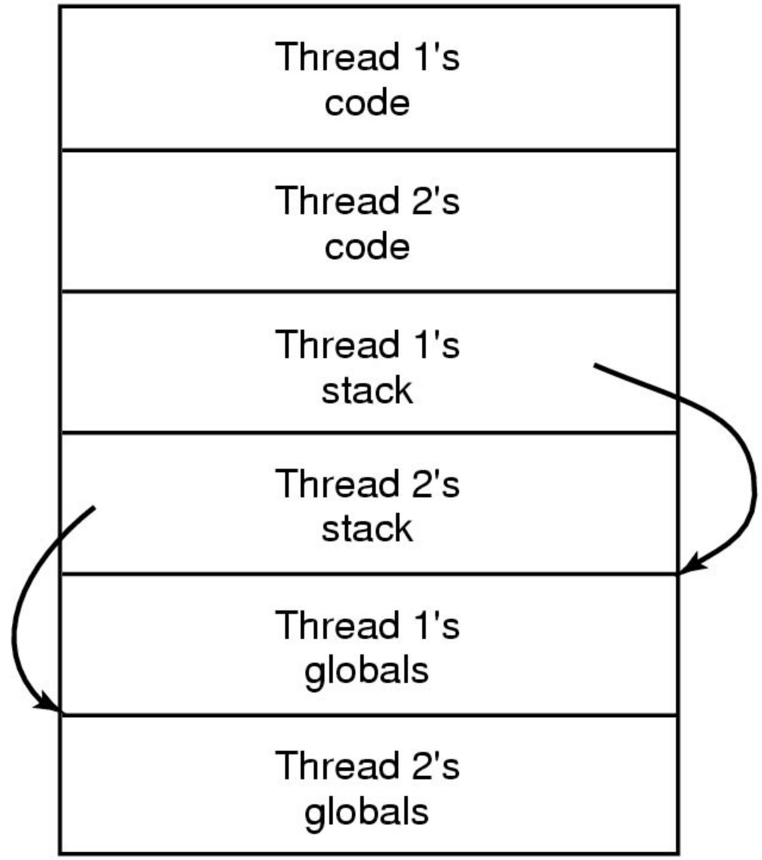
Making Single-Threaded Code Multithreaded



Making Single-Threaded Code Multithreaded



Making Single-Threaded Code Multithreaded



Threads can have private global variables

User-Level Threads

Threads can be implemented...

- By the OS, or
- At user level

Kernel-Level Thread Implementation

The Kernel contains the code to switch

switch between different threads.

User-Level Thread Implementations

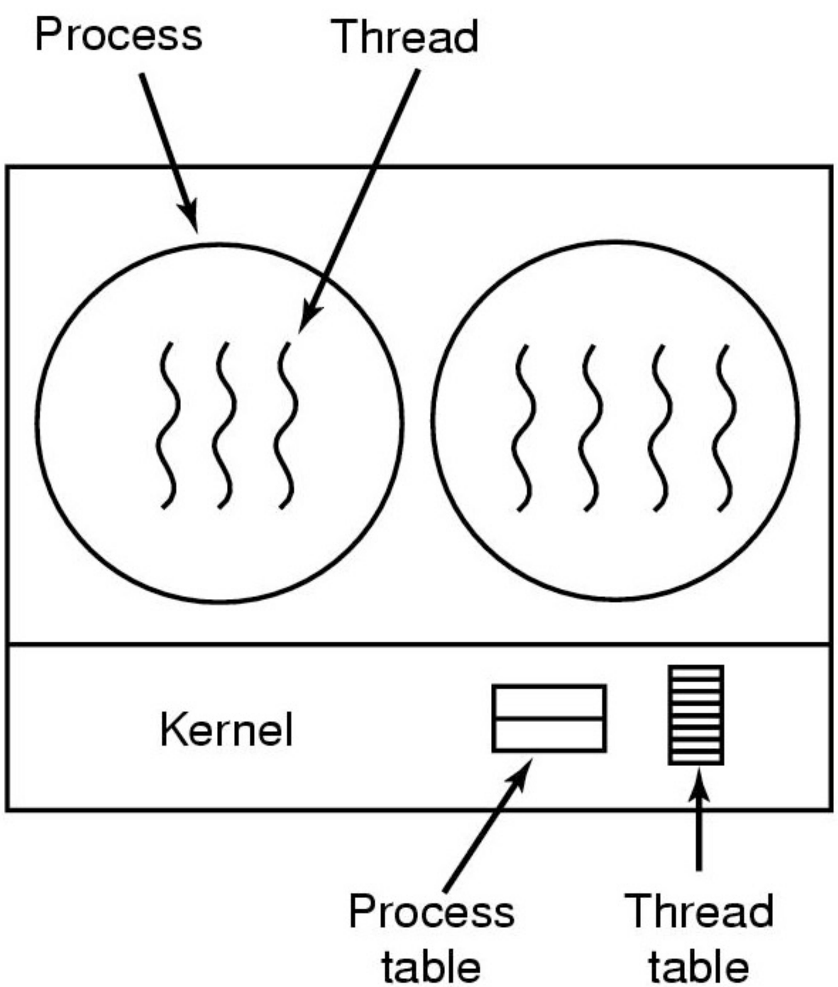
Thread scheduler runs as user code.

All thread management done by user code.

(Kernel sees only a traditional process.)

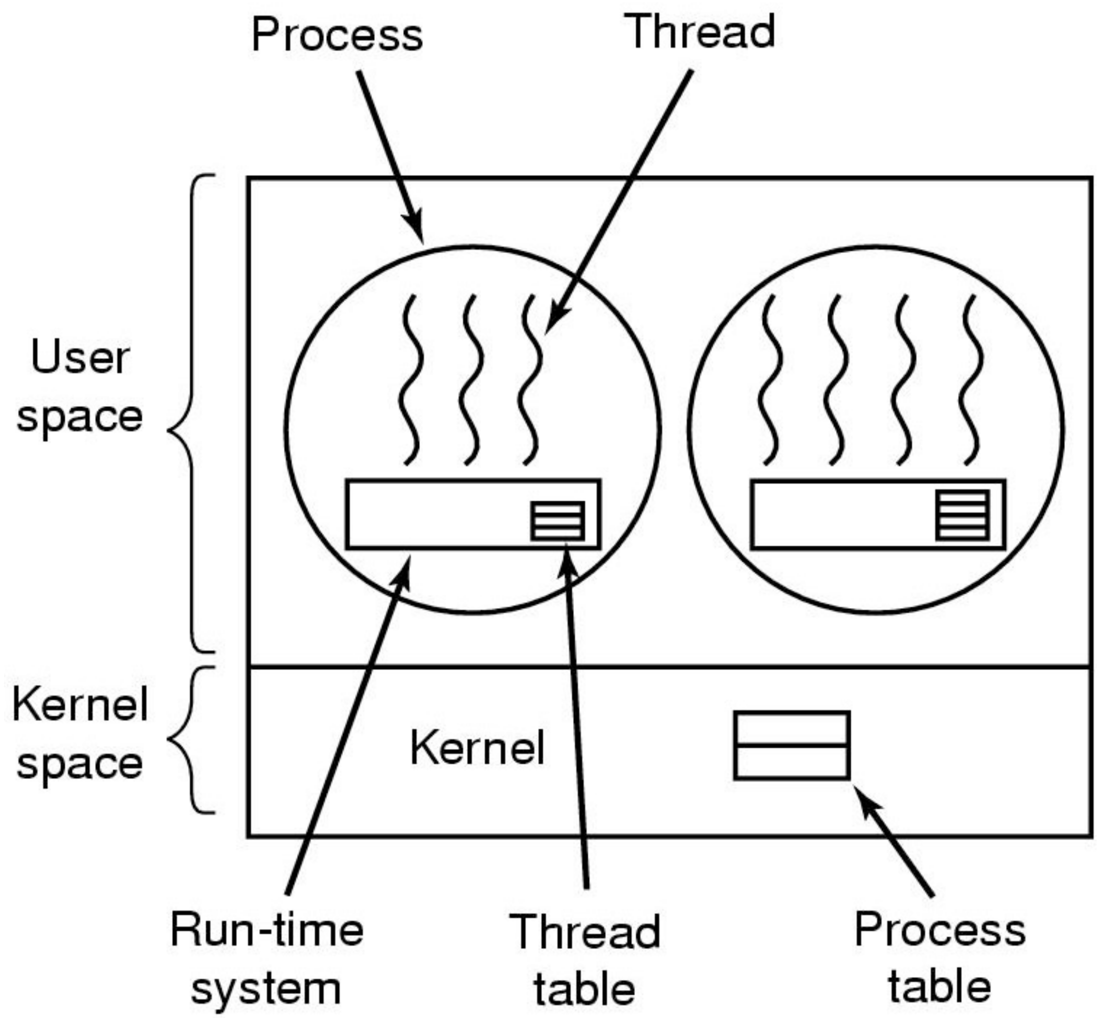
1: Implementing Threads in the Kernel

The thread switching code is in the kernel.



2: A “User-Level Threads Package”

The thread switching code is in the user address space.



User-level threads

Advantages

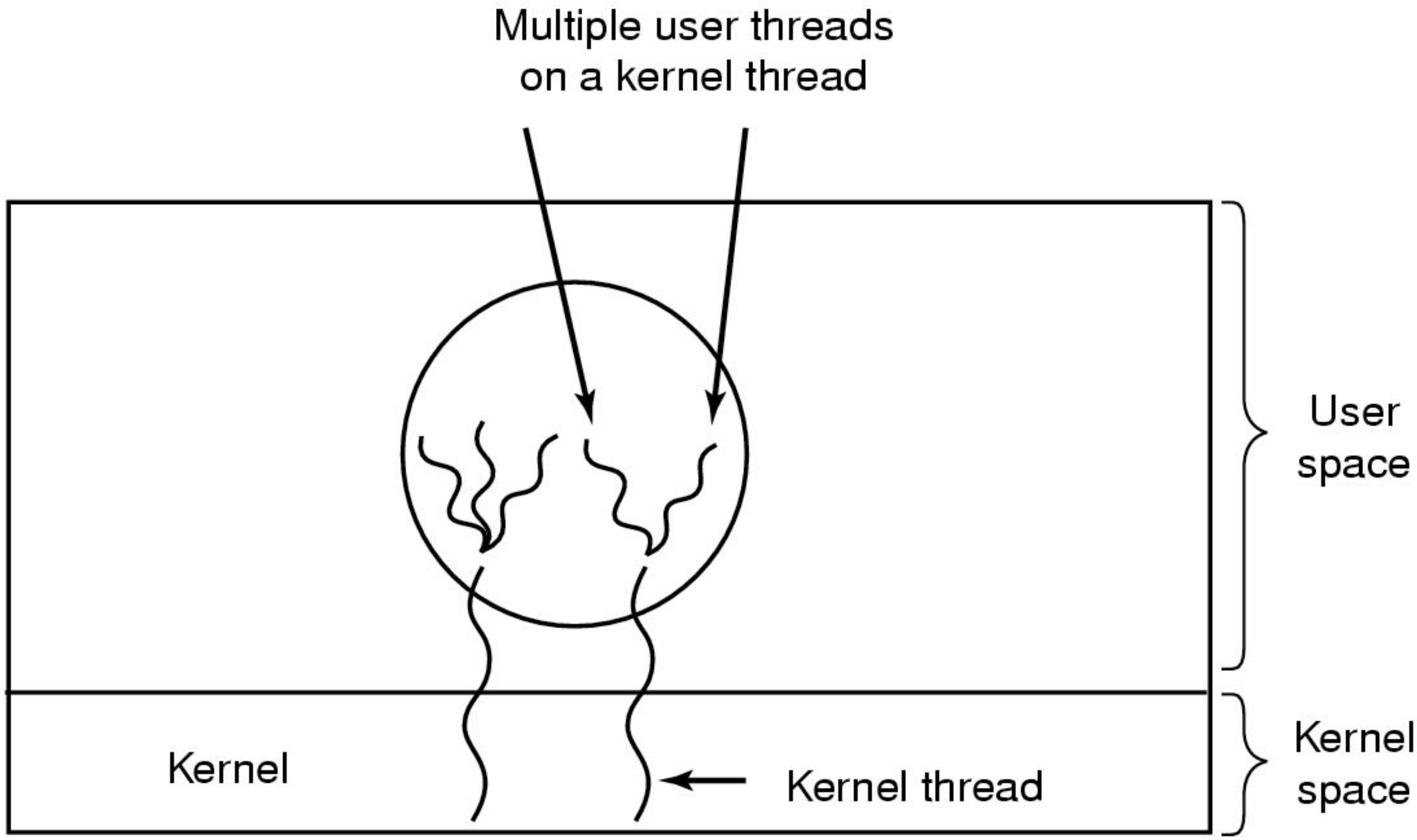
- Cheap context switch costs!
- User-programmable scheduling policy!

Disadvantages

- How to deal with blocking system calls!
- How to overlap I/O and computation!

Hybrid Thread Implementations

Multiplexing user-level threads onto kernel-level threads



Scheduler Activations

<u>Goals:</u> • Mimic functionality of kernel threads
• Gain performance of user space threads

The idea - kernel upcalls to user-level thread scheduling code when it handles a blocking system call or page fault

- User level thread scheduler can choose to run a different thread rather than blocking
- Kernel upcalls when system call or page fault returns

Kernel assigns virtual processors to each process
(which contains a user level thread scheduler)
Lets user level thread scheduler allocate threads to processors

<u>Problem:</u> Relies on upcalls
Kernel (lower layer) calls procedures in user space (higher layer)

Summary

- Processes
- Threads

Project 2:

Due in 1 week!

Okay to discuss my code,

... but write your own code!!!