OPERATING SYSTEMS

MASTER IN COMPUTER SCIENCE & BUSINESS TECHNOLOGY

Threads(2) & Process Cooperation

Professor: Olivier Perard

Email: operard@faculty.ie.edu

Github: https://github.com/operard/opsys_parallel/blob/master/mcsbt/README.md

Threads Implementation

Contents

- Multithreading Levels
- Multithreading Models
- Threading Issues

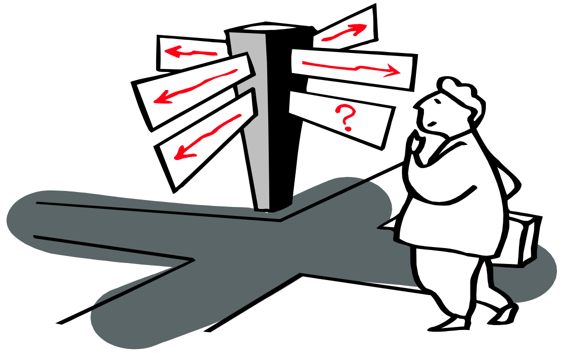
Multithreading vs. Single threading

- Single threading: when the OS does not recognize the concept of thread.
- Multithreading: when the OS supports multiple threads of execution within a single process.
- MS-DOS supports a single user process and a single thread.
- Older UNIXs supports multiple user processes but only support one thread per process.
- Solaris and Windows NT support multiple threads.

Multithreading Levels

- Thread library provides programmer with API for creating and managing threads.
- Three multithreading levels:

1) User-Level Threads (ULT)
   - Library entirely in user space.
2) Kernel-Level Threads (KLT)
   - Kernel-level library supported by the OS.
3) Hybrid ULT/KLT Approach

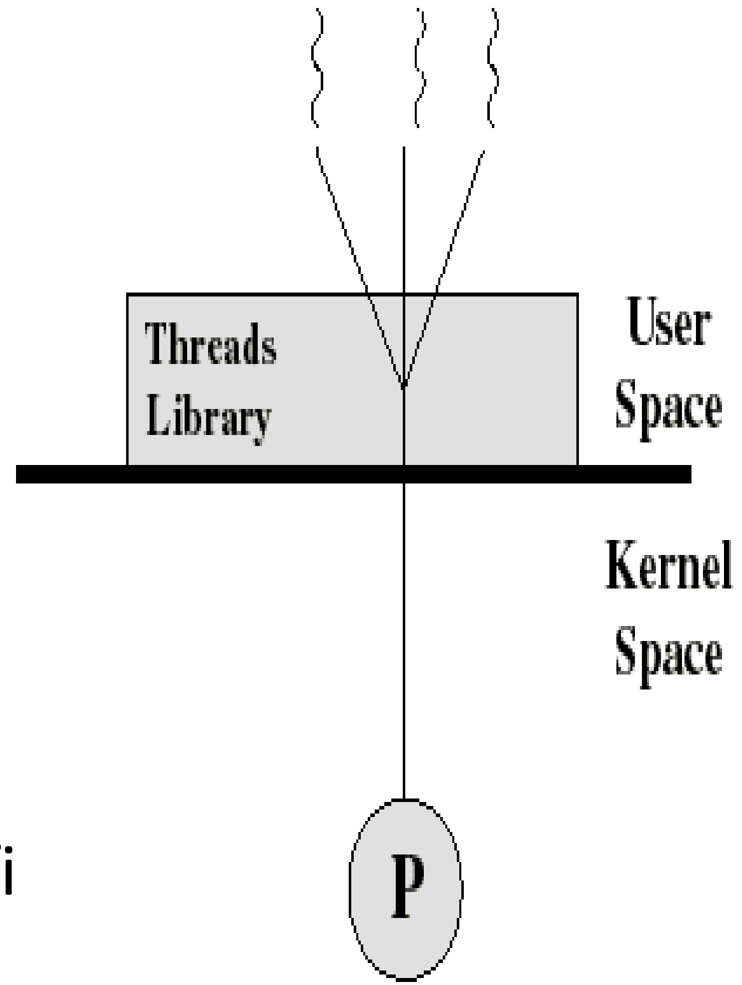


User(-level) and Kernel(-level) Threads

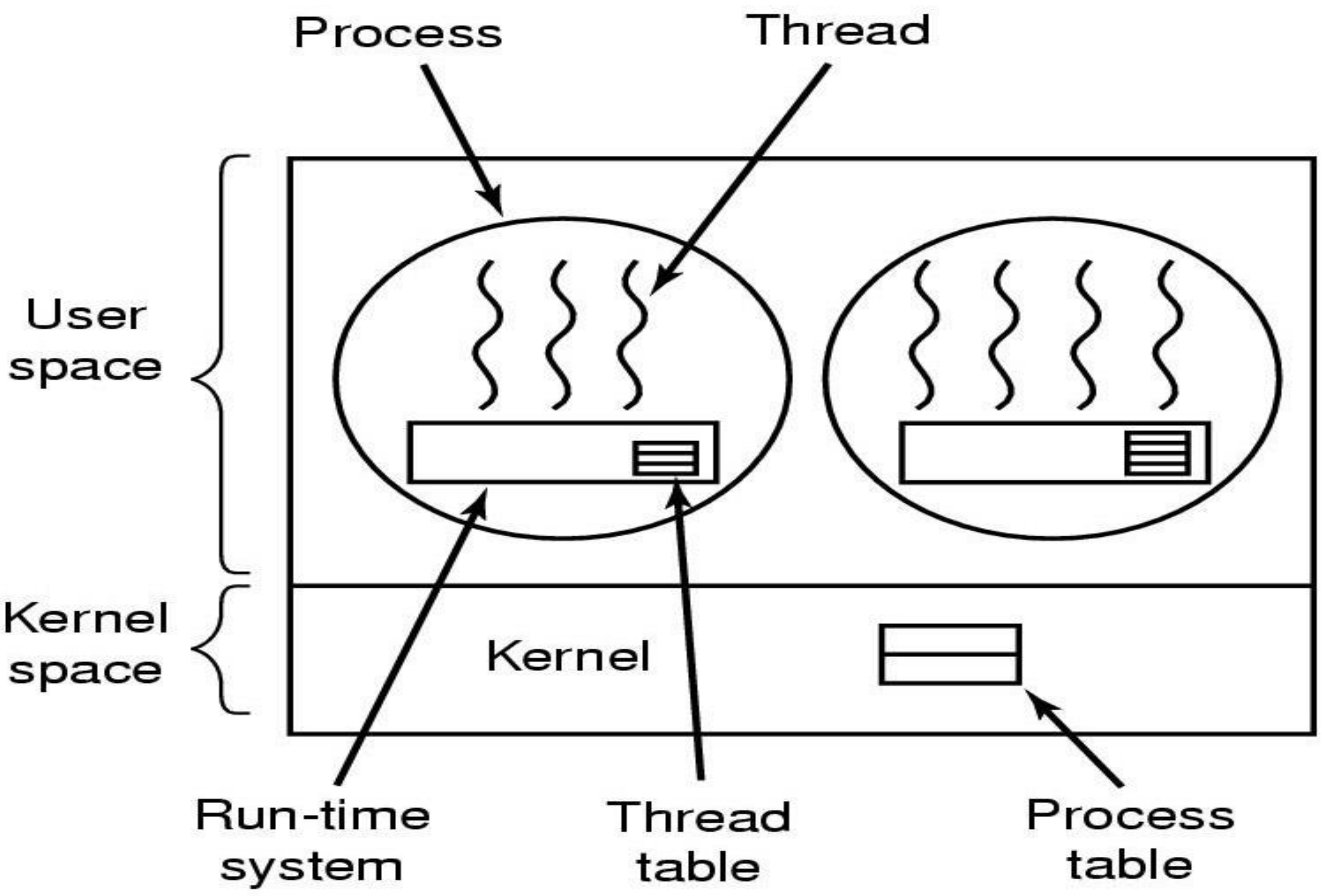
- User(-level) threads – management by user-level threads library.
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel(-level) threads – supported by the Kernel.
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

1) User-Level Threads (ULT)

- Thread management done by user-level threads library
- The kernel is not aware of the existence of threads.
- All thread management is done by the application by using a thread library.
- Thread switching does not require kernel mode privileges.
- Scheduling is application specifi



Implementing Threads in User Space



ULT Idea

- Thread management done by user-level threads library.
- Threads library contains code for:
  - creating and destroying threads.
  - passing messages and data between threads.
  - scheduling thread execution.
  - saving and restoring thread contexts.
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

POSIX Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- May be provided either as ULT or KLT.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems (Solaris, Linux, Mac OS X).

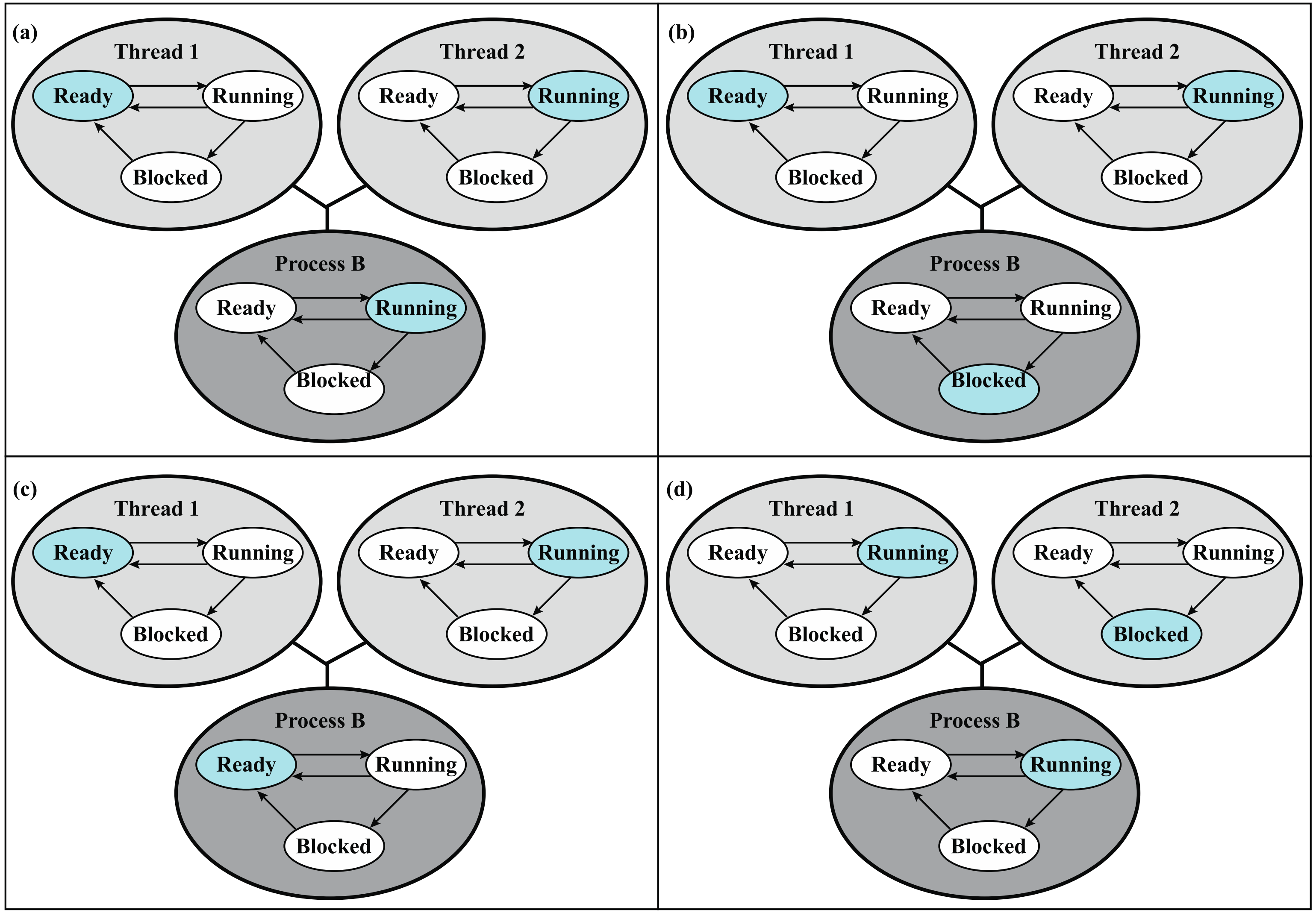
Some of the Pthreads function calls

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Kernel activity for ULTs

- The kernel is not aware of thread activity but it is still managing process activity.
- When a thread makes a system call, the whole task will be blocked.
- But for the thread library that thread is still in the running state.
- So thread states are independent of process states.

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY



Colored state
is current state

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

Advantages of ULTs

Thread switching does not require kernel mode privileges

Scheduling can be application specific

ULTs can run on any OS

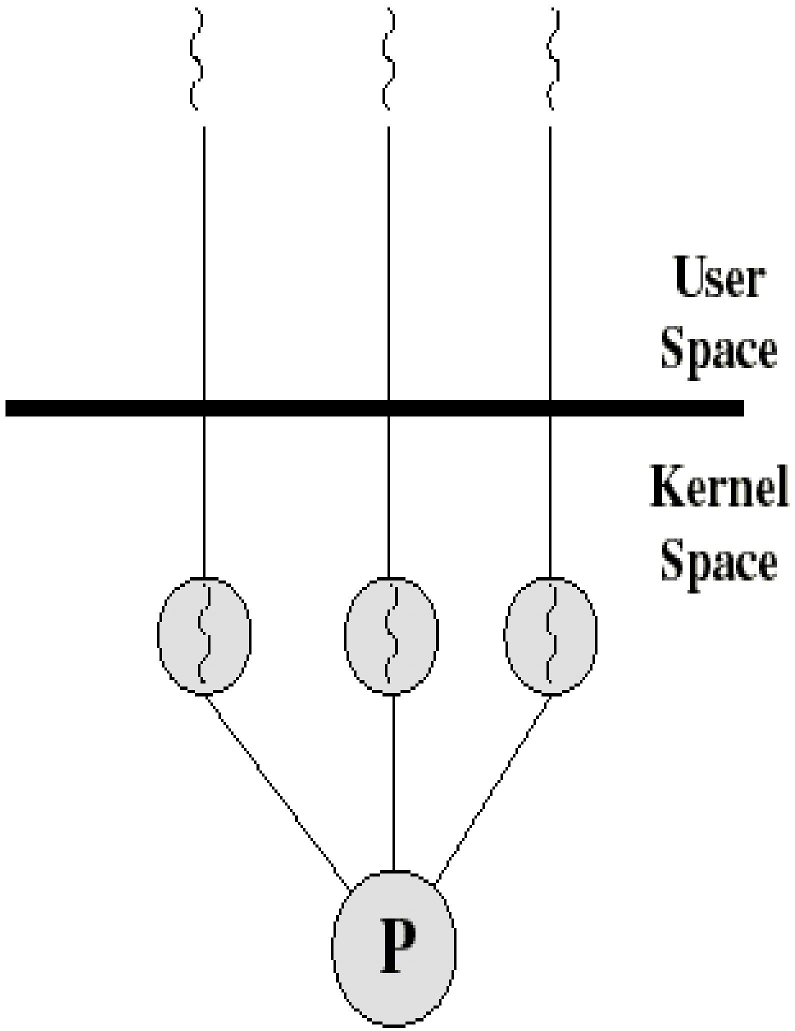
Advantages and inconveniences of ULT

- Advantages
  - Thread switching does not involve the kernel: no mode switching.
  - Scheduling can be application specific: choose the best algorithm.
  - ULTs can run on any OS. Only needs a thread library.

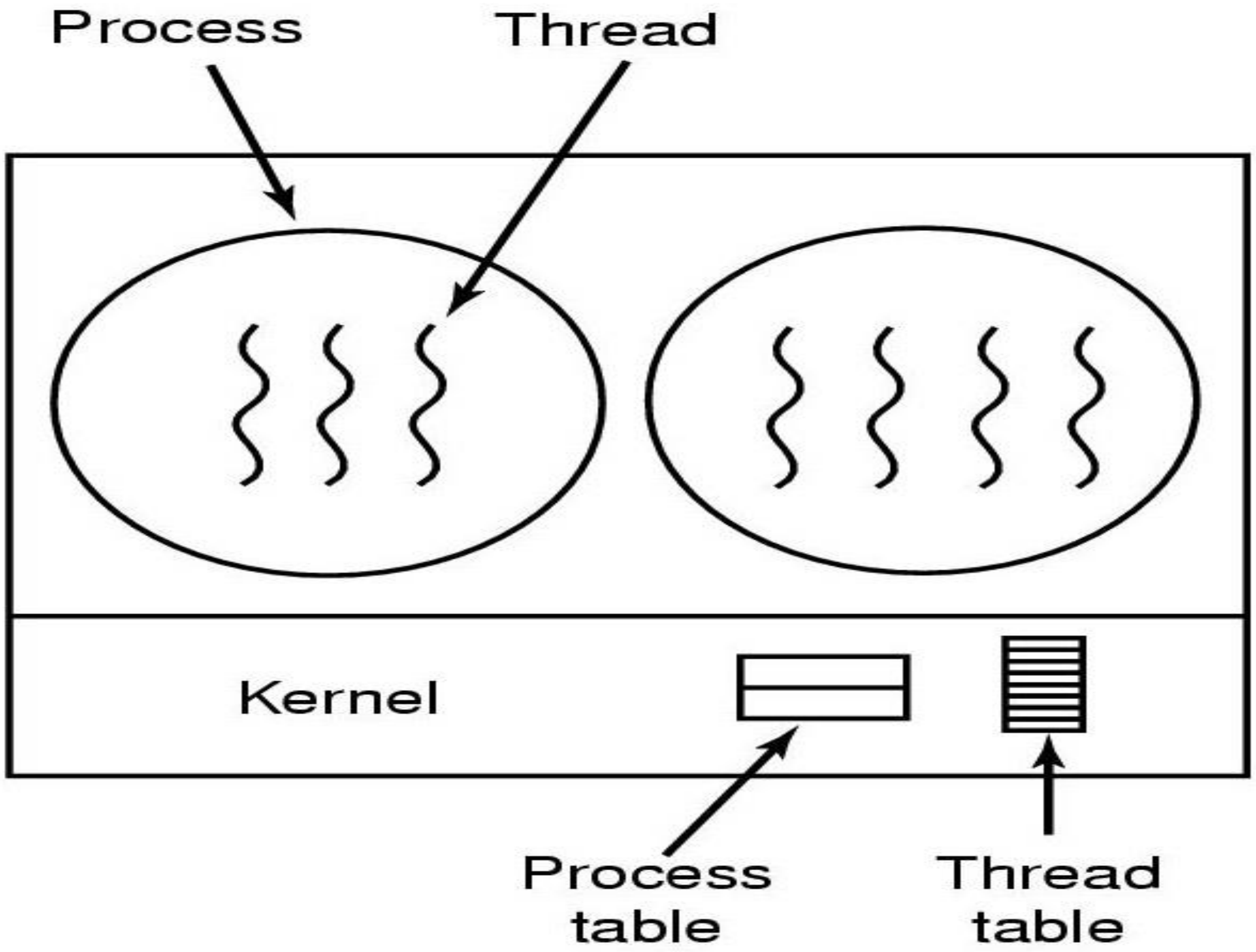
- Inconveniences
  - Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked.
  - The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors.

2) Kernel-Level Threads (KLT)

- All thread management is done by kernel.
- No thread library but an API to the kernel thread facility.
- Kernel maintains context information for the process and the threads.
- Switching between threads requires the kernel.
- Scheduling on a thread basis.



Implementing Threads in the Kernel



KLT Idea

- Threads supported by the Kernel.
- Examples:
  - Windows 2000/XP
  - OS/2
  - Linux
  - Solaris
  - Tru64 UNIX
  - Mac OS X

Linux Threads

- Linux refers to them as tasks rather than threads.
- Thread creation is done through **clone()** system call.
- **clone()** allows a child task to share the address space of the parent task (process).
- This sharing of the address space allows the cloned child task to behave much like a separate thread.

Advantages and inconveniences of KLT

- Advantages
  - the kernel can simultaneously schedule many threads of the same process on many processors.
  - blocking is done on a thread level.
  - kernel routines can be multithreaded.
- Inconveniences
  - thread switching within the same process involves the kernel. We have 2 mode switches per thread switch.
  - this results in a significant slow down.

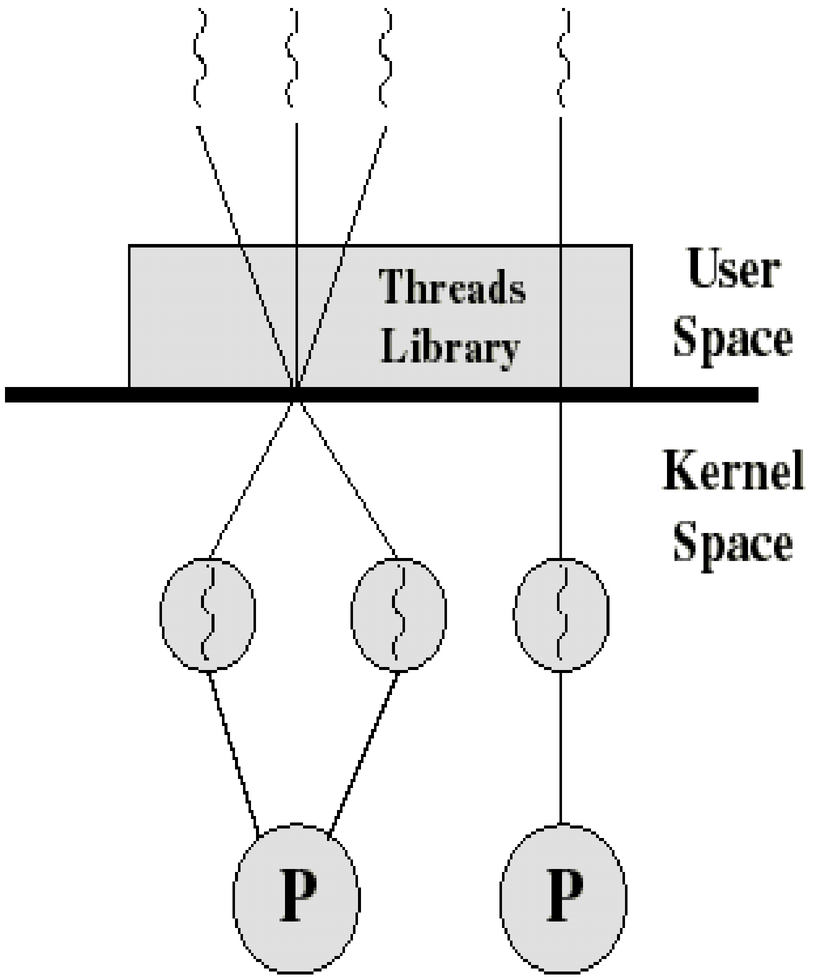
Thread operation latencies (μs)

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

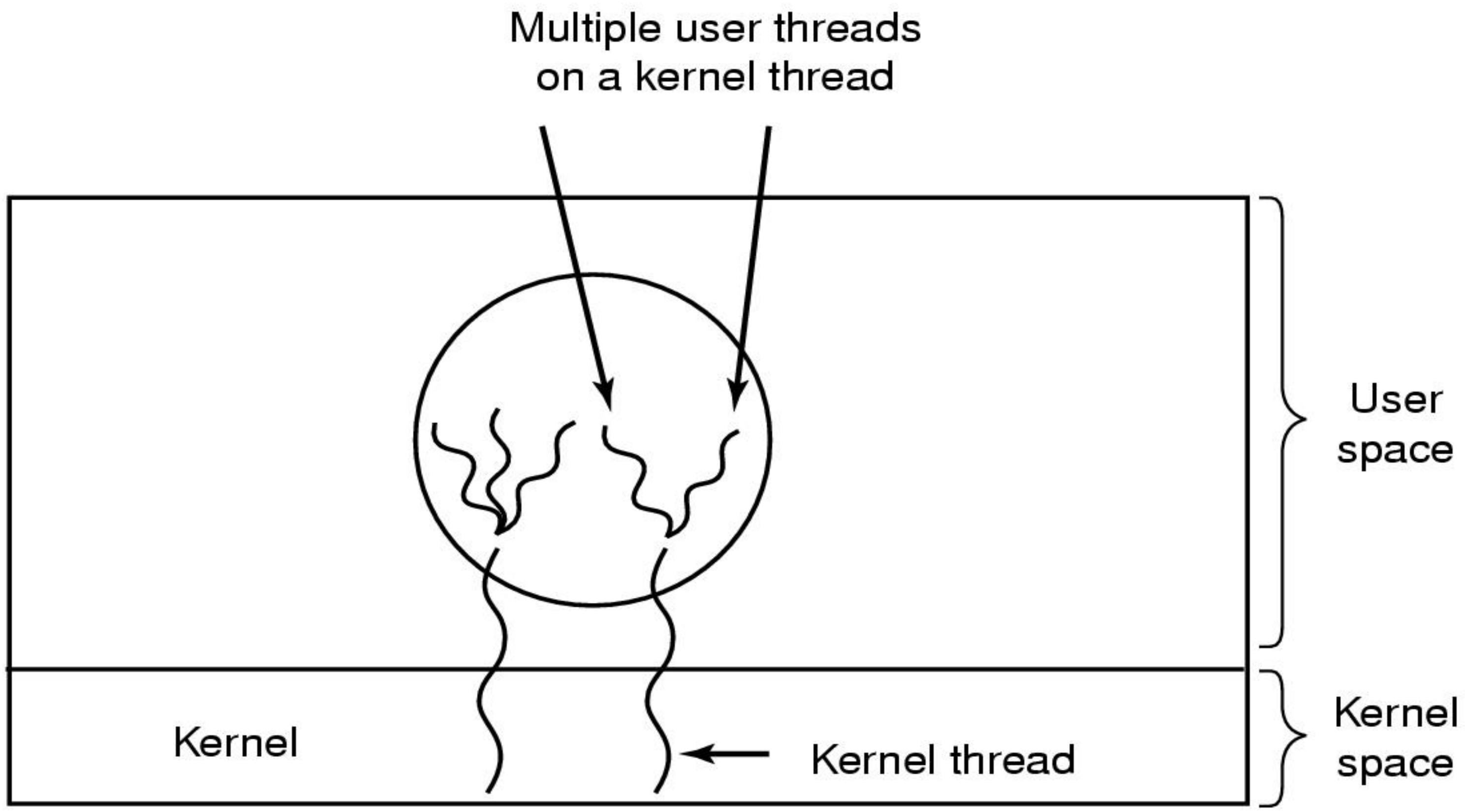
Source: Anderson, T. et al, “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, ACM TOCS, February 1992.

3) Hybrid ULT/KLT Approaches

- Thread creation done in the user space.
- Bulk of scheduling and synchronization of threads done in the user space.
- The programmer may adjust the number of KLTs.
- May combine the best of both approaches.
- Example is Solaris prior to version 9.

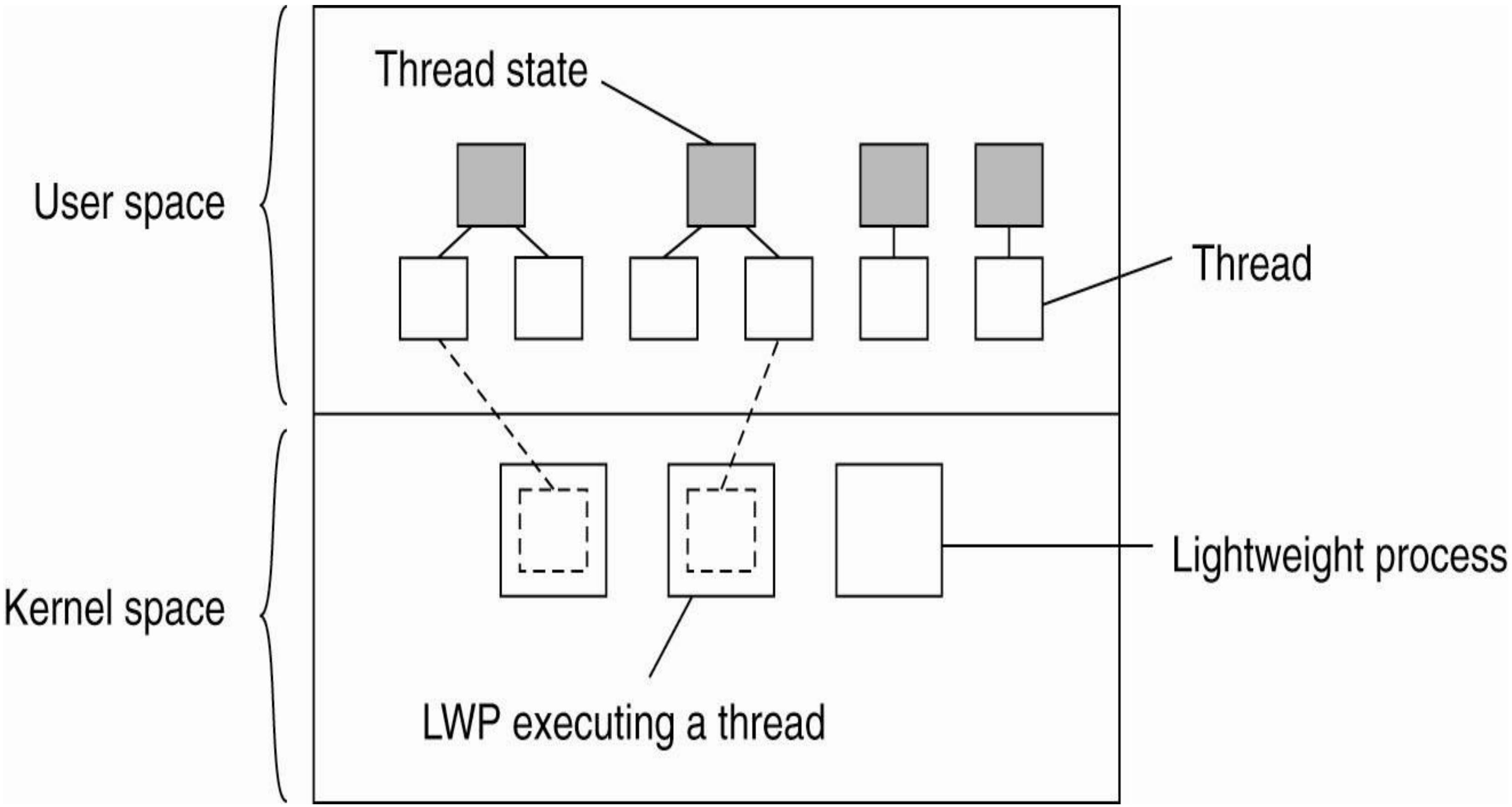


Hybrid Implementation (1)

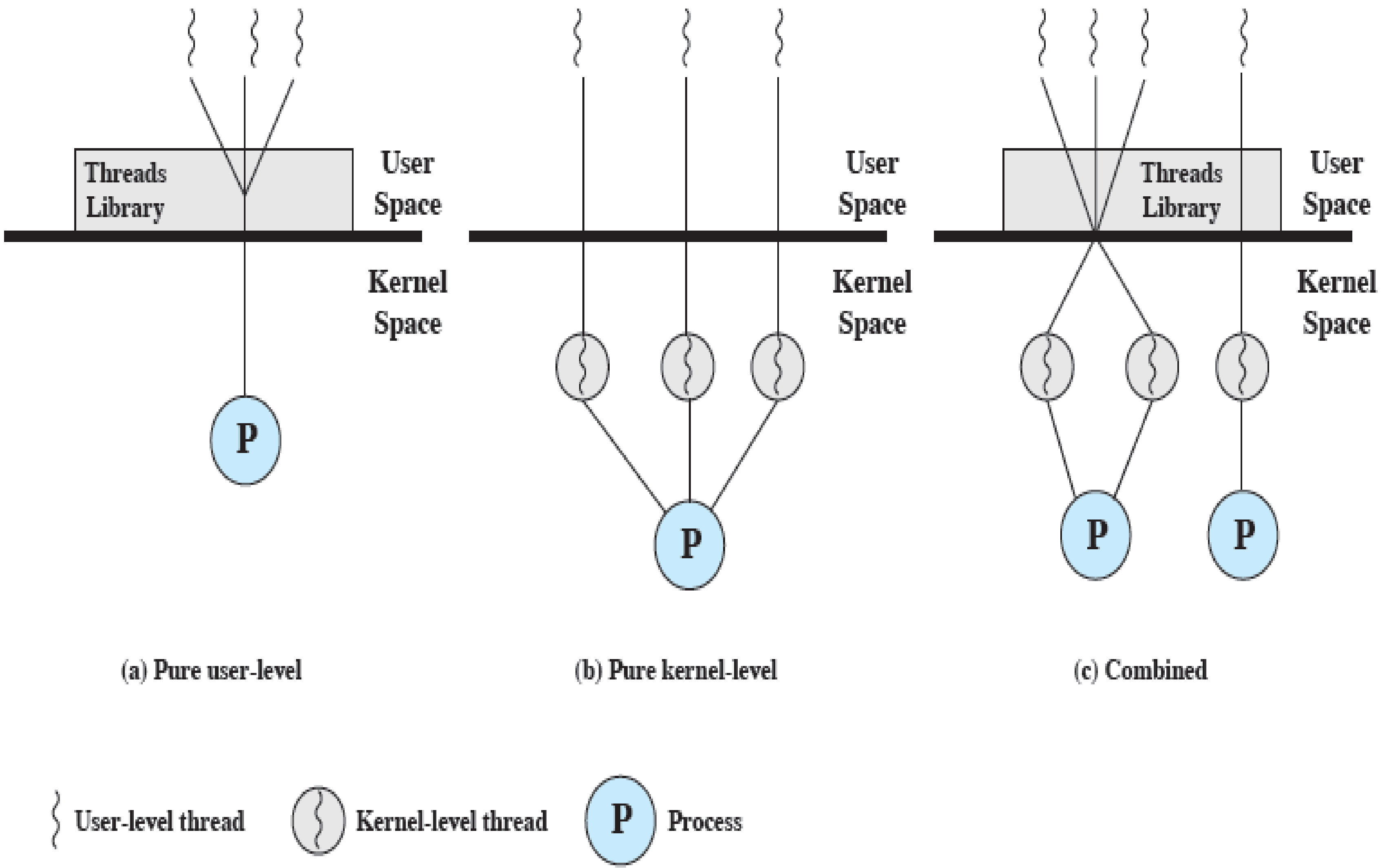


Multiplexing user-level threads onto kernel-level threads.

Hybrid Implementation (2)

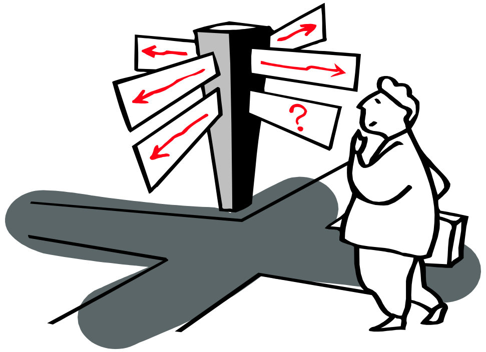


ULT, KLT and Combined Approaches



Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
- Two-level Model

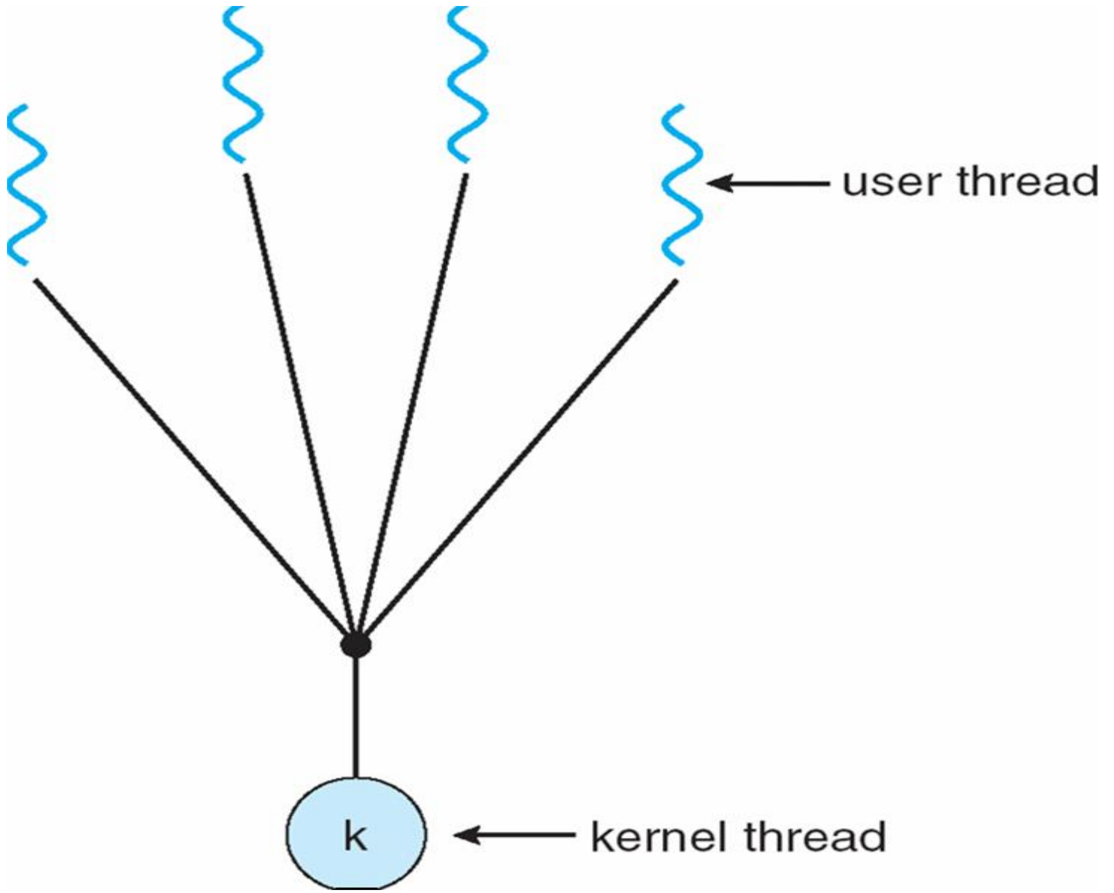


Relationship between Threads and Processes

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

Many-to-One Model (1)

- Many user-level threads mapped to single kernel-level thread.

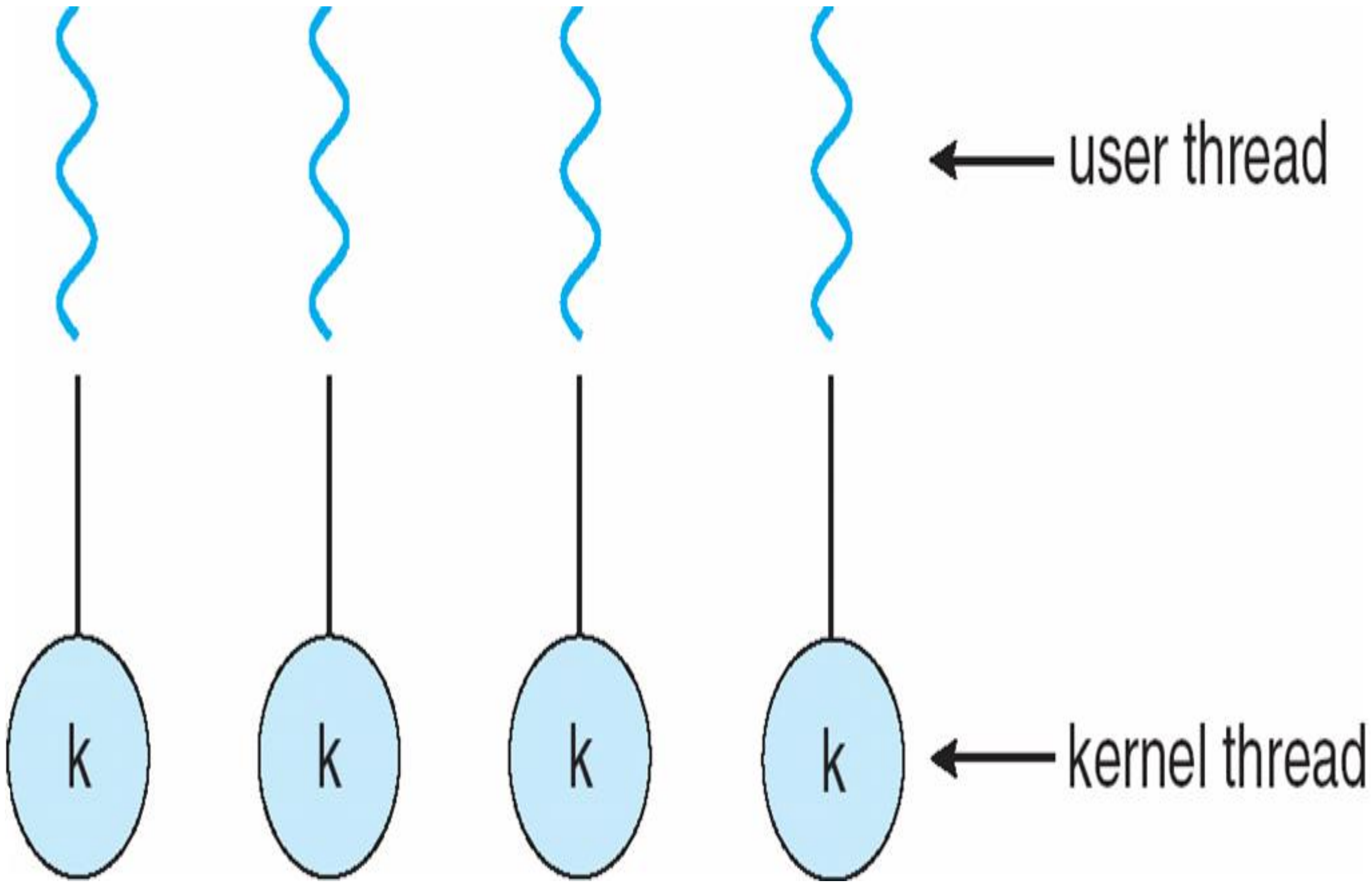


Many-to-One Model (2)

- One thread blocking causes all to block.
- Multiple threads may not run in parallel on multi-core system because only one may be in kernel at a time.
- Few systems currently use this model.
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

One-to-One Model (1)

- Each user-level thread maps to kernel-level thread.

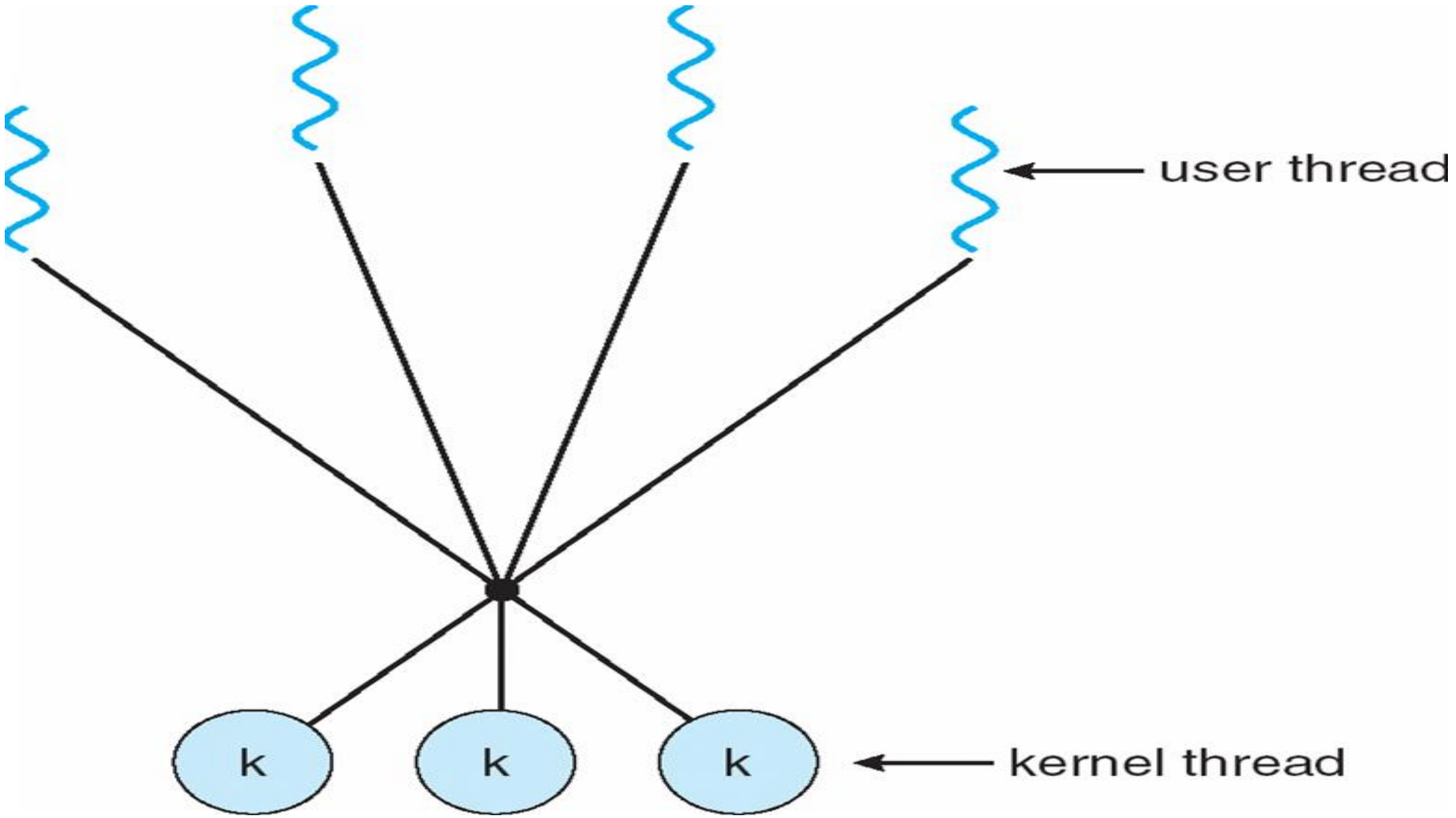


One-to-One Model (2)

- Creating a user-level thread creates a kernel thread.
- More concurrency than many-to-one.
- Number of threads per process sometimes restricted due to overhead.
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

Many-to-Many Model (1)

Allows many user level threads to be mapped to many kernel threads.

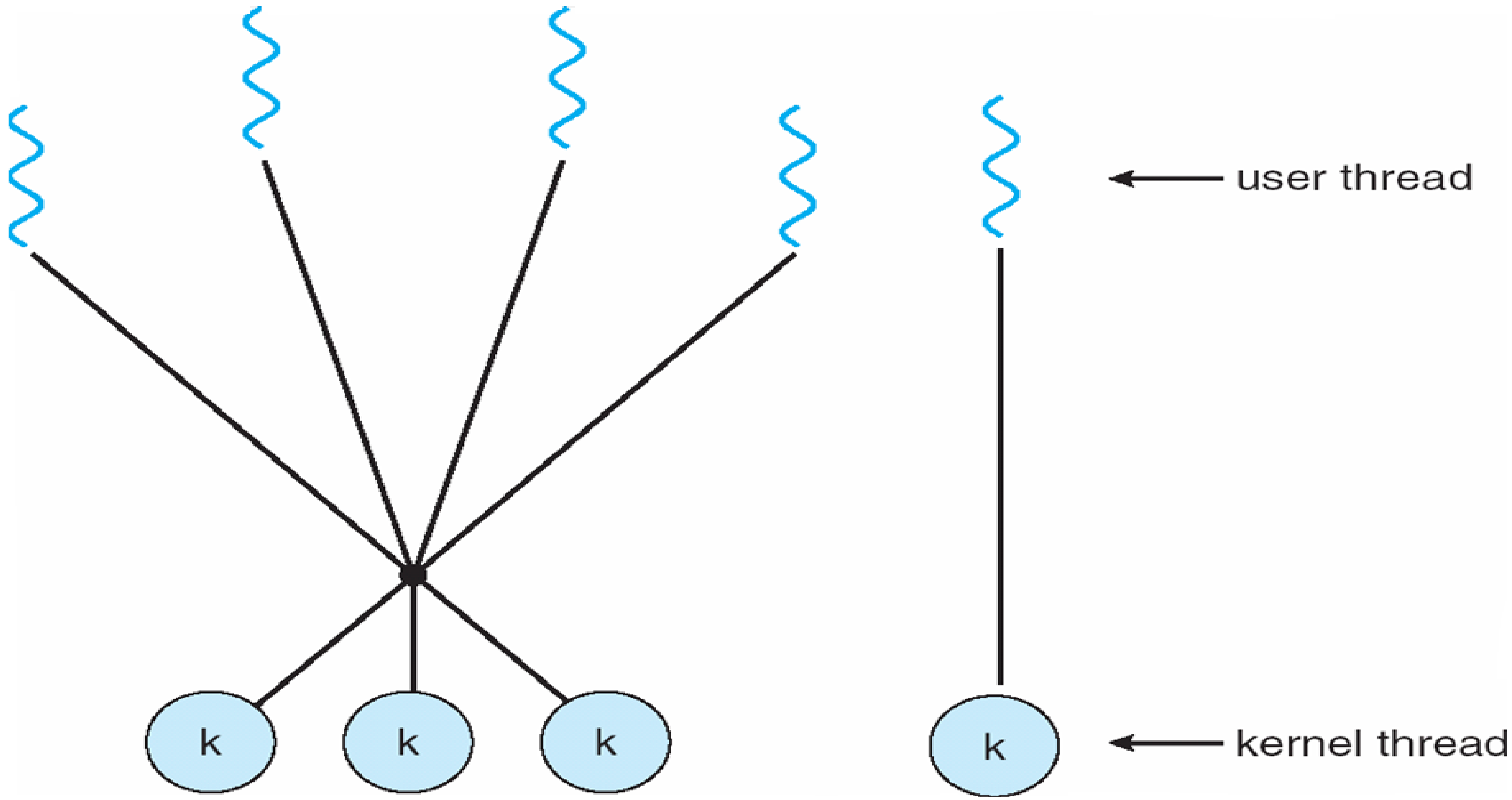


Many-to-Many Model (2)

- Allows the operating system to create a sufficient number of kernel threads.
- Solaris prior to version 9.
- Windows with the *ThreadFiber* package.

Two-level Model (1)

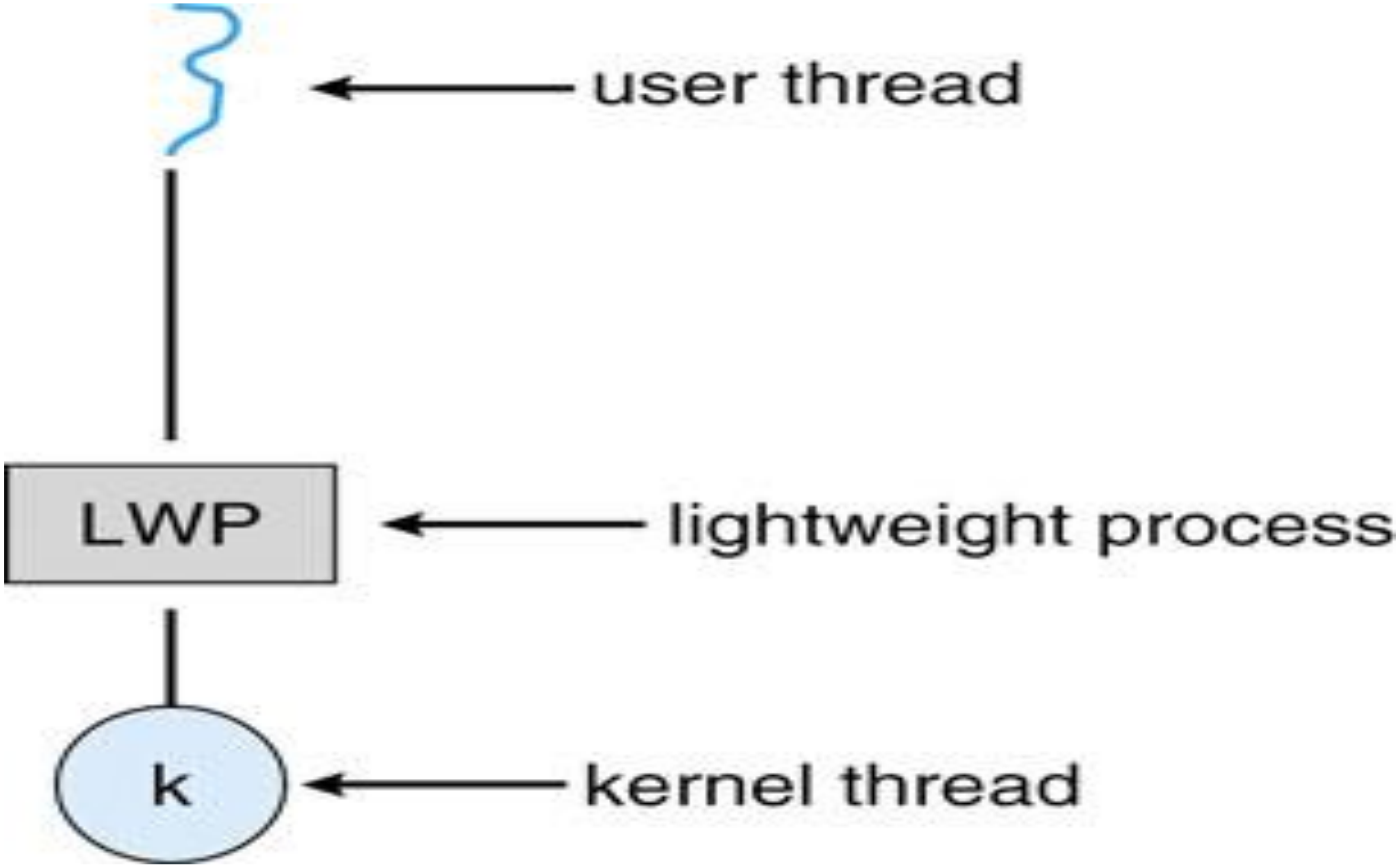
- Similar to Many-to-Many model, except that it allows a user thread to be **bound** to kernel thread.



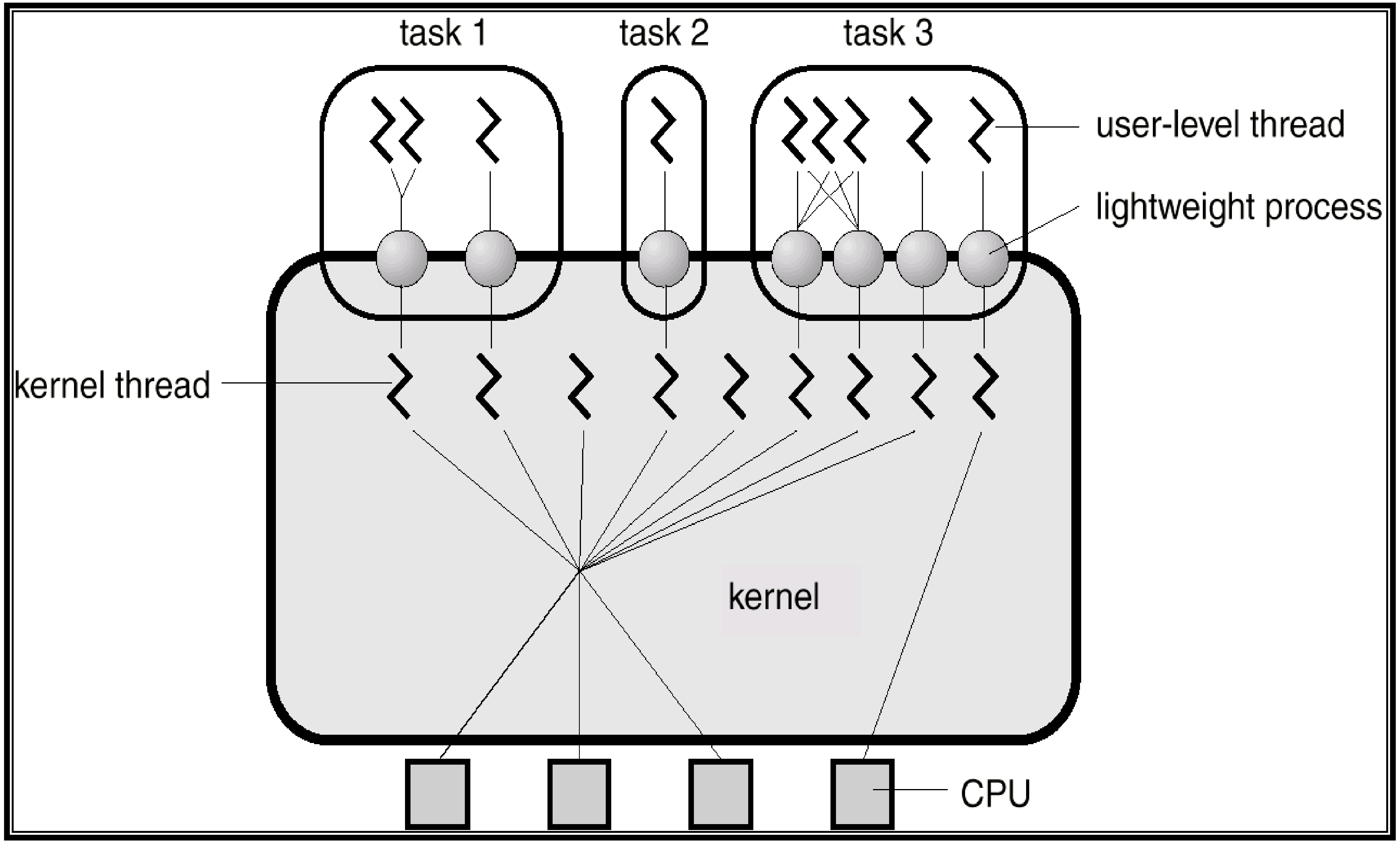
Two-level Model (2)

- Examples:
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Lightweight Process (LWP)

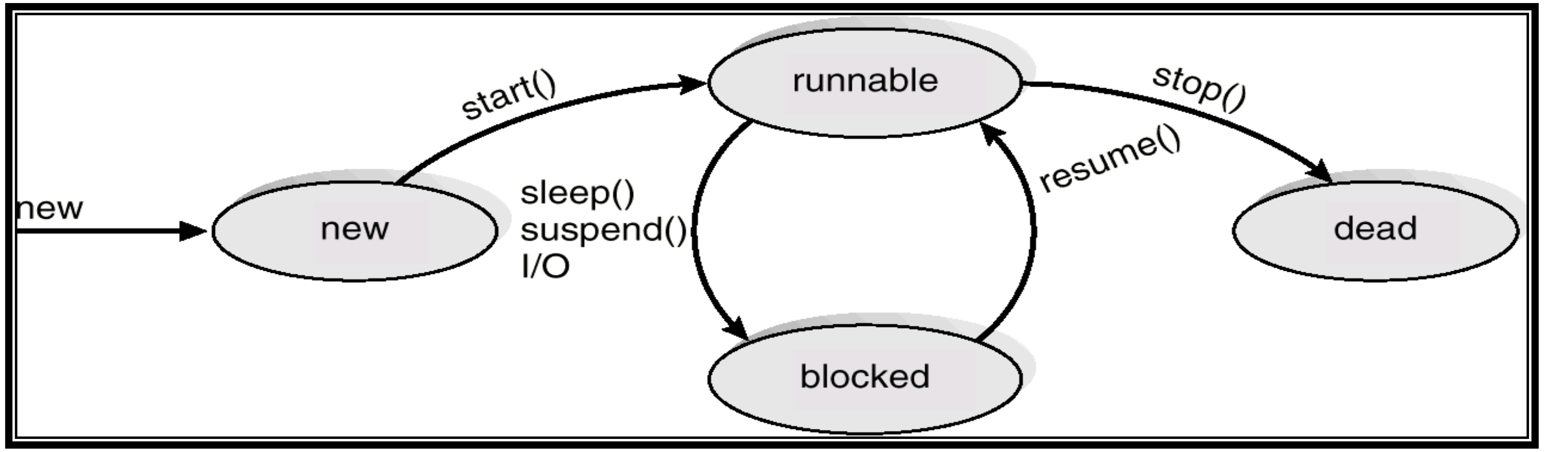


Solaris 2 Threads



Java Threads

- Java threads are managed by the JVM.
- Typically implemented using the threads model provided by underlying OS.
- Java threads may be created by:
  - Extending Thread class (at language-level)
  - Implementing the Runnable interface



Threading Issues

- Semantics of **fork()** and **exec()** system calls
  - Does **fork()** duplicate only the calling thread or all threads?
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-local storage
- Scheduler activations

Thread Cancellation

- Terminating a thread before it has finished.
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately.
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

Signal Handling (1)

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals:
  1. Signal is generated by particular event.
  2. Signal is delivered to a process.
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has default handler that kernel runs when handling signal:
  - User-defined signal handler can override default.
  - For single-threaded, signal delivered to process.

Signal Handling (2)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies.
  - Deliver the signal to every thread in the process.
  - Deliver the signal to certain threads in the process.
  - Assign a specific thread to receive all signals for the process.

Thread Pools

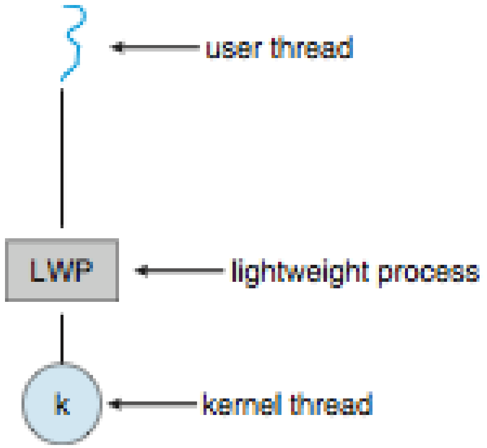
- Create a number of threads in a pool where they await work.
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool.

Thread-Local Storage

- Thread-local storage (TLS) allows each thread to have its own copy of data.
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool).
- Different from local variables:
  - Local variables visible only during single function invocation.
  - TLS visible across function invocations.
- Similar to `static` data:
  - TLS is unique to each thread.

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application/
- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP):
  - Appears to be a virtual processor on which process can schedule user thread to run.
  - Each LWP attached to kernel thread.
  - How many LWPs to create?
- Scheduler activations provide upcalls – a communication mechanism from the kernel to the upcall handler in the thread library.
- This communication allows an application to maintain the correct number kernel threads.



Operating System Examples

- Windows Threads
- Linux Threads

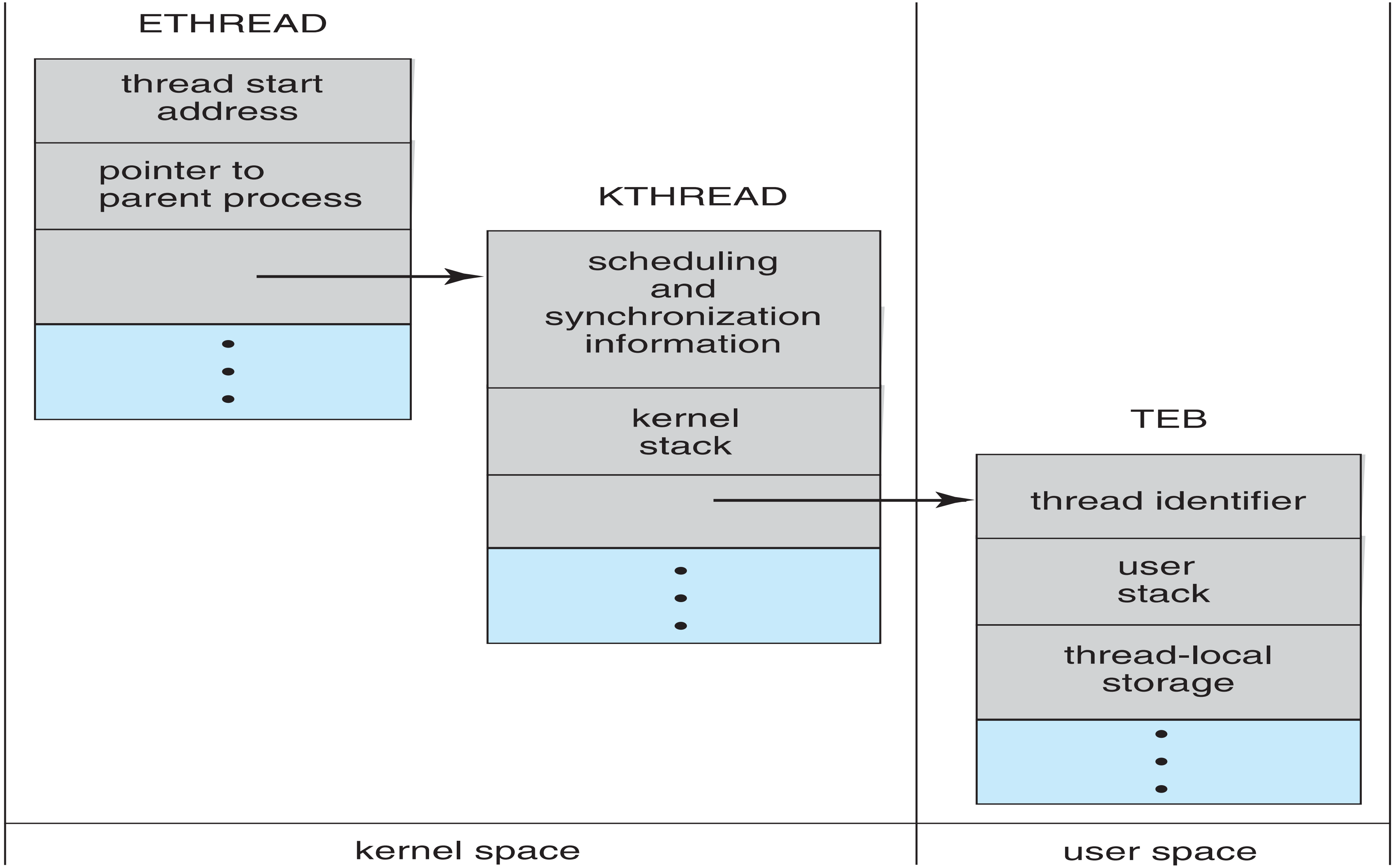
Windows Threads (1)

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7.
- Implements the one-to-one mapping, kernel-level.
- Each thread contains:
  - A thread id.
  - Register set representing state of processor.
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode.
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs).
- The register set, stacks, and private storage area are known as the context of the thread.

Windows Threads (2)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space.
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space.
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space.

Windows Threads Data Structures



Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads.***
- Thread creation is done through **`clone()`** system call.
- **`clone()`** allows a child task to share the address space of the parent task (process).
  - Flags control behavior.

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.

- **`struct task_struct`** points to process data structures (shared or unique).

Thread Python Implementation

Starting a new Thread

EXAMPLE:

```
#!/usr/bin/python
import thread
import time
# Define a function for the thread
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )
# Create two threads as follows
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"
while 1:
   pass
```

Creating Thread using Threading Module

- To implement a new thread using the threading module, need to do the following:
  - Define a new subclass of the Thread class.
  - Override the __init__(self [,args]) method to add additional arguments.
  - override the run(self [,args]) method to implement what the thread should do when started.
- Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the start(), which will in turn call run() method.

EXAMPLE:

```
#!/usr/bin/python
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter)
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name
def print_time(threadName, delay, counter):
     While counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName,
time.ctime(time.time()))
        counter -= 1
```

```
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
print "Exiting Main Thread"
```

When the above code is executed, it produces the following result:

```
Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2
```

Synchronizing Threads

- The threading module provided with Python includes a simple-toimplement
- locking mechanism that will allow you to synchronize threads
- ● A new lock is created by calling the Lock() method, which returns the new
- lock.
- ● The acquire(blocking) method of the new lock object would be used to
- force threads to run synchronously
- ● The optional blocking parameter enables you to control whether the
- thread will wait to acquire the lock.
- ● If blocking is set to 0, the thread will return immediately with a 0 value if
- the lock cannot be acquired and with a 1 if the lock was acquired. If
- blocking is set to 1, the thread will block and wait for the lock to be
- released.
- ● The release() method of the the new lock object would be used to release
- the lock when it is no longer required.

SCHOOL OF HUMAN SCIENCES &

EXAMPLE:

```
#!/usr/bin/python
import threading
import time
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        # Get lock to synchronize threads
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # Free lock to release next thread
        threadLock.release()
def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName,
time.ctime(time.time()))
        counter -= 1
threadLock = threading.Lock()
threads = []
```

```
# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)
# Start new Threads
thread1.start()
thread2.start()
# Add threads to thread list
threads.append(thread1)
threads.append(thread2)
# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
```

When the above code is executed, it produces the following result:

```
Starting Thread-1
Starting Thread-2
Thread-1: Thu Mar 21 09:11:28 2013
Thread-1: Thu Mar 21 09:11:29 2013
Thread-1: Thu Mar 21 09:11:30 2013
Thread-2: Thu Mar 21 09:11:32 2013
Thread-2: Thu Mar 21 09:11:34 2013
Thread-2: Thu Mar 21 09:11:36 2013
Exiting Main Thread
```

MultiProcessing

```python
import multiprocessing
def spawn():
  print('test!')

if __name__ == '__main__':
  for i in range(5):
    p = multiprocessing.Process(target=spawn)
    p.start()
```

If you have a shared database, you want to make sure that you're waiting for relevant processes to finish before starting new ones.

```python
for i in range(5):
  p = multiprocessing.Process(target=spawn)
  p.start()
  p.join() # this line allows you to wait for processes
```

MultiProcessing

```python
import multiprocessing
def spawn(num):
  print(num)

if __name__ == '__main__':
  for i in range(25):
    ## right here
    p = multiprocessing.Process(target=spawn, args=(i,))
    p.start()
```

R Implementation

Methods of Parallelization

- There are two main ways in which code can be parallelized, via *sockets* or via *forking*. These function slightly differently:
  - The *socket* approach launches a new version of R on each core. Technically this connection is done via networking (e.g. the same as if you connected to a remote server), but the connection is happening all on your own computer[3] I mention this because you may get a warning from your computer asking whether to allow R to accept incoming connections, you should allow it.
  - The *forking* approach copies the entire current version of R and moves it to a new core.

Methods of Parallelization

- Socket:
  - Pro: Works on any system (including Windows).
  - Pro: Each process on each node is unique so it can't cross-contaminate.
  - Con: Each process is unique so it will be slower
  - Con: Things such as package loading need to be done in each process separately. Variables defined on your main version of R don't exist on each core unless explicitly placed there.
  - Con: More complicated to implement.
- Forking:
  - Con: Only works on POSIX systems (Mac, Linux, Unix, BSD) and not Windows.
  - Con: Because processes are duplicates, it can cause issues specifically with random number generation (which should usually be handled by parallel in the background) or when running in a GUI (such as RStudio). This doesn't come up often, but if you get odd behavior, this may be the case.
  - Pro: Faster than sockets.
  - Pro: Because it copies the existing version of R, your entire workspace exists in each process.
  - Pro: Trivially easy to implement.

Forking with `mclapply`

The most straightforward way to enable parallel processing is by switching from using `lapply` to `mclapply`. (Note I'm using `system.time` instead of `profvis` here because I only care about running time, not profiling.)

```
library(lme4)
```

```
## Loading required package: Matrix
```

```
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time(save1 <- lapply(1:100, f))
```

```
##    user  system elapsed
##   2.048   0.019   2.084
```

```
system.time(save2 <- mclapply(1:100, f))
```

```
##    user  system elapsed
##   1.295   0.150   1.471
```

Using sockets with `parLapply`

As promised, the sockets approach to parallel processing is more complicated and a bit slower, but works on Windows systems. The general process we'll follow is

1. Start a cluster with n nodes.
2. Execute any pre-processing code necessary in each node (e.g. loading a package)
3. Use `par*apply` as a replacement for `*apply`. Note that unlike `mcapply`, this is *not* a drop-in replacement.
4. Destroy the cluster (not necessary, but best practices).

Starting a cluster

The function to start a cluster is `makeCluster` which takes in as an argument the number of cores:

```
numCores <- detectCores()
numCores
```

```
## [1] 4
```

```
cl <- makeCluster(numCores)
```

The function takes an argument `type` which can be either `PSOCK` (the socket version) or `FORK` (the fork version). Generally, `mclapply` should be used for the forking approach, so there's no need to change this.

```r
### lapply
library(parallel)
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  library(lme4)
  save1 <- lapply(1:100, f)
})

### mclapply
library(parallel)
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  library(lme4)
  save2 <- mclapply(1:100, f)
})


### mclapply
library(parallel)
f <- function(i) {
  lmer(Petal.Width ~ . - Species + (1 | Species), data = iris)
}

system.time({
  cl <- makeCluster(detectCores())
  clusterEvalQ(cl, library(lme4))
  save3 <- parLapply(cl, 1:100, f)
  stopCluster(cl)
})
```

lapply	mclapply	parLapply
4.237	4.087	6.954

```r
library(parallel)
library(MASS)

starts <- rep(100, 40)
fx <- function(nstart) kmeans(Boston, 4, nstart=nstart)
numCores <- detectCores()
numCores
```

```
## [1] 8
```

```r
system.time(
  results <- lapply(starts, fx)
)
```

```
##    user  system elapsed
##   1.346   0.024   1.372
```

```r
system.time(
  results <- mclapply(starts, fx, mc.cores = numCores)
)
```

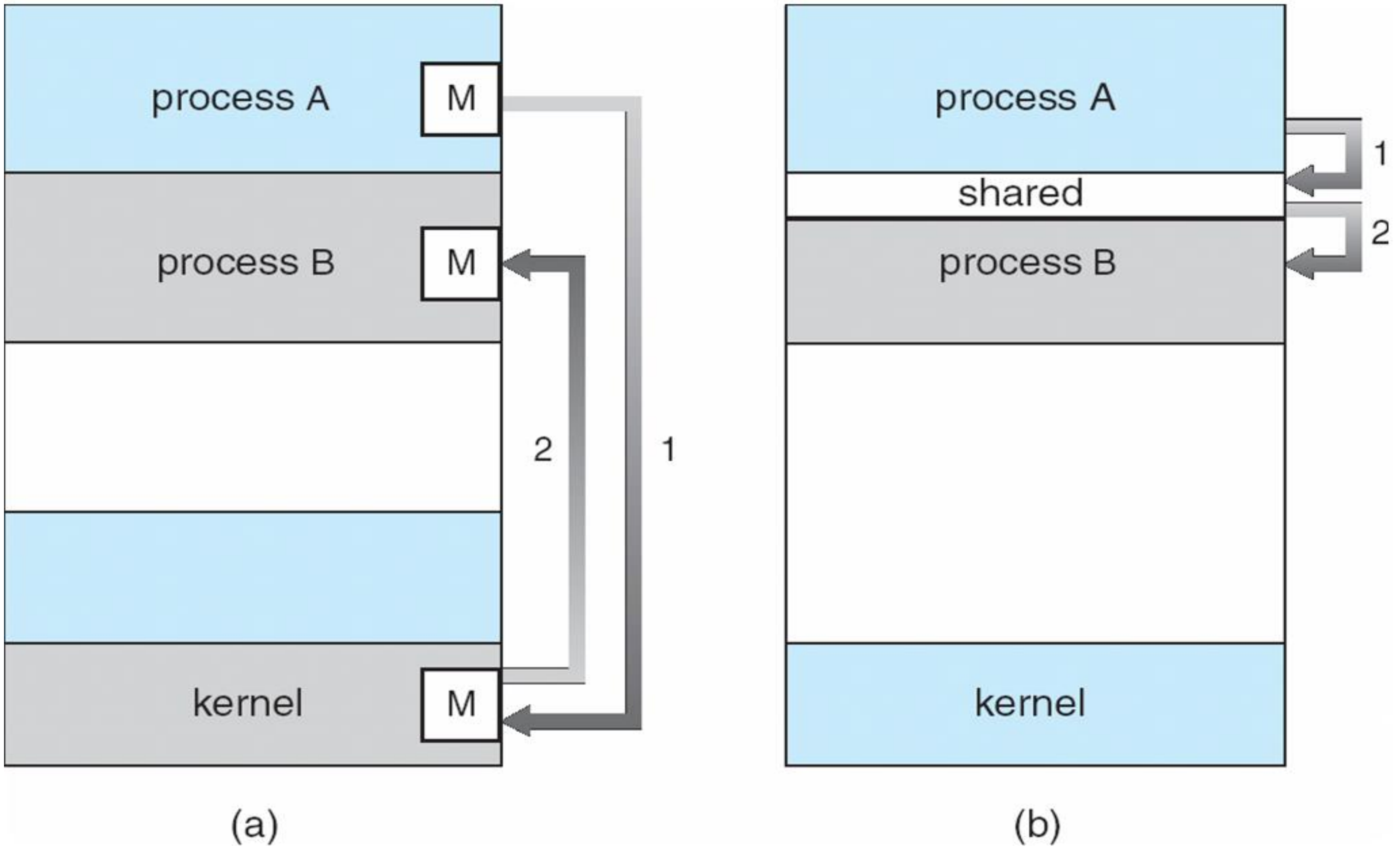
```
##    user  system elapsed
##   0.801   0.178   0.367
```

Introduction to Cooperating Process

Introduction to Cooperating Processes

- Processes within a system may be independent or cooperating.
- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by other processes, including sharing data.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Cooperation Models



Cooperation among Processes by Sharing

- Processes use and update shared data such as shared variables, memory, files, and databases.
- Writing must be mutually exclusive to prevent a race condition leading to inconsistent data views.
- Critical sections are used to provide this data integrity.
- A process requiring the critical section must not be delayed indefinitely; no deadlock or starvation.

Cooperation among Processes by Communication

- Communication by messages provides a way to synchronize, or coordinate, the various activities.
- Possible to have deadlock –
  - each process waiting for a message from the other process.
- Possible to have starvation –
  - two processes sending a message to each other while another process waits for a message.

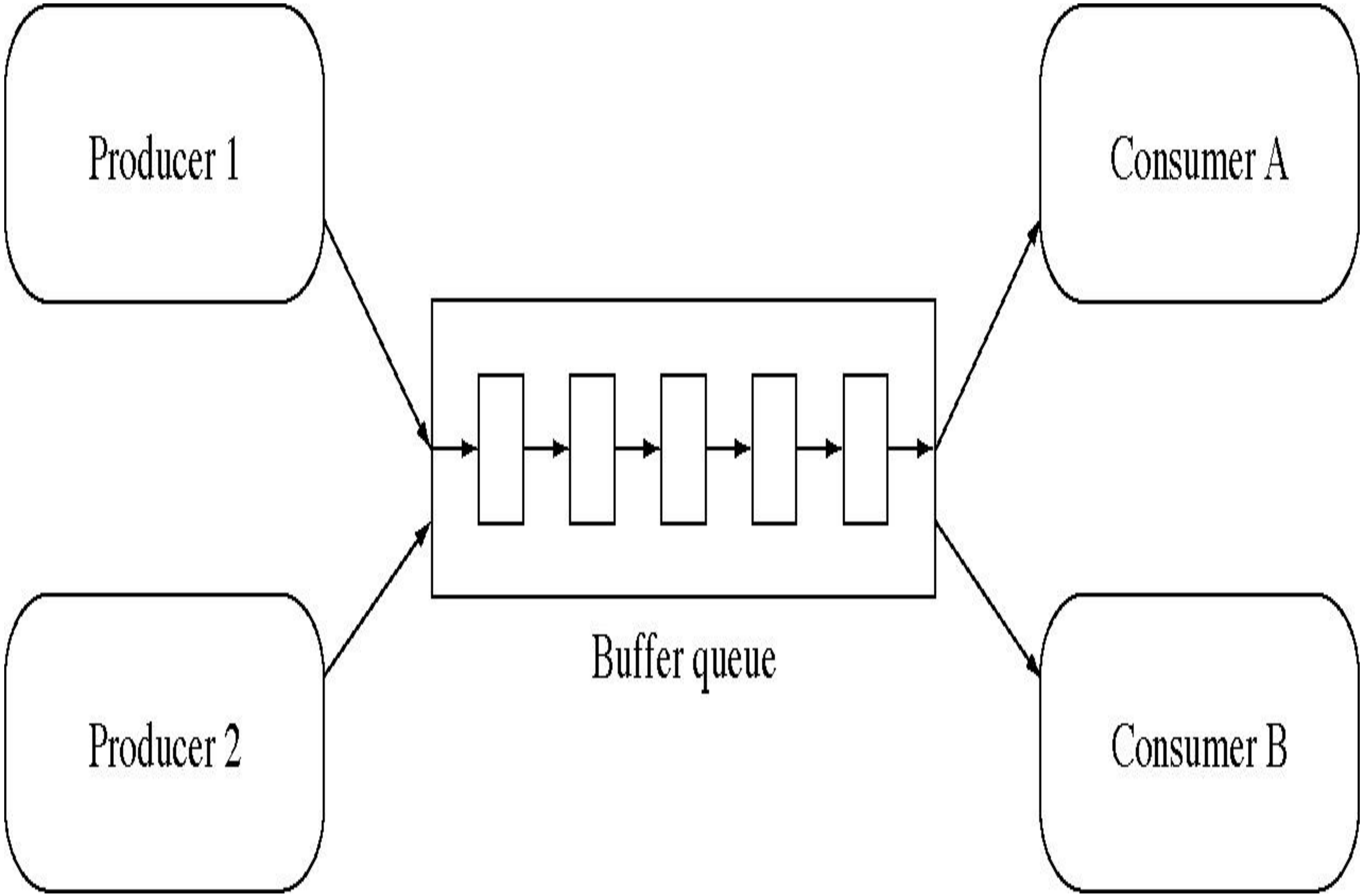
Producer/Consumer (P/C) Problem (1)

- Paradigm for cooperating processes – *Producer* process produces information that is consumed by a *Consumer* process.
  - Example 1: a print program produces characters that are consumed by a printer.
  - Example 2: an assembler produces object modules that are consumed by a loader.
  - Example 3: a client produces a message that a server could consume it.

Producer/Consumer (P/C) Problem (2)

- We need a buffer to hold items that are produced and later consumed:
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.

Multiple Producers and Consumers

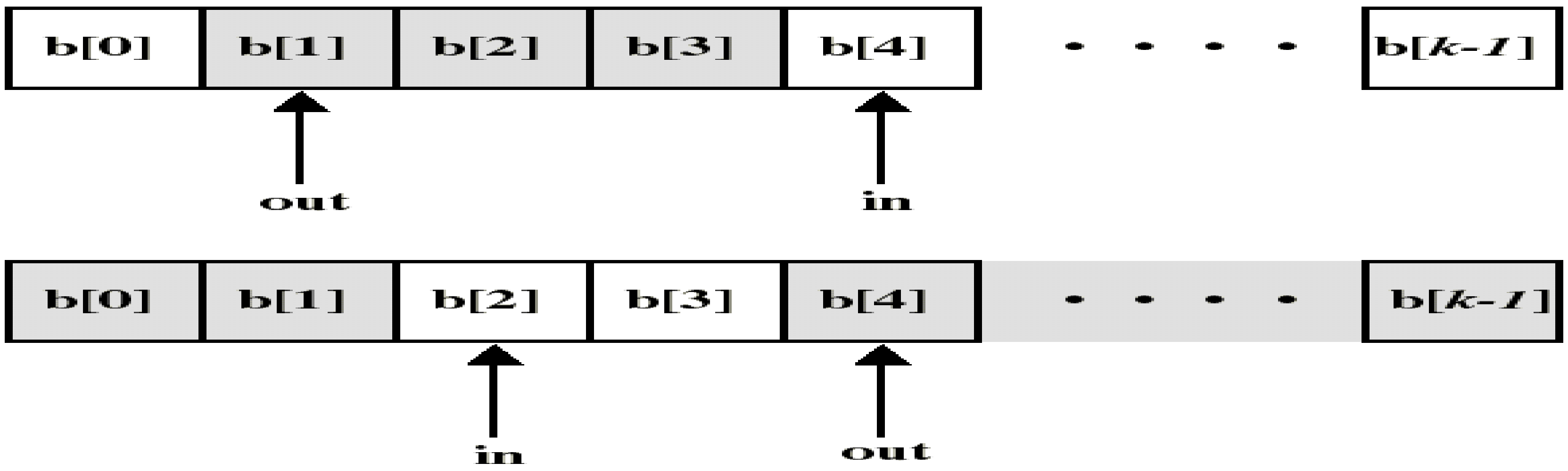


Producer/Consumer (P/C) Dynamics

- A producer process produces information that is consumed by a consumer process.
- At any time, a producer activity may create some data.
- At any time, a consumer activity may want to accept some data.
- The data should be saved in a buffer until they are needed.
- If the buffer is finite, we want a producer to block if its new data would overflow the buffer.
- We also want a consumer to block if there are no data available when it wants them.

Idea for Producer/Consumer Solution

- The bounded buffer is implemented as a circular array with 2 logical pointers: **`in`** and **`out`**.
- The variable **`in`** points to the next free position in the buffer.
- The variable **`out`** points to the first full position in the buffer.



Problems with concurrent execution

- Concurrent processes (or threads) often need to share data (maintained either in shared memory or files) and resources.
- If there is no controlled access to shared data, some processes will obtain an inconsistent view of this data.
- The action performed by concurrent processes will then depend on the order in which their execution is interleaved.

Example of inconsistent view

- 3 variables: A, B, C which are shared by thread T1 and thread T2.
- T1 computes C = A+B.
- T2 transfers amount X from A to B
  - T2 must do: A = A-X and B = B+X (so that A+B is unchanged).
- But if T1 computes A+B after T2 has done A = A-X but before B = B+X.
- Then T1 will not obtain the correct result for C = A+B.

Data Consistency

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffer. We can do so by having an integer count that keeps track of the number of items in the buffer.
- Initially, the count is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes a item.

This is the Race Condition

- **Race condition**: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must coordinate or be **synchronized**.

Race condition updating a variable (1)

shared double balance;

Code for p1:

```
. . .
balance += amount;
. . .
```

Code for p1:

```
. . .
Load    R1, balance
Load    R2, amount
Add     R1, R2
Store   R1, balance
. . .
```

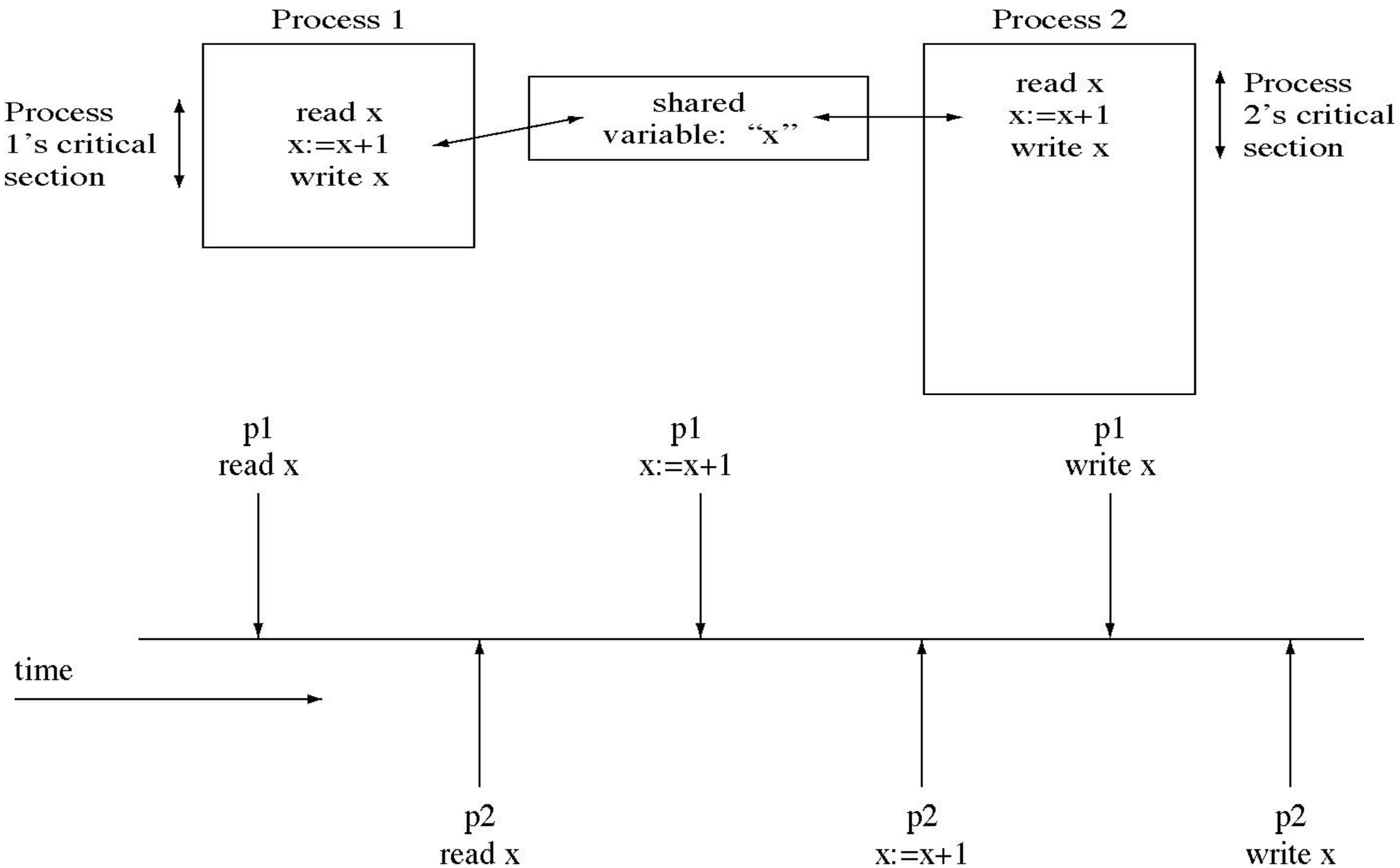
Code for p2:

```
. . .
balance += amount;
. . .
```

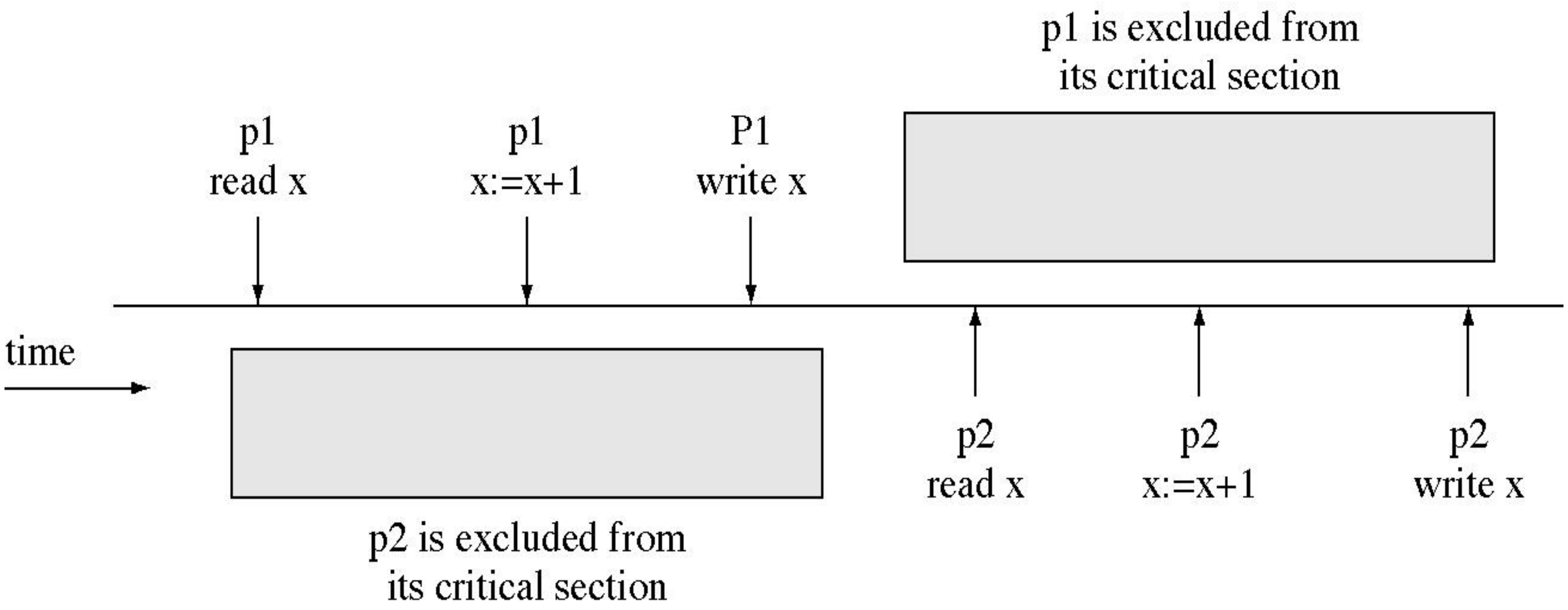
Code for p2:

```
. . .
Load    R1, balance
Load    R2, amount
Add     R1, R2
Store   R1, balance
. . .
```

Race condition updating a variable (2)



Critical section to prevent a race condition



- Multiprogramming allows logical parallelism, uses devices efficiently but we lose correctness when there is a race condition.
- So we forbid logical parallelism inside critical section so we lose some parallelism but we regain correctness.

SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY