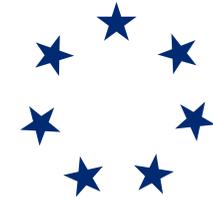


Thread



Instructor: Hengming Zou, Ph.D.

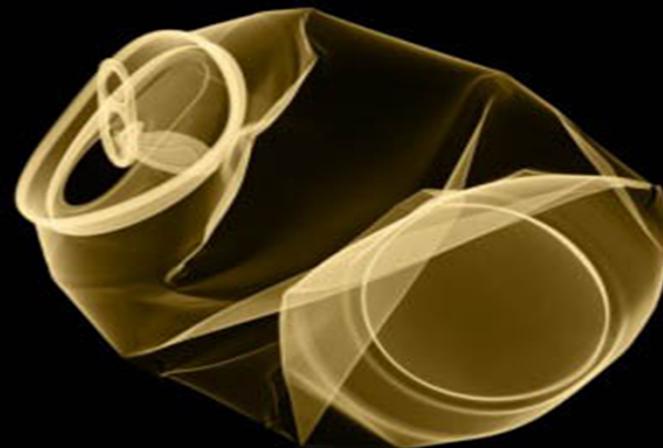
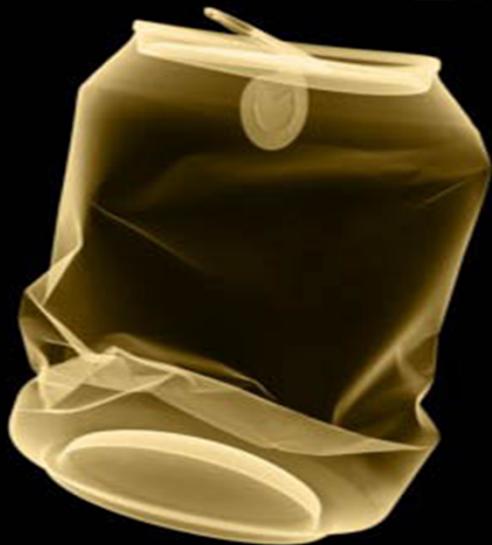


In Pursuit of Absolute Simplicity 求于至简，归于永恒

Thread



Lock Implementation



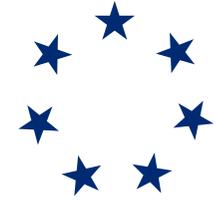
Synchronization

Content



- ⇒ Problem with process
- ⇒ Thread
 - ⇒ Thread model
- ⇒ Thread implementation
 - ⇒ User level threads
 - ⇒ Kernel level threads
 - ⇒ Hybrid
- ⇒ Thread cooperation
 - ⇒ Non-determinism

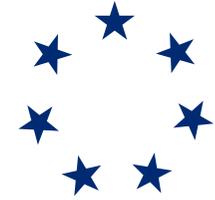




Problems with process

- ⇒ While supporting multiprogramming on shared hardware
- ⇒ Itself is single threaded!
 - i.e. a process can do only one thing at a time
 - blocking call renders entire process un-run-able
- ⇒ Thus, time calls for something else
 - threads





Threads

- ➔ Invented to support multiprocessing on process level
- ➔ Manage OS complexity
 - Multiple users, programs, I/O devices, etc.
- ➔ Each thread dedicated to do one task

- ➔ Sequence of executing instructions from a program
 - i.e. the running computation
- ➔ Play analogy: one actor on stage in a play



Threads



- ⇒ threads decompose mix of activities into several parallel tasks
- ⇒ Each job can work independently of others



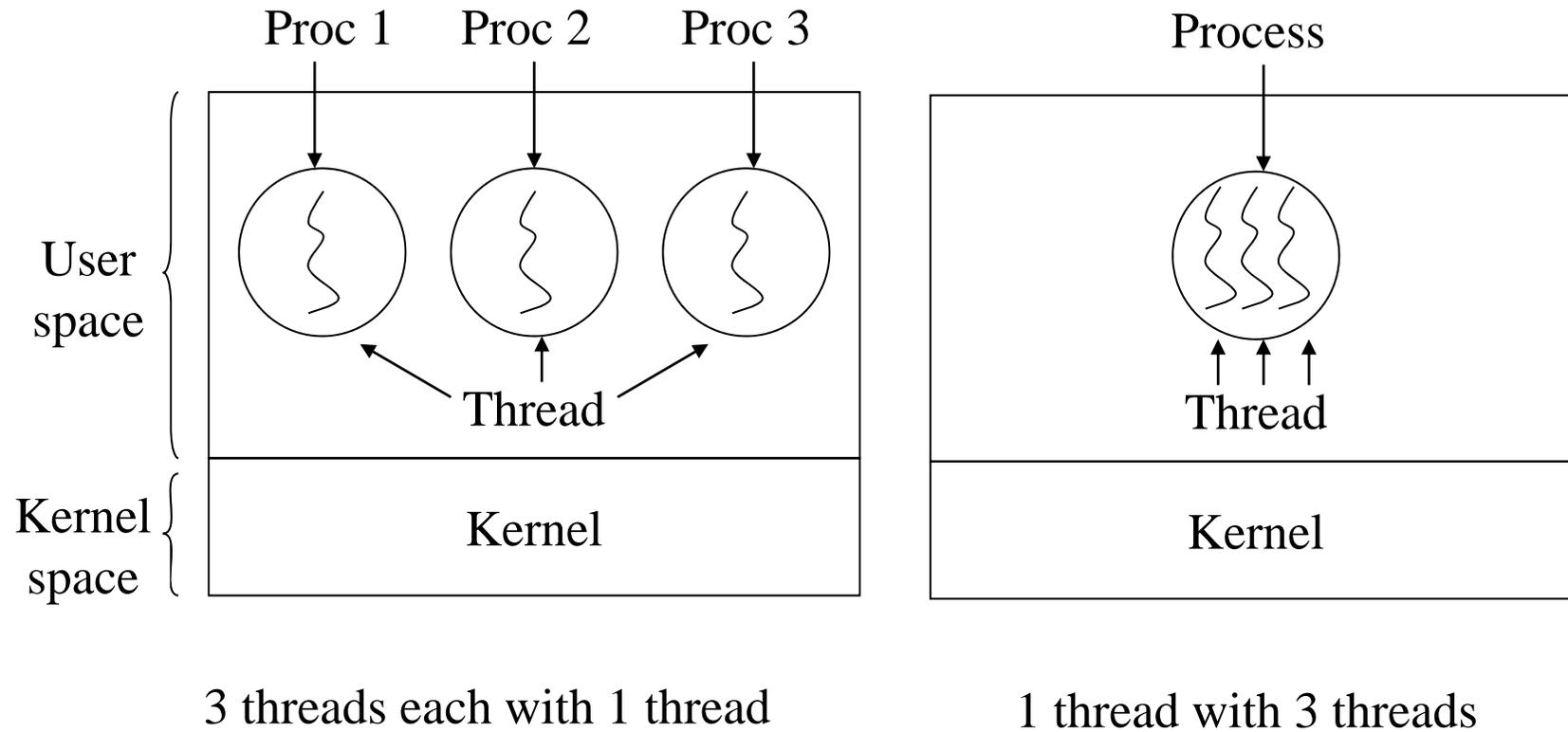
Thread 1

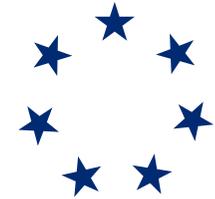
Thread 2

Thread 3



The Thread Model





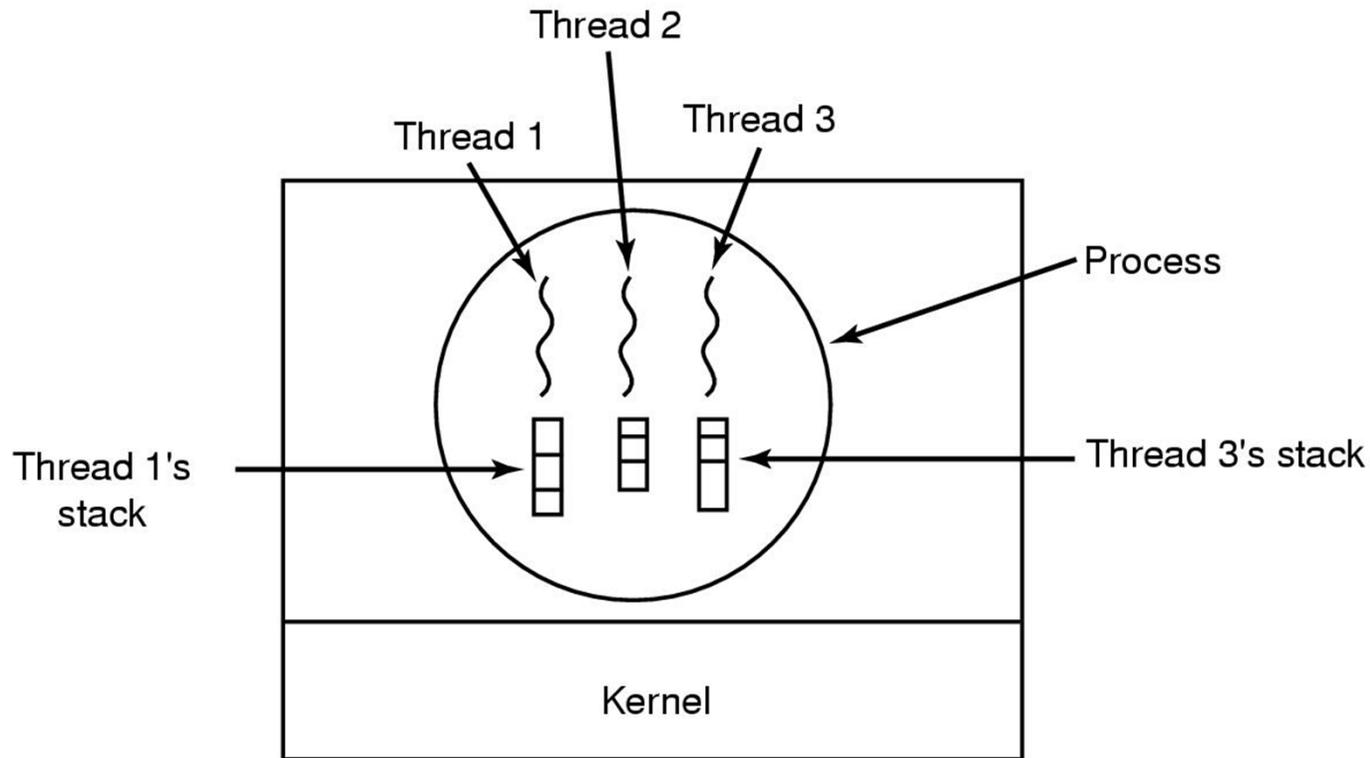
Shared and Private Items

- ➔ Some items shared by all threads in a thread
- ➔ Some items private to each thread

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



Shared and Private Items



Each thread has its own stack

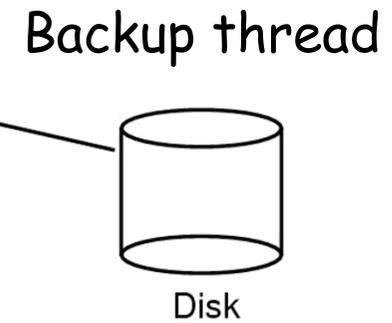
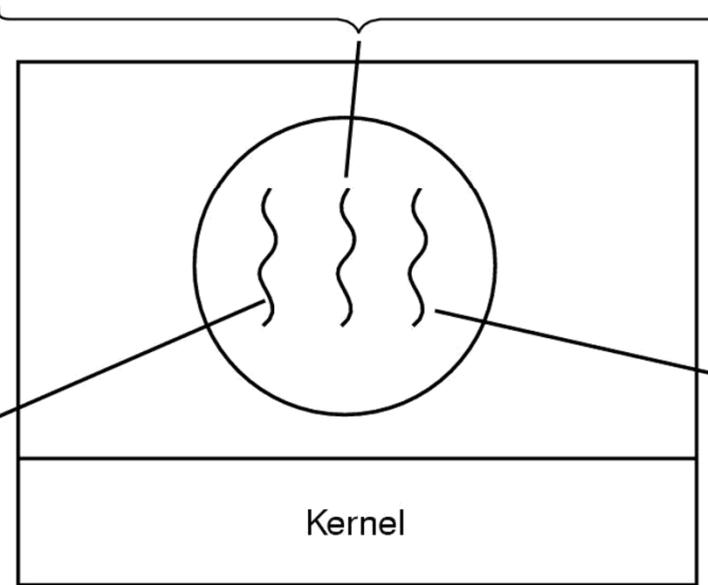
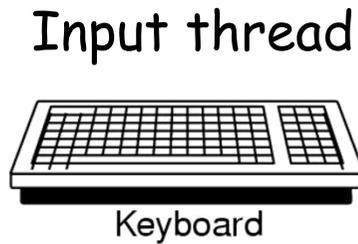


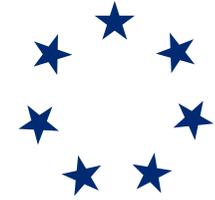
A Word Process w/3 Threads



Display thread

Four score and seven years ago, our fathers brought forth upon this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war testing whether that	nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting place for those who here gave their	lives that this nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead,	who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated	here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which	they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people
---	--	---	---	---	---

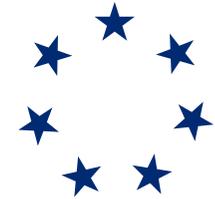




Implementation of Thread

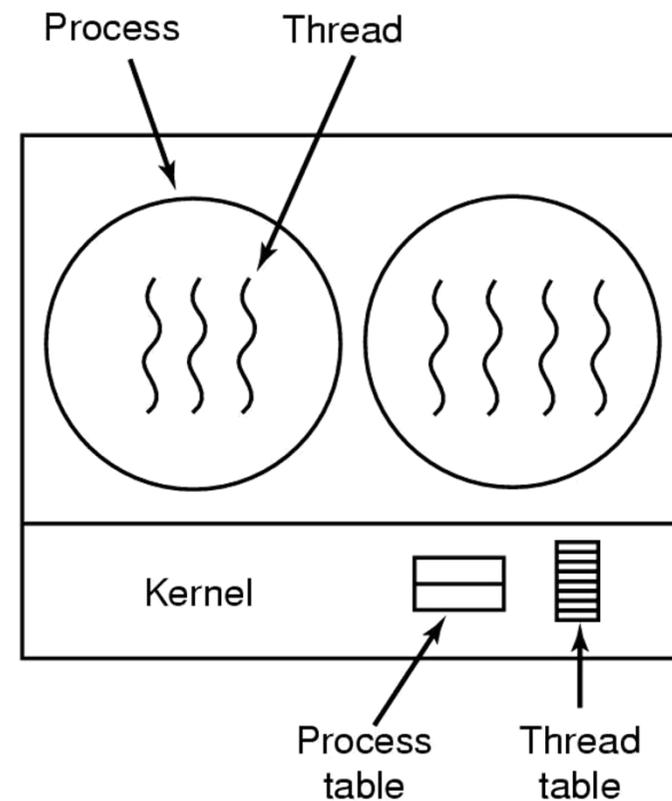
- ⇒ How many options to implement thread?
- ⇒ Implement in kernel space
- ⇒ Implement in user space
- ⇒ Hybrid implementation





Kernel-Level Implementation

- ➔ Completely implemented in kernel space
- ➔ OS acts as a scheduler
- ➔ OS maintains information about threads
 - In additions to process



Kernel-level implementation



Kernel-Level Implementation



⇒ Advantage:

- Easier to programming
- Blocking threads does not block process

⇒ Problems:

- Costly: need to trap into OS to switch threads
- OS space is limited (for maintaining info)
- Need to modifying OS!

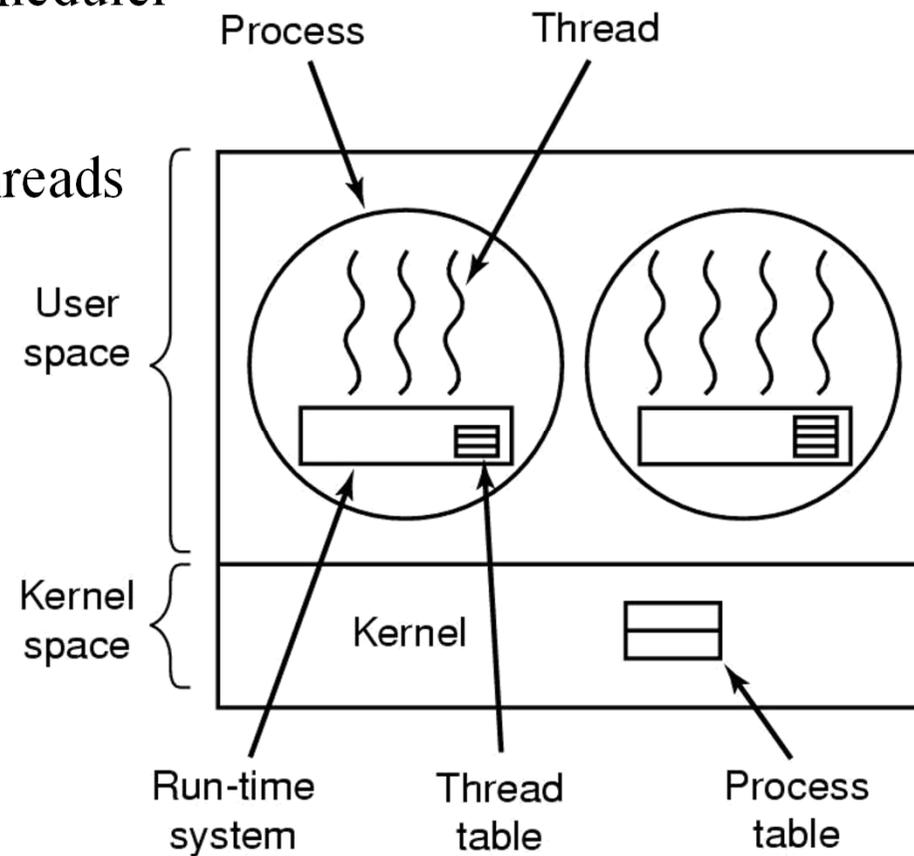




User-Level Implementation

- ➔ Implemented in user space
- ➔ A run-time system acts as a scheduler
- ➔ Threads voluntarily cooperate
 - i.e. yield control to other threads
- ➔ OS need not know about it

User-level implementation



User-Level Implementation



⇒ Advantage:

- Flexible, can be implemented on any OS
- Faster: no need to trap into OS

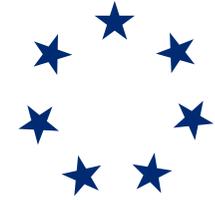
⇒ Problems:

- Programming is tricky
- blocking threads block process!

⇒ Question:

- How do we solve the problem of blocking thread blocks the process?



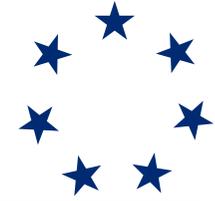


User-Level Implementation

- ➔ Modifying system calls to be unblocking
- ➔ Write a wrap around blocking calls
 - i.e. call only when it is safe to do so

- ➔ Example: Scheduler Activations
 - A technique solves the problem of blocking calls in user-level threads
- ➔ Method:
 - use up-call
- ➔ Goal – mimic functionality of kernel threads
 - gain performance of user space threads





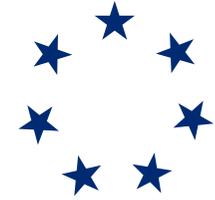
Scheduler Activations

- ⇒ Kernel assigns virtual processors to thread
- ⇒ Runtime sys. allocate threads to processors
- ⇒ Blocking threads are handled by OS up-call
 - i.e. OS notify runtime system about blocking calls

⇒ Problems?:

- Reliance on kernel (lower layer) calling
 - procedures in user space (higher layer)
- Violates layered structure of OS design
- OS correctness depends on run-time system





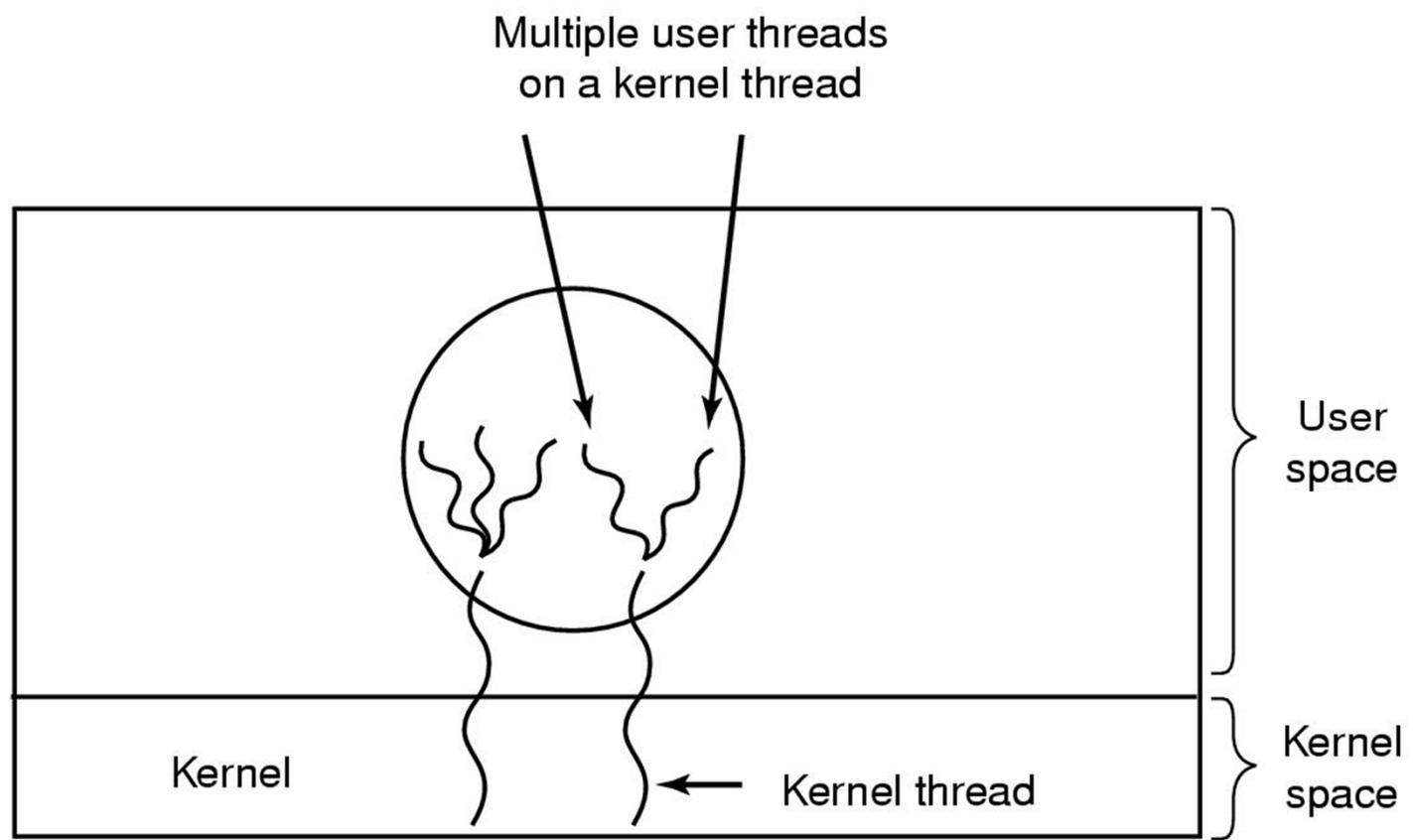
Hybrid Implementation

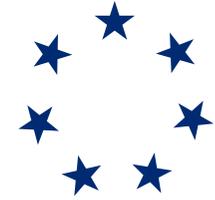
- ⇒ Can we have the best of both worlds
 - i.e. kernel-level and user-level implementations
- ⇒ While avoiding the problems of either?

- ⇒ Hybrid implementation
 - User-level threads are managed by runtime systems
 - Kernel-level threads are managed by OS
 - Multiplexing user-level threads onto kernel-level threads



Hybrid Implementation





Multiple Threads

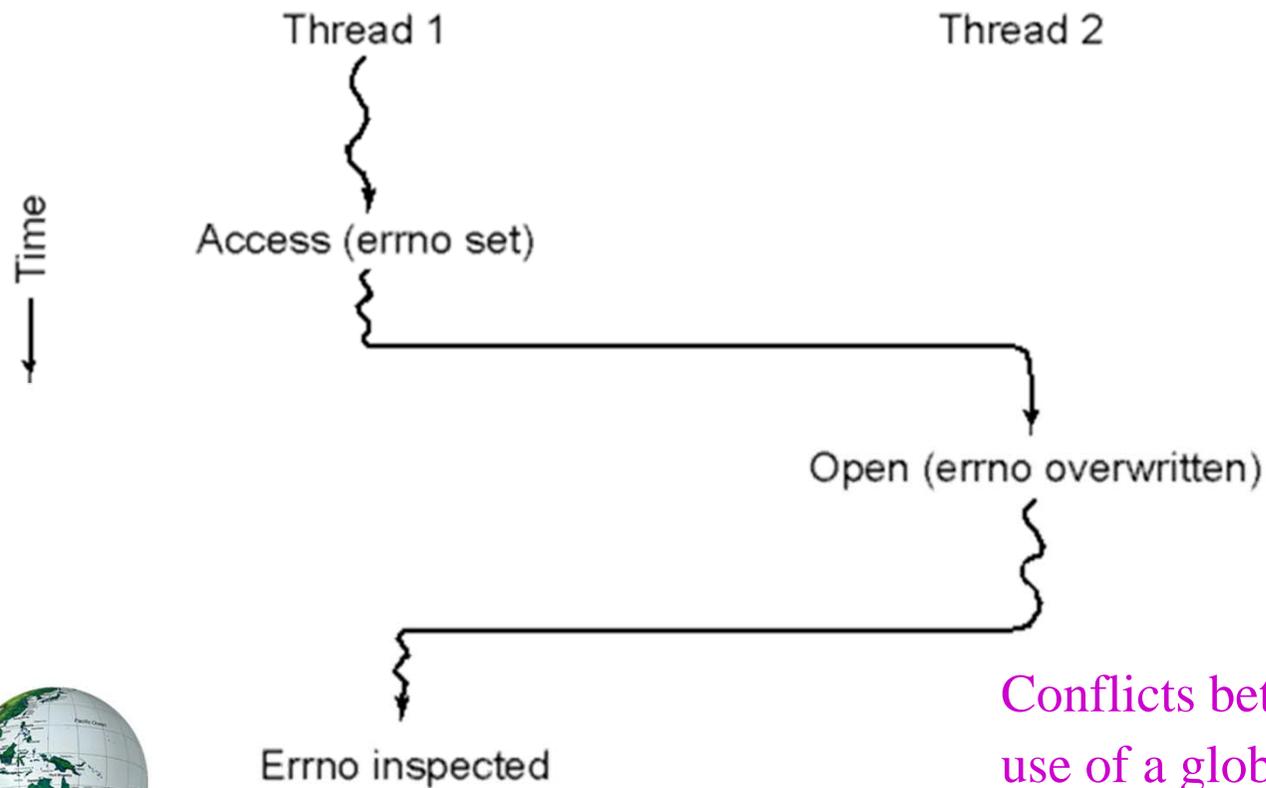
- ➔ Can have several threads in a single address space
 - That is what thread is invented for
- ➔ Play analogy: several actors on a single set
 - Sometimes interact (e.g. dance together)
 - Sometimes do independent tasks
- ➔ Private state for a thread vs. global state shared between threads
 - What private state must a thread have?
 - Other state is shared between all threads in a thread



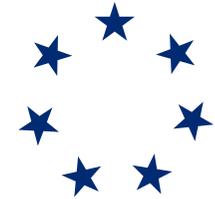


Multiple Threads

- ➔ Many programs are written in single-threaded threads
 - Make them multithreaded is very tricky

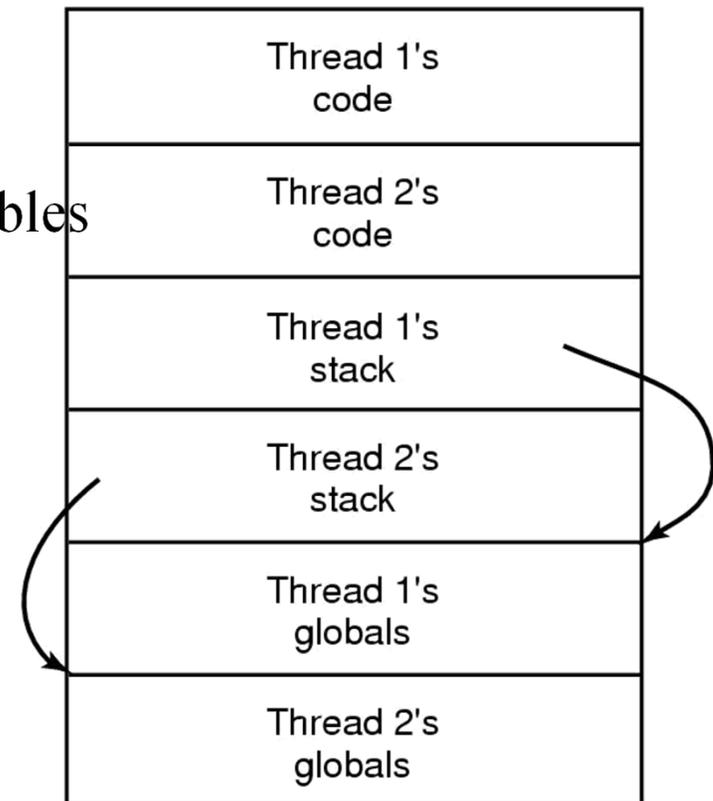


Conflicts between threads over use of a global variable



Multiple Threads

- ➔ So what can we do about it?
- ➔ Many solutions:
 - Prohibit global variables
 - Assign each thread private global variables



Threads can have private global variables



Cooperating Threads



- ➔ Often we create threads to cooperate:
 - Each thread handles one request
 - Each thread can issue a blocking disk I/O,
 - wait for I/O to finish
 - then continue with next part of its request

- ➔ Even though thread blocks, other threads can make progress
 - and new threads can start to handle new requests



Cooperating Threads



- ⇒ Issues with cooperating threads?
- ⇒ Where is the state of each request stored?
 - In thread space shared by all threads?
 - In private space of the thread?
- ⇒ How to communicate with each other?
- ⇒ How to synchronize with each other





Cooperating Threads

- ⇒ Ordering of events from different threads is non-deterministic
 - different threads may have differing amounts of work done in 10s

⇒ thread A ----->

⇒ thread B - - - - ->

⇒ thread C - - - - ->



Cooperating Threads



⇒ Consequence:

- Results of multi-threaded programming can be non-deterministic

⇒ Example

- thread A: $x=1$
- thread B: $x=2$

⇒ Possible results?

⇒ Is 3 a possible output?

- yes



Atomic Operations



⇒ Another example:

– thread A

– $i=0$

– while ($i < 10$) {

– $i++$

– }

– print “A finished”

thread B

$i=0$

while ($i > -10$) {

$i--$

}

print “B finished”

⇒ Who will win?





Thoughts Change Reality
意念改变现实