UNIT 2

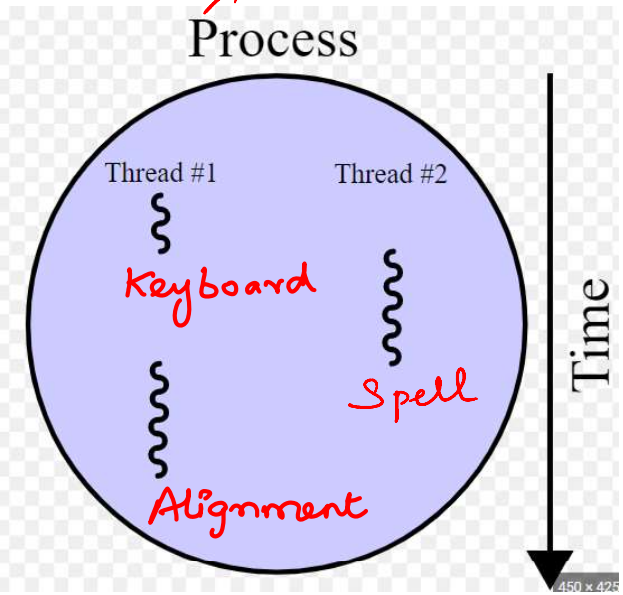
Lorry ⇒ Process (Task)

THREADS → Light weight process

Mini Lorry

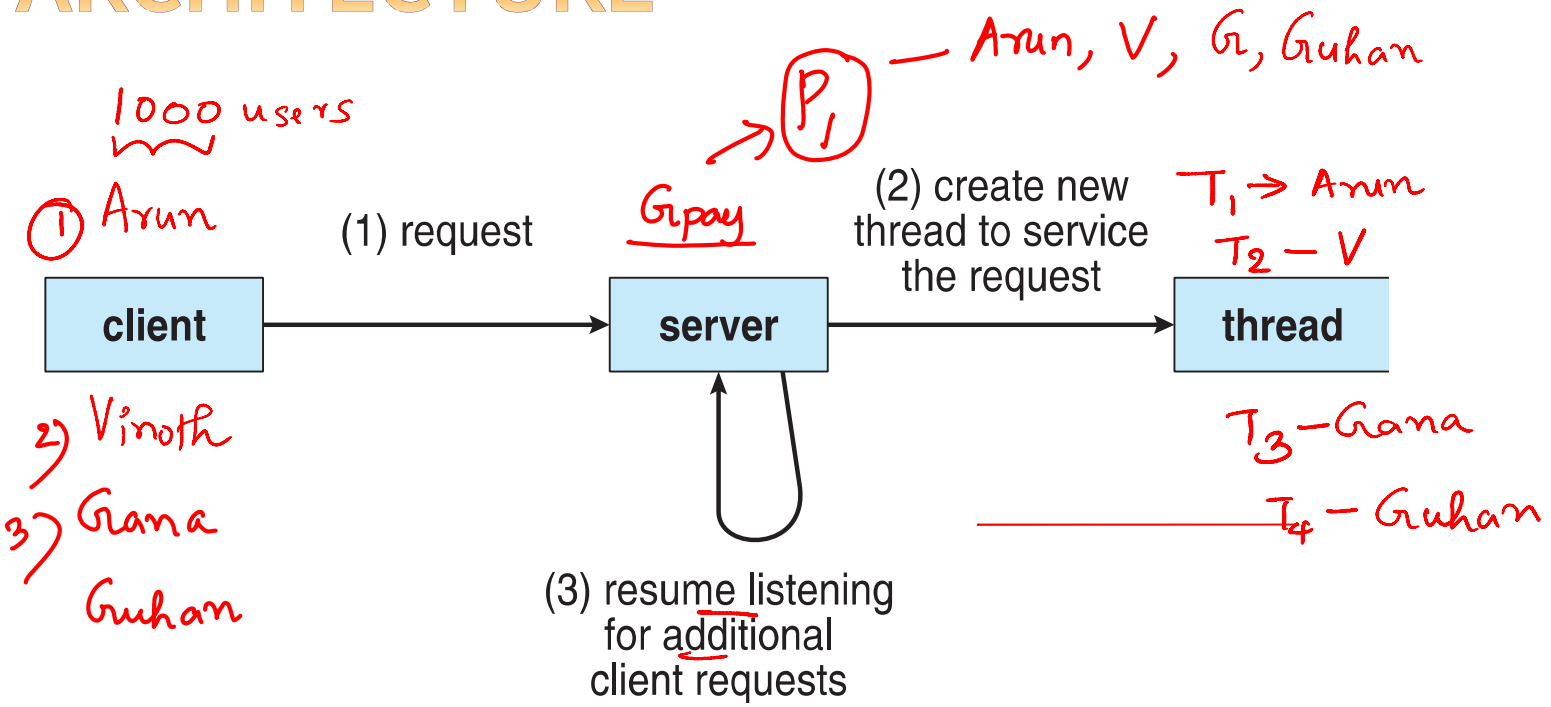
Time↑

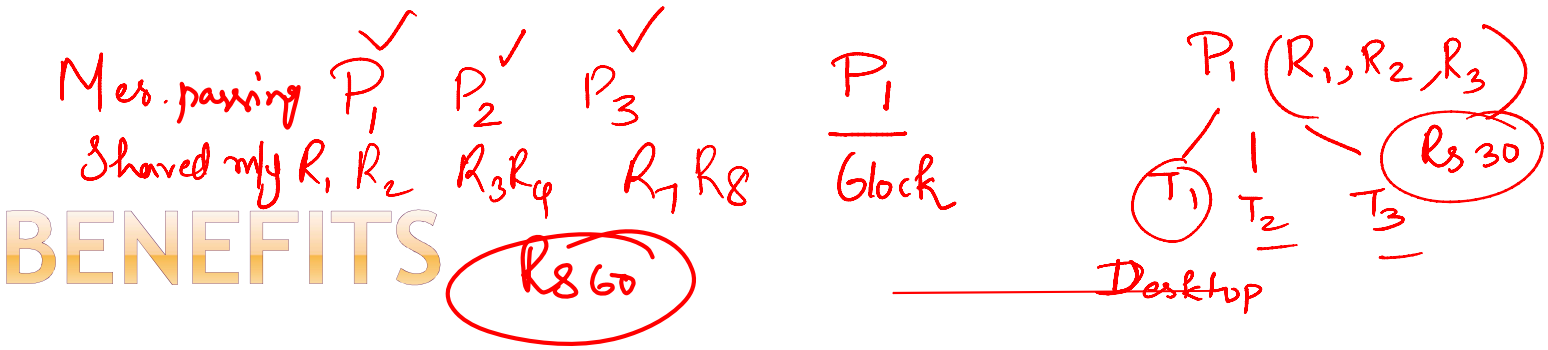
Word



- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

MULTI-THREADED SERVER ARCHITECTURE

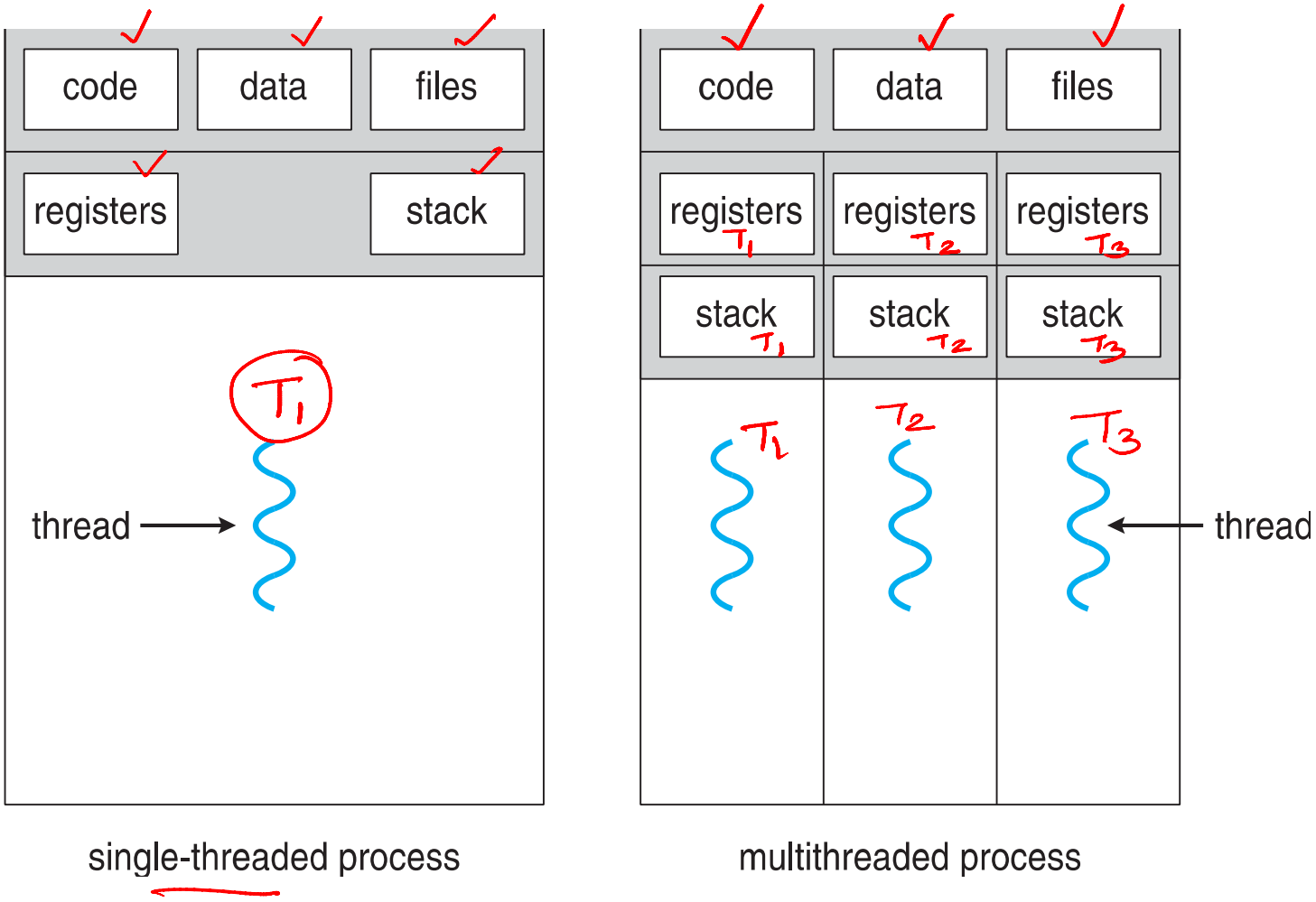




BENEFITS

1) **Responsiveness** - may allow continued execution if part of process is blocked, especially important for user interfaces
2) **Resource Sharing** - threads share resources of process, easier than shared memory or message passing
3) **Economy** - cheaper than process creation, thread switching lower overhead than context switching
4) **Scalability** - process can take advantage of multiprocessor architectures

SINGLE AND MULTITHREADED PROCESSES

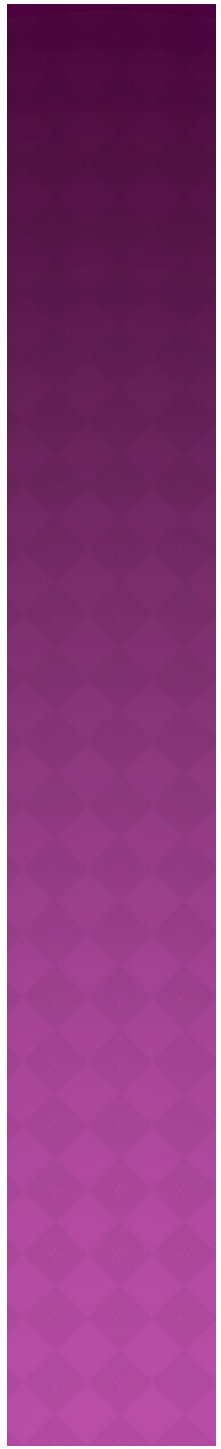


single-threaded process

multithreaded process

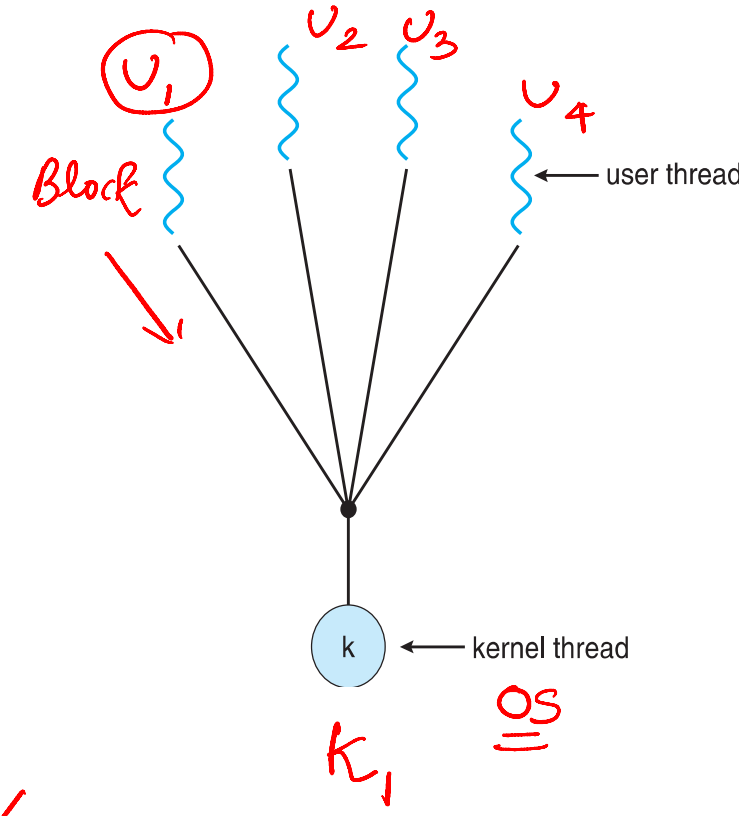
USER THREADS AND KERNEL THREADS

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



MULTITHREADING MODELS- MANY-TO-ONE

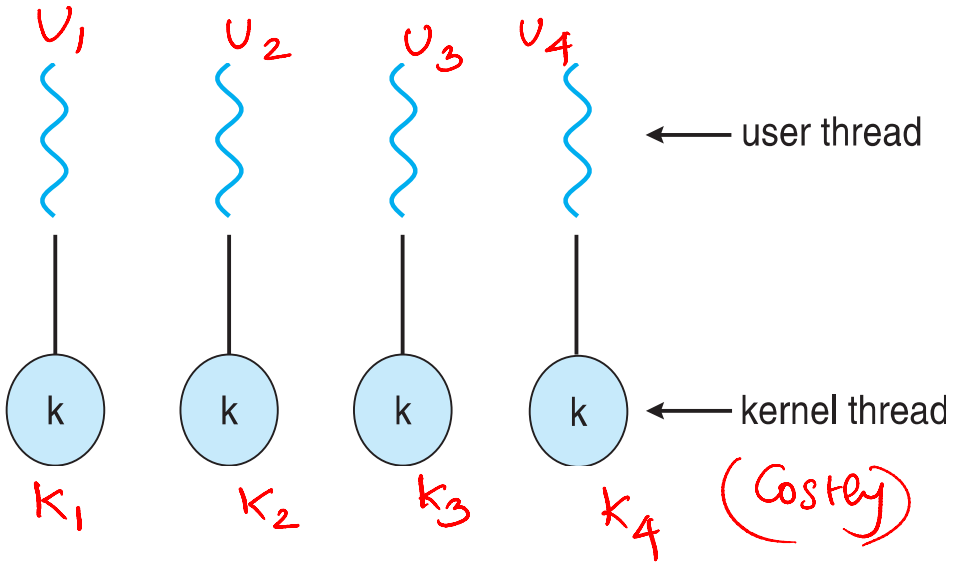
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



User ✓ Kernel

ONE TO ONE

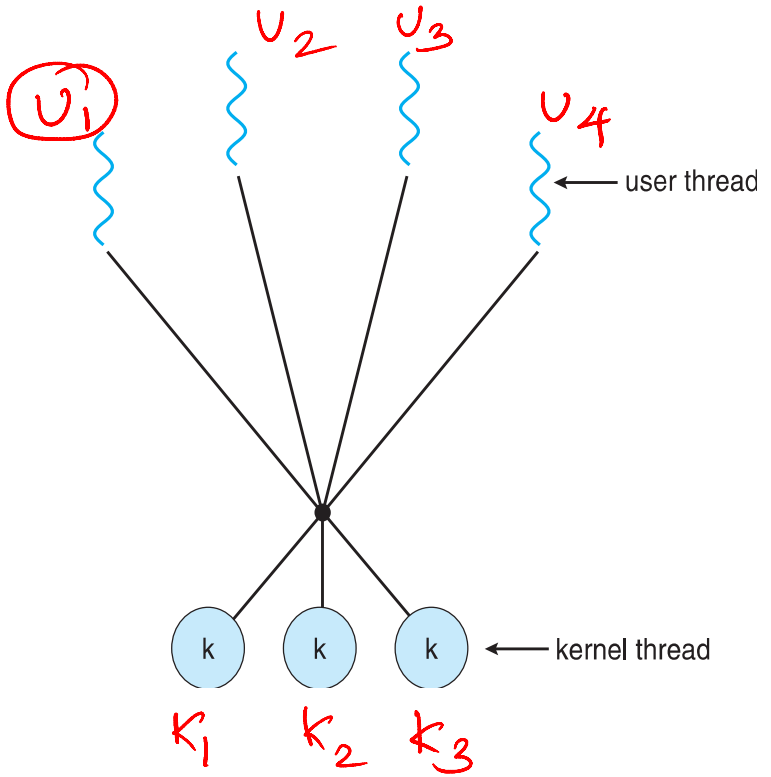
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples ✓
  - Windows ✓
  - Linux ✓
  - Solaris 9 and later



1000 user thread ⇒ 1000 k.t

MANY-TO-MANY MODEL

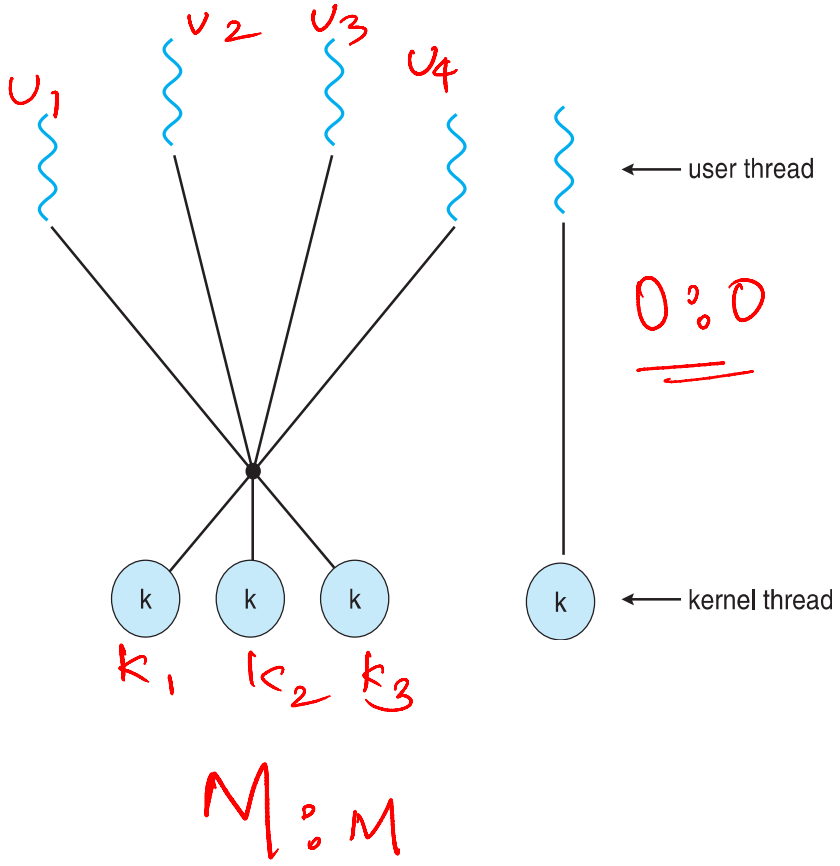
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



M:M + 0:0

TWO-LEVEL MODEL

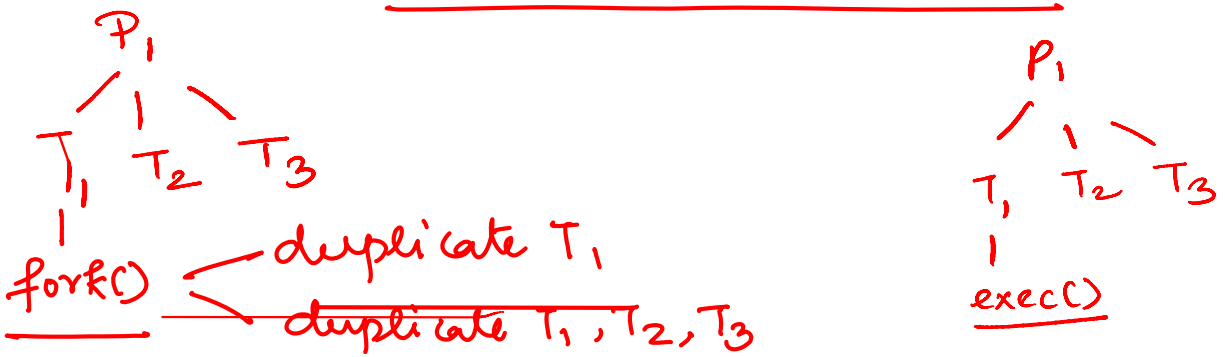
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX ✓



THREADING ISSUES

1) Semantics of **fork()** and **exec()** system calls
2) Signal handling
   - Synchronous and asynchronous
3) Thread cancellation of target thread
   - Asynchronous or deferred
4) Thread-local storage
5) Scheduler Activations

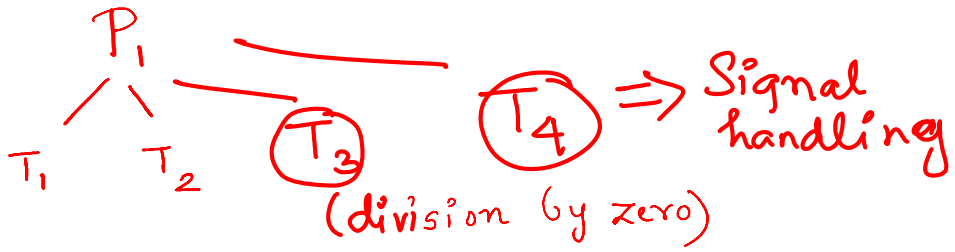
SEMANTICS OF FORK() AND EXEC()



- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal - replace the running process including all threads

SIGNAL HANDLING

- **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



- Where should a signal be delivered for multi-threaded?
  - ✓ Deliver the signal to the thread to which the signal applies T_3
  - ✓ Deliver the signal to every thread in the process T_1, T_2, T_3
  - ✓ Deliver the signal to certain threads in the process T_2, T_3
  - ✓ Assign a specific thread to receive all signals for the process

THREAD CANCELLATION

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state . (default- deferred)

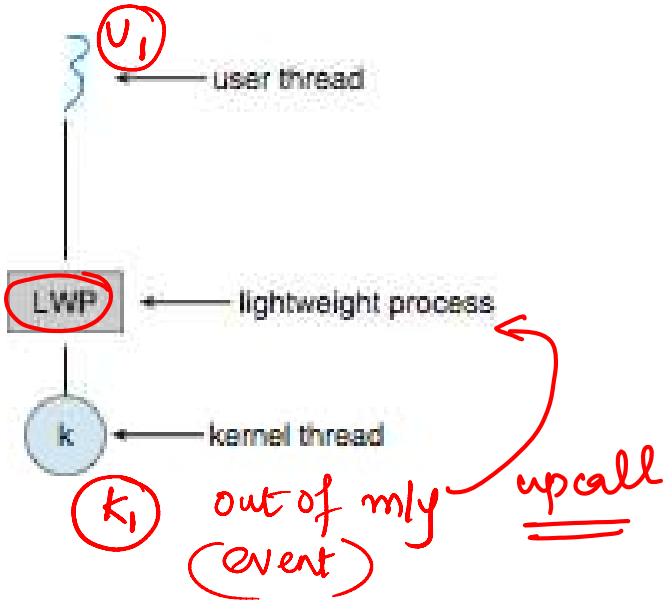
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

THREAD-LOCAL STORAGE

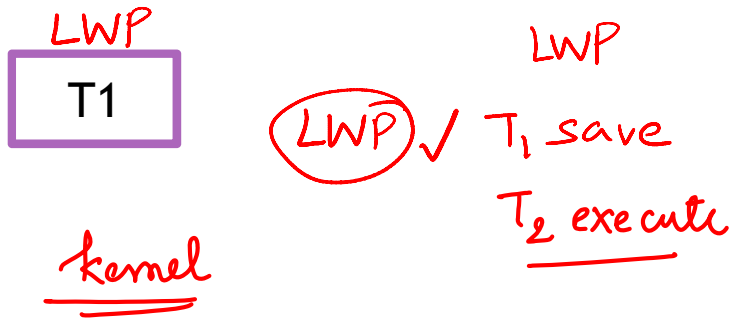
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables ✓
  - Local variables visible only during single function invocation
  - TLS visible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



EXAMPLE



- One event that triggers an upcall occurs when an application thread is about to block.
- In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
- Then the kernel allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, that saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.
- Another thread that is eligible to run on the new virtual processor is scheduled then by the upcall handler.
- Whenever the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run.
- A virtual processor is also required for The upcall handler for this event, and the kernel may allocate a new virtual processor or preempt one of the user threads and then run the upcall handler on its virtual processor.
- The application schedules an eligible thread to run on an available virtual processor, after marking the unblocked thread as eligible to run,

Process	Thread
A process is an instance of a program that is being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.
Each process is treated as a new process by the operating system.	The operating system takes all the user-level threads as a single process.
If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
The data segment and code segment of each process are independent of the other.	Threads share data segment and code segment with their peer threads; hence are the same for other threads also.
The operating system takes more time to terminate a process.	Threads can be terminated in very little time.
New process creation is more time taking as each new process takes all the resources.	A thread needs less time for creation.