



Operating Systems

Lecture 3.3

Achieving Concurrency using Threads

Lecture Agenda



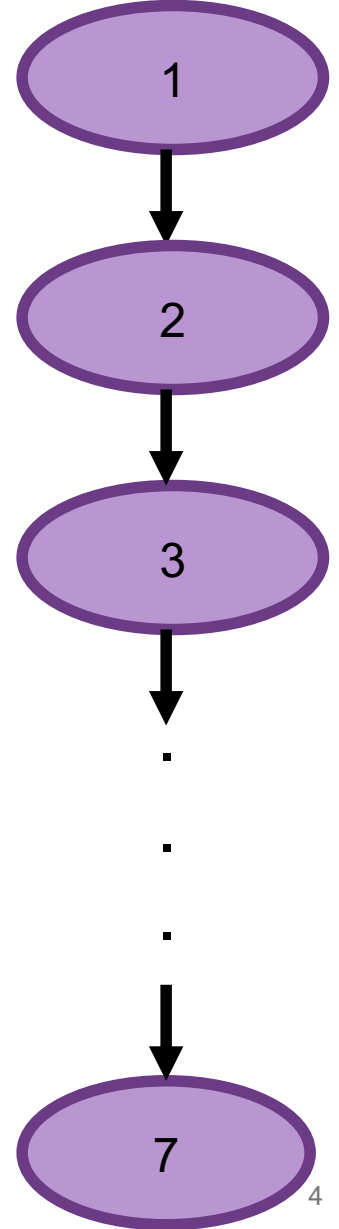
- Concurrent / Parallel Programming
- Overview of Threads
- Thread Implementation Models
- Linux Implementations of POSIX Threads
- The pthread API
- Thread Attributes
- Threads and Signals
- Threads and fork()
- Thread Cancellation

Concurrent / Parallel Programming

Sequential Programming

- Suppose we want to add eight numbers $x_1, x_2, x_3, \dots, x_8$
- There are seven addition operations and if each operation take 1 CPU cycle, the entire operation will take seven cycles

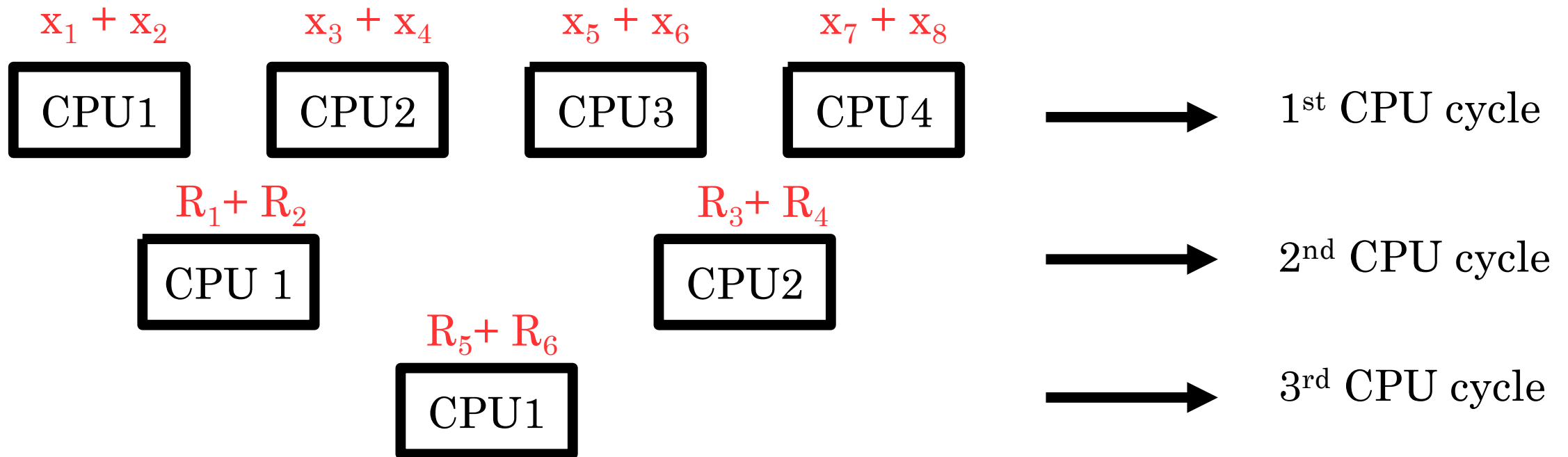
$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$$



Concurrent/Parallel Programming



- Suppose we have 4xCPU's or a 4xCore CPU, the seven addition operations can now be completed in just three CPU cycles, by dividing the task among different CPUs



Ways to Achieve Concurrency



Multiple single threaded processes:

- Use `fork()` to create a new process for handling every new task, the child process serves the client process, while the parent listens to the new request.
- Possible only if each slave can operate in isolation.
- Need IPC between processes.
- Lot of memory and time required for process creation.

Multiple threads within a single process

- Create multiple threads within a single process.
- Good if each slave need to share data.
- Cost of creating threads is low, and no IPC required.

Single process multiple events:

- Use non-blocking or asynchronous I/O, using `select()` and `poll()` system calls.

Overview of Threads

Processes and Threads

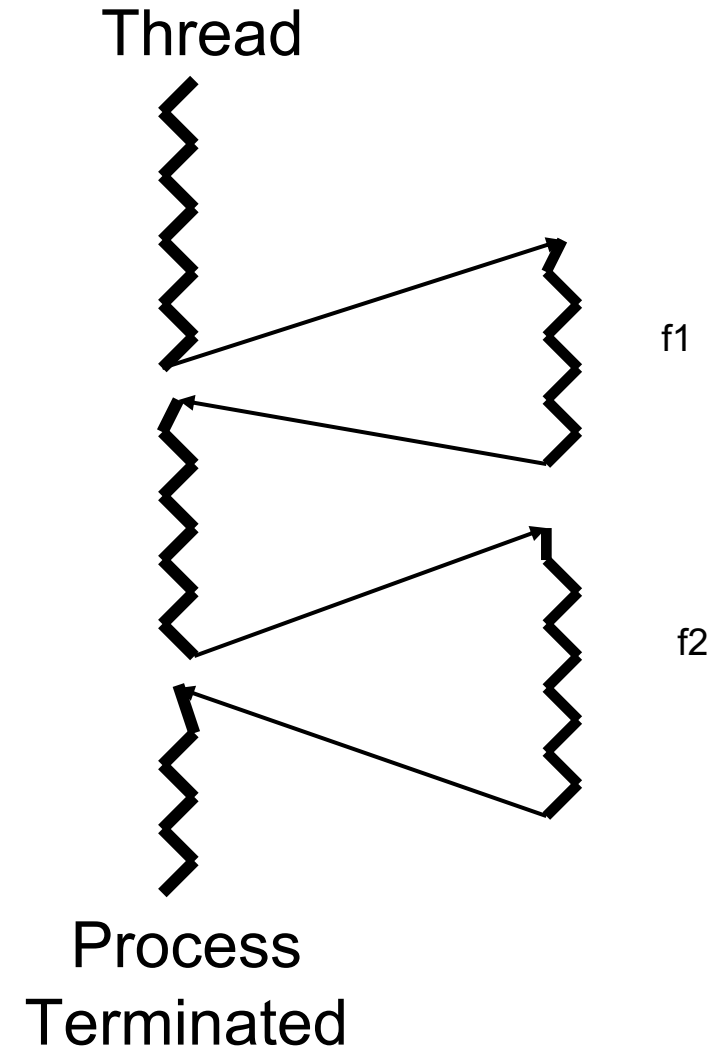


- Every process has two characteristics:
 - **Resource Ownership:** A process owns a set of resources, most importantly a virtual address space that holds the process image (which includes code, data, stack, and heap).
 - **Scheduling and Execution:** A process follows an execution path defined by its code. This execution is managed by the scheduler and may be interleaved with the execution of other processes to enable multitasking.
- These two characteristics are treated independently by the operating system. The unit of *resource ownership is referred to as a process*, while the unit of *dispatching is referred to as a thread*

A thread is an execution context that is independently scheduled, but shares a single addresses space with other threads of the same process

Single Threaded Process

```
main()  
{  
    ...  
    f1 (...);  
    ...  
    f2 (...);  
    ...  
}  
  
f1 (...)  
{ ... }  
  
f2 (...)  
{ ... }
```



Thread Concept



- Previous slide is an example of a process with a single thread. Suppose we want that functions `f1()` and `f2()` should be executed by separate threads, while `main()` function is executed concurrently by another thread.
- **Multi threading refers to the ability of an OS to support multiple threads of execution with in a single process.**
- Multithreading works similar to multiprogramming, where the CPU switches rapidly back and forth among threads providing the illusion that threads are running in parallel.

Multi-Programming Systems (1960s–1970s): Introduced the concept of concurrent execution of multiple programs by making the CPU switch between jobs during I/O idle time. This dramatically increased the overall system utilization. *Example:* IBM OS/360 MVT.

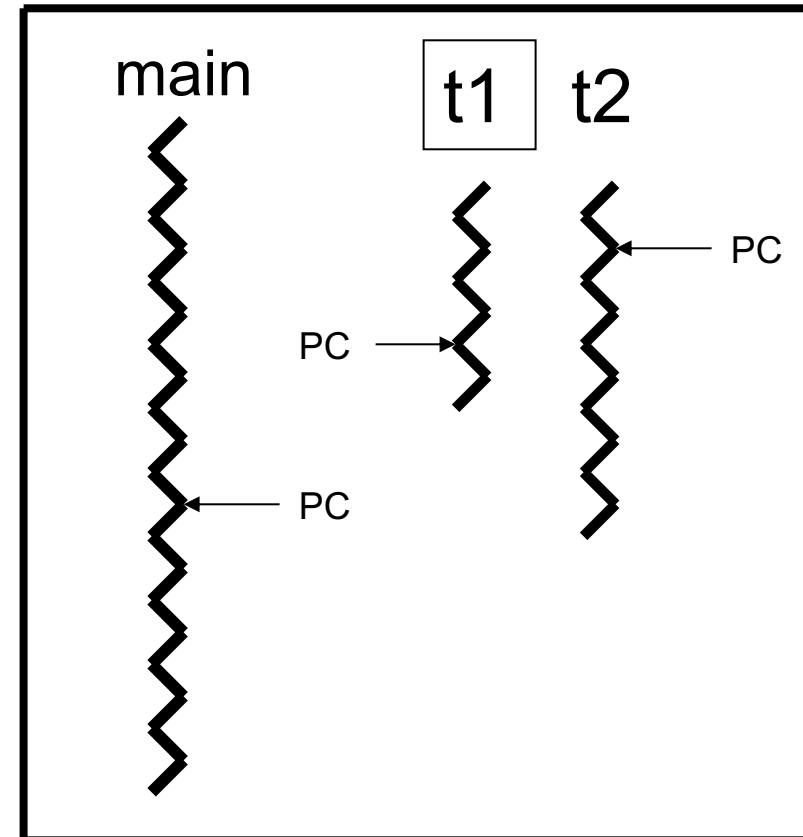
Multi-Tasking / Time-Sharing Systems (1970s): Enabled multiple users to interact with the computer simultaneously via terminals. The OS would rapidly switch between users (time slicing), giving the illusion of concurrent use. *Example:* MULTIX (1969) and UNIX (1970).

Multi-Threading Systems (1990s): Allow different threads of a single process to run concurrently, sharing the same address space and resources but executing independently, leading to finer-grained parallelism and efficiency in modern applications. *Example:* Mach (1980s), Windows NT (1993), Java (1995), LinuxThreads (1996), NPTL (2003)

Multi Threaded Process

```
main()  
{  
    ...  
    thread(t1, f1);  
    ...  
    thread(t2, f2);  
    ...  
}  
  
f1(...)  
{ ... }  
  
f2(...)  
{ ... }
```

Process Address Space



Processes and Threads (Cont...)



Similarities between Processes & Threads:

- Like a process, a thread can also be in one of many states (new, ready, running, block, terminated).
- Only one thread can be in running state (single CPU).
- Like a process a thread can create a child thread.

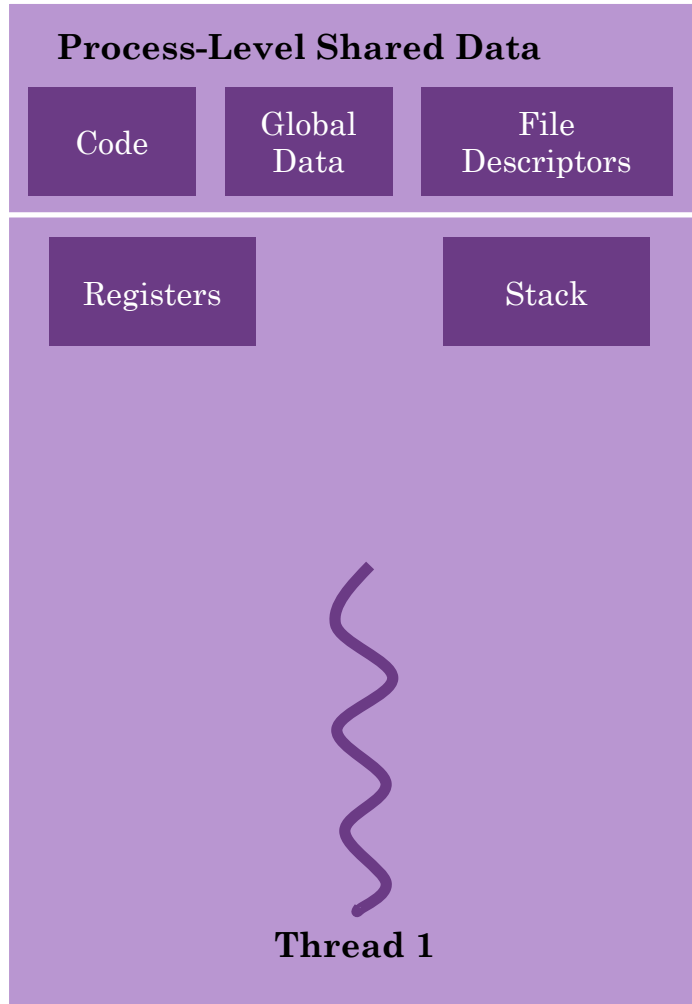
Differences between Processes & Threads:

- Processes are isolated from each other with separate address spaces, while threads share the same address space within a process.
- Threads share code, data, and open files, whereas processes do not share these resources by default.
- No automatic protection in threads.

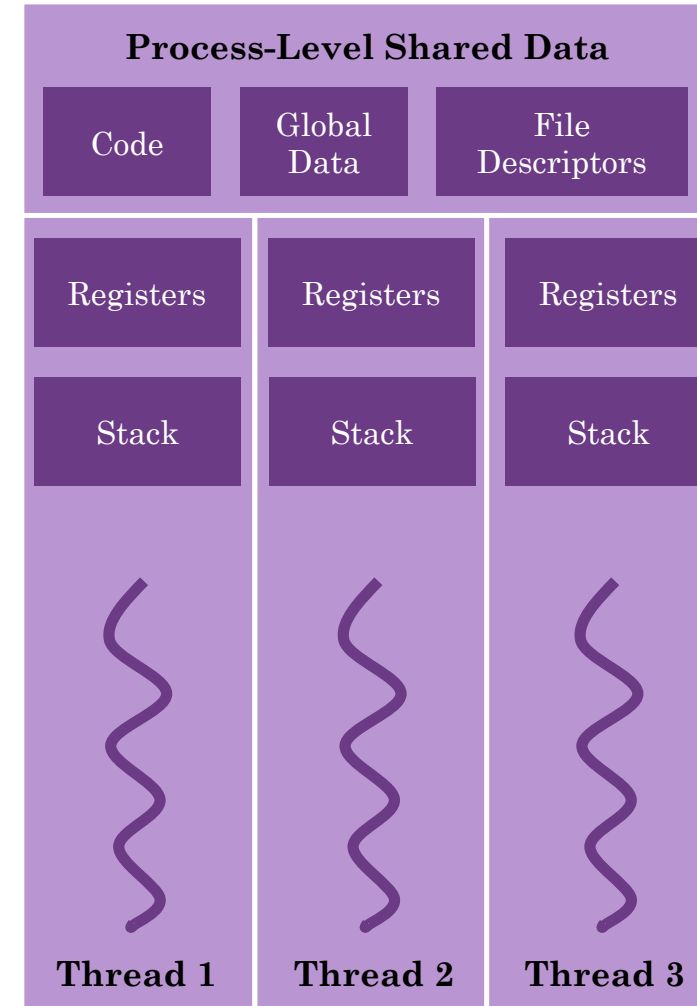
Single vs Multi-threaded Process



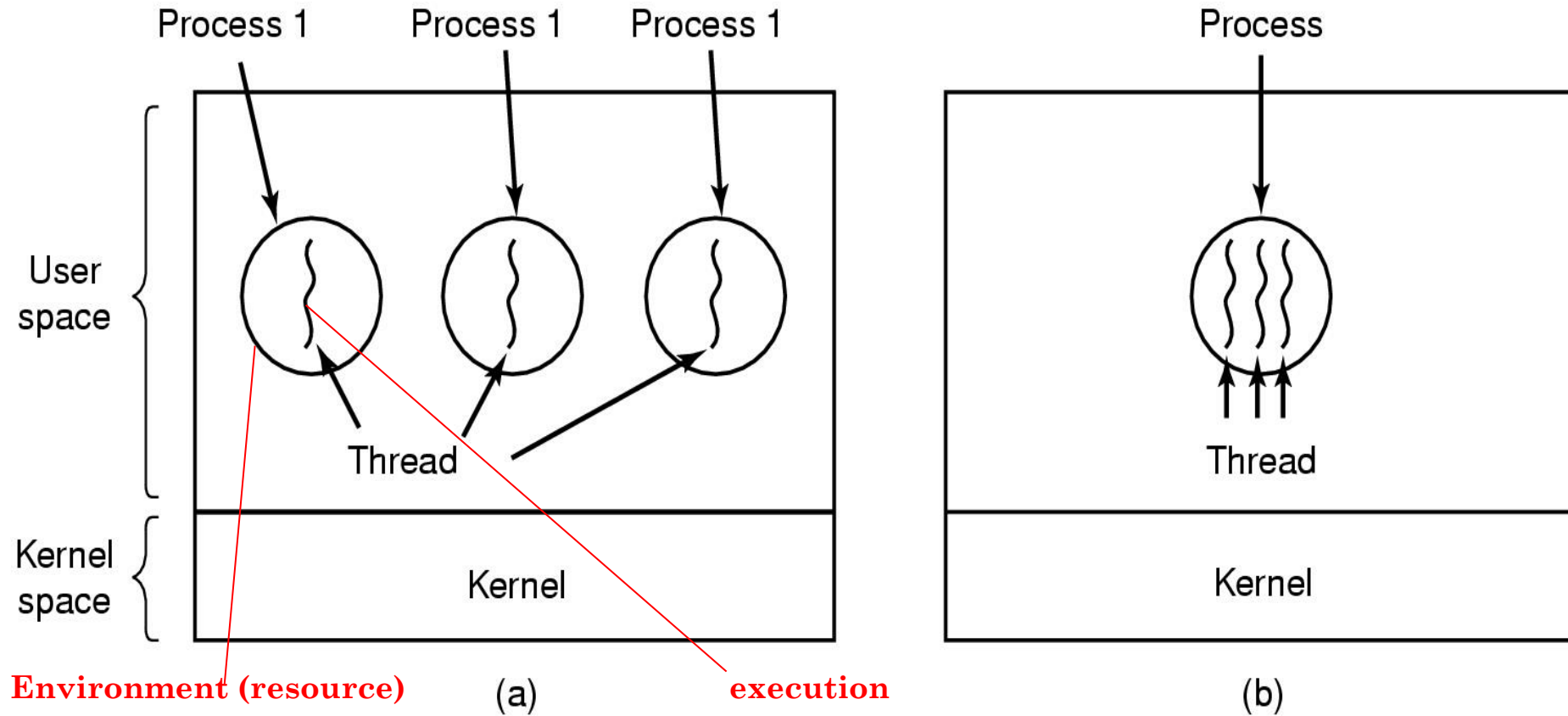
Single-Threaded Process



Multi-Threaded Process



Single vs Multi-threaded Process (cont...)



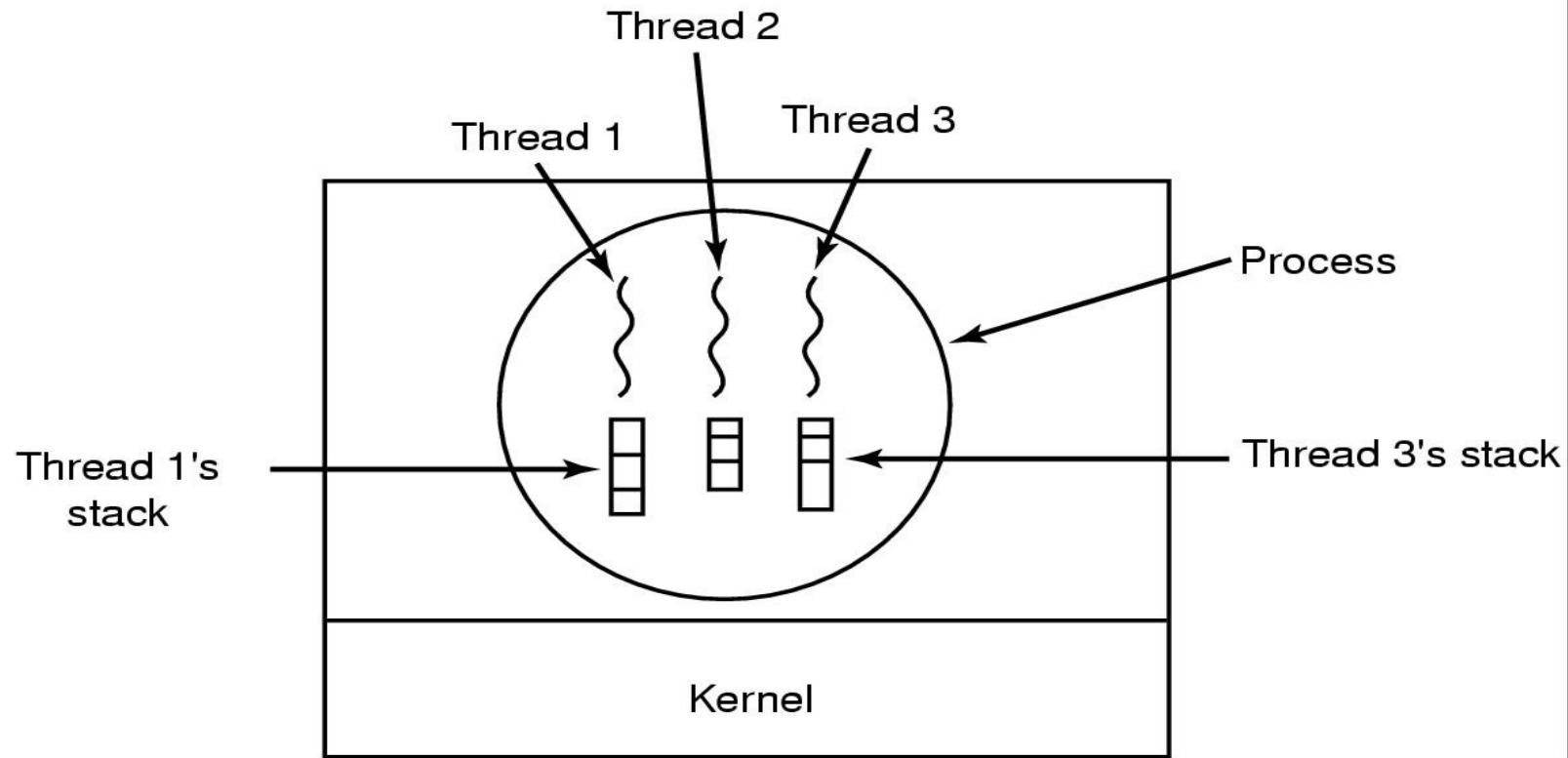
(a) Three processes, each with one thread

(b) One process with three threads

Processes are used to group resources together.

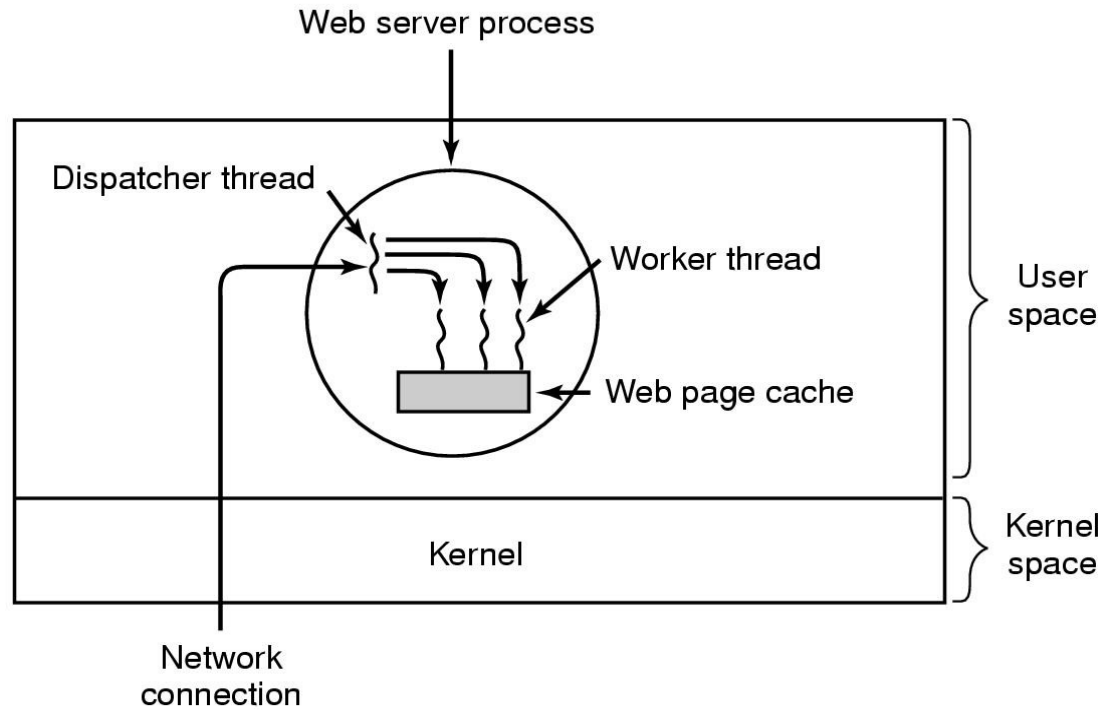
Threads are the entities scheduled for execution on the CPU.

Each Thread has its own Stack



- Each thread is invoked by invoking the thread function, therefore, it has a Function Stack Frame of its own. The FSF contains the procedure's arguments, local variables and return address.
- Suppose, a thread function X, calls another function Y, which in turn calls a function Z, while Z is executing the frames for X, Y, Z will be on the stack. Since each thread will generally call different procedures and thus has a different execution history

Multi-Threaded Web Server



Request for pages comes in and the requested page is sent back to the client.

- Dispatcher thread or main thread reads incoming requests from the NW. After examining the request, it chooses an idle worker thread and hands it the request. It also wakes up the worker from blocked state to ready state.
- Worker now checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a `read()` operation to get the page from the disk and blocks until the disk operation completes. When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

Multi-Threaded Process

Threads within a process share :

- PID, PPID, PGID, SID, UID, GID
- Controlling Terminals
- Code and Data Section
- Global Variables
- Open files via PFDT
- Signal Dispositions
- Umask value
- Current Working Directory
- Interval Timers
- CPU time consumed
- Resource Limits
- Nice value
- Record locks (using `fcntl()`)

Threads have their own:

- Thread ID
- CPU Context (PC, and other registers)
- Stack
- State
- The `errno` variable
- Priority
- CPU affinity
- Signal mask

Temporal vs Simultaneous Multi-threading



At the hardware level there are two main types of multi-threading models:

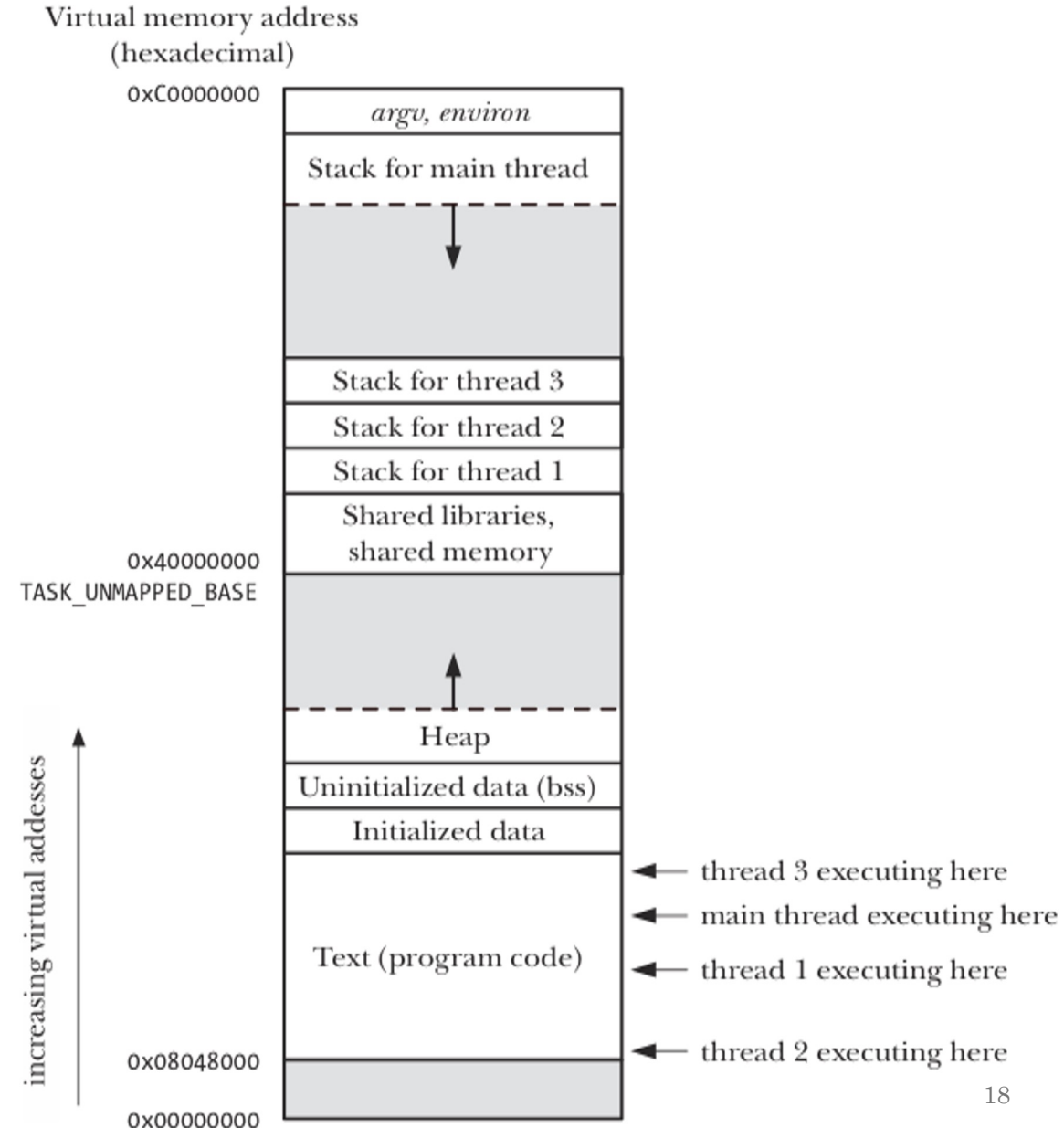
Temporal Multi-threading:

- At any given time, only one thread can execute in a pipeline stage.
- Thread execution is time-sliced, meaning threads take turns using the pipeline.
- Helps utilize CPU resources during stalls (e.g., cache misses).

Simultaneous Multi-threading (SMT/HT):

- Multiple threads can execute simultaneously in different parts of the same pipeline stage.
- Takes advantage of superscalar architecture, where multiple instructions can be issued per cycle.
- Improves instruction-level parallelism and CPU throughput by utilizing idle execution units.

Instructor: Muhammad Arif Butt, PhD

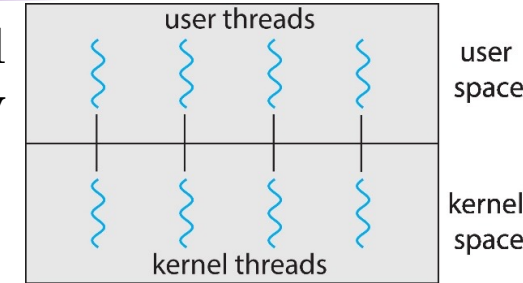


Thread Implementation Models

Thread Implementation Models (1:1)



Kernel-level Threads: In the 1:1 model, each user thread maps directly to a kernel thread (Kernel Scheduling Entity or KSE). All thread operations are carried out by system calls, with the kernel managing thread creation, scheduling, and synchronization.



Advantages:

- Since kernel is aware of the existence of multiple threads within the process, it can schedule these threads independently and run in parallel on multiprocessor systems.
- When one thread makes a blocking system call (e.g., `read()`), only that thread is blocked; other threads can continue to execute.

Disadvantages:

- Thread operations (creation, synchronization, and context switching) are slow as a switch to kernel mode is required.
- Overhead of managing separate KSE for each of the thread places a significant load on kernel scheduler, degrading overall performance.

Usage:

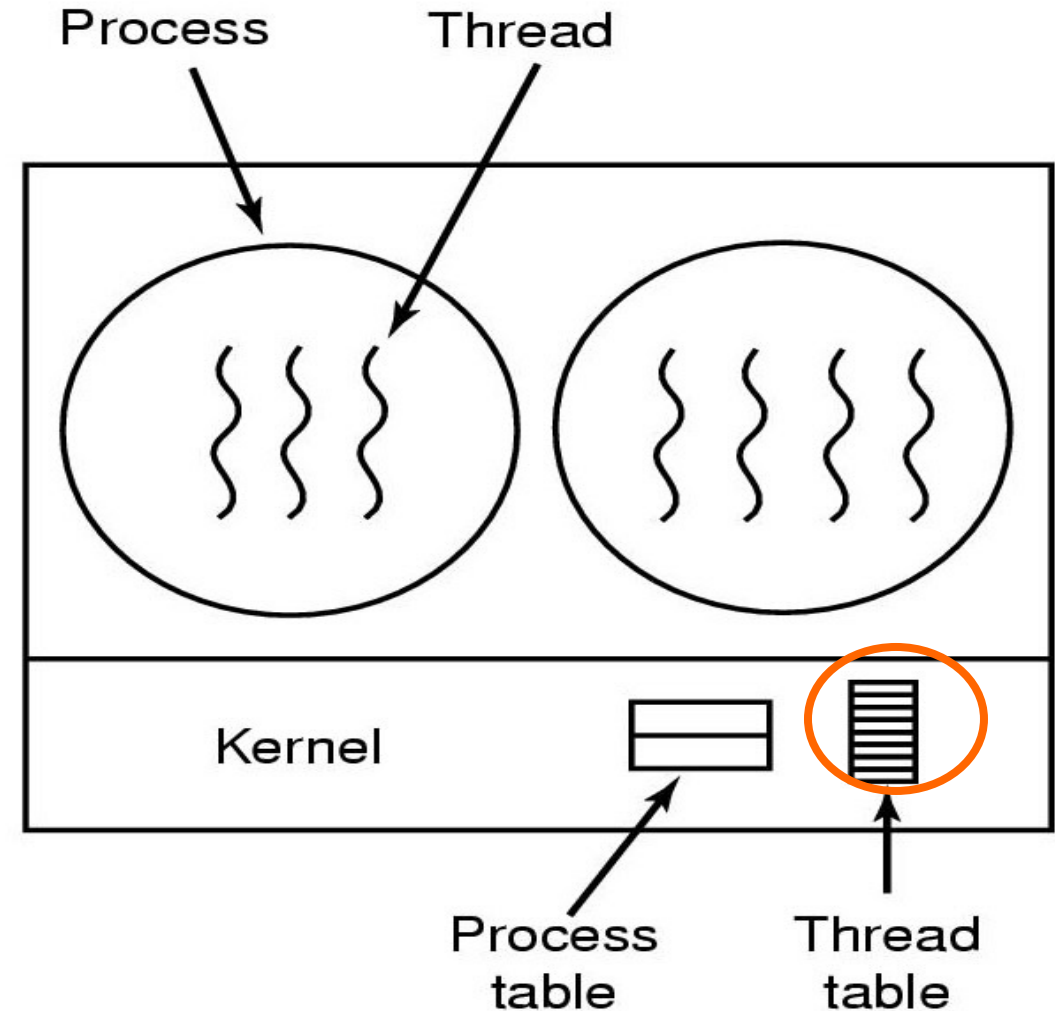
- LinuxThreads (used in older versions of glibc) provided a somewhat limited 1:1 model, but had some issues like non-compliance with POSIX and inconsistent signal handling.
- NPTL (Native POSIX Thread Library), introduced in Linux 2.6, fully implements the 1:1 model and is the standard threading implementation on modern Linux systems. It is fully POSIX-compliant, highly performant, and supports real parallelism on SMP systems.

Thread Implementation Models (1:1)



Kernel-level Threads

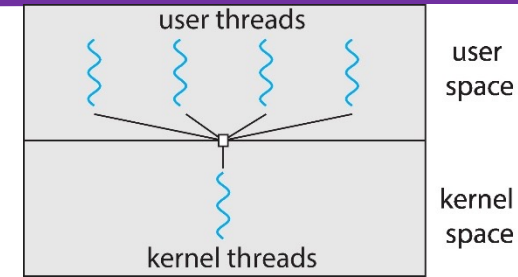
- All the thread management code is inside the kernel space, implemented within a system call. The system call may be called by a library (as in case of NPTL).
- The **Thread table** is maintained by kernel inside the kernel space.
- Kernel knows about individual threads within each process.



Thread Implementation Models (M:1)



User-level Threads: In the many-to-one (M:1) model, multiple user-level threads are mapped to a single kernel thread. All thread management (creation, scheduling, synchronization) occurs in user space. The kernel is unaware of the existence of multiple threads within the process.



Advantages:

- Thread operations (creation, synchronization, and context switching) are fast as no mode switch is required.
- User-level threads can be used even if the underlying platform does not support multi-threading at the kernel level.

Disadvantages:

- When one thread makes a blocking system call (e.g., `read()`), the entire process is blocked.
- Since the kernel is unaware of the existence of multiple threads within a process, it cannot schedule separate threads to different CPU cores.

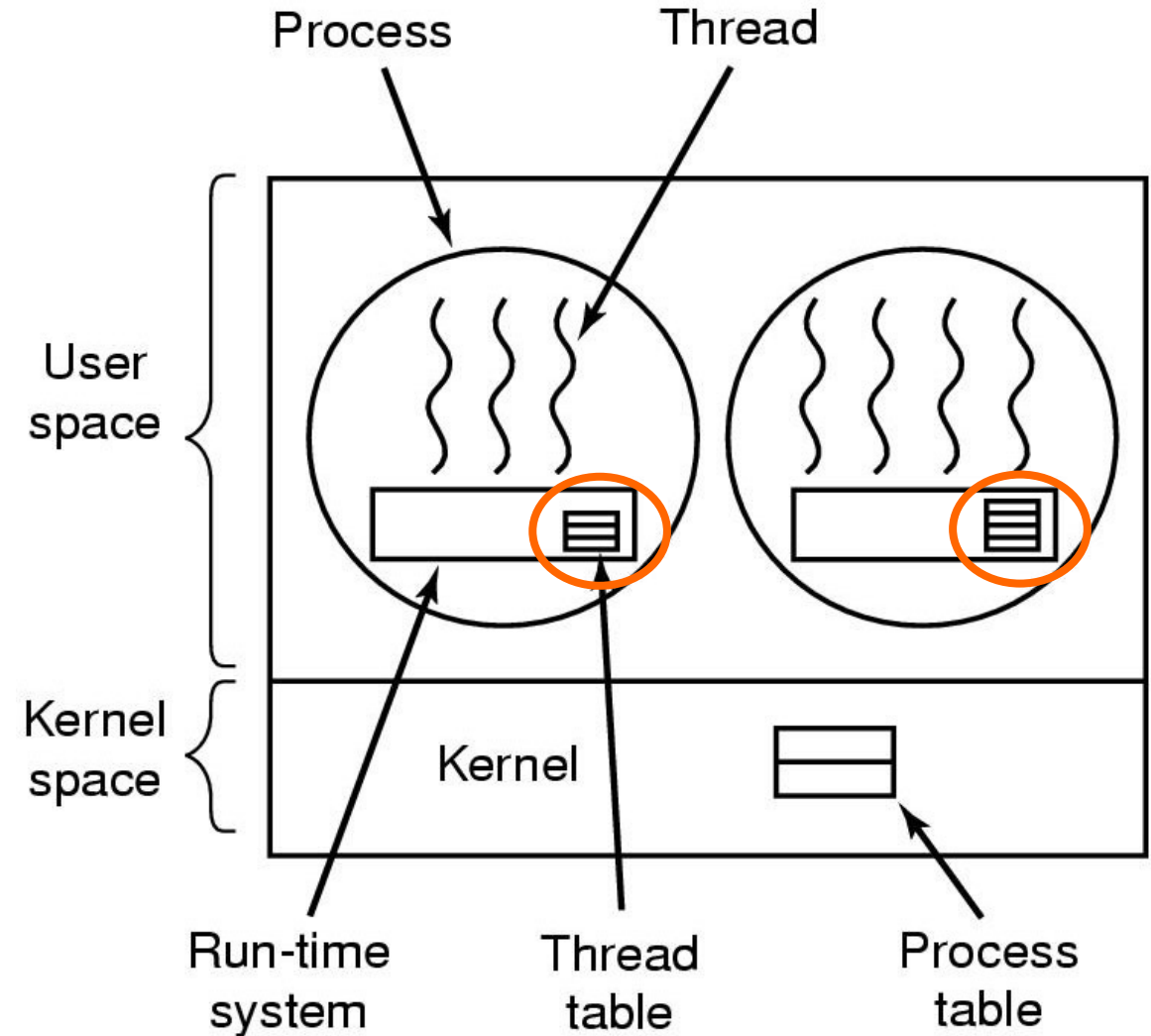
Usage: The M:1 model was used by some early threading libraries (e.g., GNU Portable Threads or Solaris green threads in early Java implementations).

Thread Implementation Models (M:1)



User-level Threads

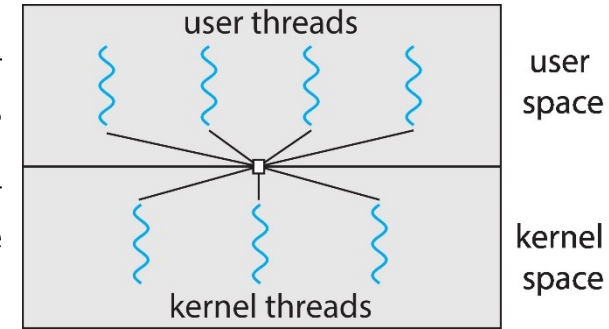
- Mostly implemented by a thread library, where all the thread management code is inside the user space.
- The **Thread table** is maintained by the library and exist inside the user space.
- Kernel knows nothing about threads.



Thread Implementation Models (M:N)



In the many-to-many (M:N) model, multiple user-level threads are mapped to a smaller or equal number of kernel threads (KSEs). The mapping is dynamic, and multiple user threads can be multiplexed over fewer kernel threads. Both the kernel and user-space threading library share responsibilities for scheduling.



Advantages:

- **Combines the benefits of M:1 and 1:1** – Allows high-performance user-level thread management while still enabling true parallel execution and isolation of blocking operations.
- **Resource efficiency** – Reduces kernel-level thread overhead by reusing a smaller number of kernel threads for many user threads.

Disadvantages:

- **Complexity:** Requires sophisticated cooperation between the user-level threading library and the kernel scheduler. The scheduling logic is split across layers, which makes implementation and debugging difficult.
- **Inconsistency:** Poor cooperation between user-space and kernel schedulers can lead to suboptimal thread performance or starvation.

Usage: Not used in mainstream Linux implementations. Some experimental systems implemented M:N models (e.g., older versions of GNU Hurd or Windows UMS)

Linux Implementation of POSIX Threads



LinuxThreads (Original Linux pthreads implementation)

Overview:

- LinuxThreads was the initial implementation of POSIX threads for Linux; it's now deprecated (unsupported since `glibc 2.4`).
- In addition to application-created threads, it spawns a special “manager” thread responsible for thread creation and termination, which can be a single point of failure if killed.
- Internally it relies on `clone()`, but threads appear as separate processes.
- LinuxThreads is a 1:1 model with significant deviations from POSIX behaviour, resulting in unpredictable and inconsistent semantics—this is one of the main reasons it was superseded.

POSIX Deviations & Behavioral Quirks:

- The `getpid()` call returns a unique PID per thread, instead of a common process ID.
- Only the thread that invoked `fork()` can `wait()` on the child—others can't.
- On `execve()`, all other threads are terminated, but the resulting PID corresponds to the calling thread, not the main thread.
- Threads do not share: User/group IDs, SID and PGID, File locks, Interval timers, semaphore undo values, nice values etc

NPTL (Native POSIX Thread Library)



Overview:

- NPTL is a POSIX-compliant user-space threading library that was introduced in `glibc 2.3.2` in 2003.
- It employs a 1:1 threading model, so even though `pthread_create()` is a user space function, but the thread it creates is a Kernel Scheduling Entity (KSE)
- Every `pthread_create()` library function internally make the `clone()` system call with flags that cause the new thread to:
 - Share memory space and resources (like file descriptors) with its parent.
 - Be scheduled independently by the kernel.
 - Appear as a separate thread in tools like `top`, `htop`, or `/proc`.

Linux Implementation:

- To check out which Thread implementation your Linux system support, run the following command:

```
$ getconf GNU_LIBPTHREAD_VERSION
```

```
NPTL 2.41
```

pthread API

pthread API



The pthread API defines a number of data types and should be used to ensure the portability of programs and mostly defined in `/usr/include/x86_64-linux-gnu/bits/pthreadtypes.h`. Remember you should not use the C `==` operator to compare variables of these types

Data Type	Description
<code>pthread_t</code>	Used to identify a thread
<code>pthread_attr_t</code>	Used to identify a thread attributes object
<code>pthread_mutex_t</code>	Used for mutex
<code>pthread_mutexattr_t</code>	Used to identify mutex attributes object
<code>pthread_cond_t</code>	Used for condition variable
<code>pthread_cond_attr_t</code>	Used to identify condition variable attributes object
<code>pthread_key_t</code>	Key for thread specific data
<code>pthread_once_t</code>	One-time initialization control context
<code>pthread_spinlock_t</code>	Used to identify spinlock
<code>pthread_rwlock_t</code>	Used for read-write lock
<code>pthread_rwlockattr_t</code>	Used for read-write lock attributes
<code>pthread_barrier_t</code>	Used to identify a barrier
<code>pthread_barrierattr_t</code>	Used to identify a barrier attributes object

pthread API (cont...)



```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,  
                  void *(*start)(void *), void *arg) ;
```

- This function starts a new thread in the calling process. The new thread starts its execution by invoking the start function which is the 3rd argument to above function.
- On success, the TID of the new thread is returned through 1st argument to above function
- The 2nd argument specifies the attributes of the newly created thread. Normally we pass NULL pointer for default attributes.
- The 4th argument is a pointer of type void which points to the value to be passed to thread start function. It can be NULL if you do not want to pass any thing to the thread function. It can also be address of a structure if you want to pass multiple arguments.

pthread API (Cont...)



```
void pthread_exit(void *status);
```

- This function terminate the calling thread.
- The status value is returned to some other thread in the calling process, which is blocked on the `pthread_join()` call.
- The pointer status must not point to an object that is local to the calling thread, since that object disappears when the thread terminates.

Ways for a thread to terminate:

- The thread function calls the `return` statement.
- The thread function calls `pthread_exit()`
- The main thread `returns` or call `exit()`
- Any sibling thread calls `exit()`

pthread API (Cont...)



```
int pthread_join(pthread_t tid, void **retval);
```

- Any peer thread can wait for another thread to terminate by calling `pthread_join()` function, similar to `waitpid()`. Failing to do so will produce the thread equivalent of a zombie process.
- The 1st argument is the ID of thread for which the calling thread wish to wait. Unfortunately, we have no way to wait for any of our threads like `wait()`
- The 2nd argument can be `NULL`, if some peer thread is not interested in the return value of the new thread. Otherwise, it can be a double pointer which will point to the status argument of the `pthread_exit()`

Example: t0.c



- Consider the following code and determine its output and behavior:

```
void f1();
void f2();
int main(){
    f1();
    f2();
    printf("\nBye Bye from main\n");
    return 0;
}
```

```
void f1(){
    for(int i=0; i<5; i++){
        printf("%s", "PUCIT");
        sleep(1);
    }
}
```

```
void f1(){
    for(int i=0; i<5; i++){
        printf("%s", "ARIF");
        sleep(1);
    }
}
```

Example: t1.c



```
void* f1(void*);
void* f2(void*);
int main(){
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, f1, NULL);
    pthread_create(&tid2, NULL, f2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("\nBye Bye from main thread\n");
    return 0;
}
```

```
void * f1(void * arg){
    for(int i=0; i<5; i++){
        printf("%s", "PUCIT");
        fflush(stdout);
        sleep(1);
    }
    pthread_exit(NULL);
}
```

```
void * f2(void * arg){
    for(int i=0; i<5; i++){
        printf("%s", "ARIF");
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}
```

```
$ gcc -c t1.c -D_REENTRANT
```

```
$ gcc t1.o -o t1 -lpthread
```


```
$/t1
```

```
PUCITARIFARIFPUCITARIFPUCITPUCITARIFPUCITARIF
```

```
Bye Bye from main thread
```

Demonstration

Thread Creation



```
Lec3.3/t0.c  
Lec3.3/t1.c  
Lec3.3/t2.c  
Lec3.3/t3.c  
Lec3.3/t4.c
```

GitHub Code Repository Link: <https://github.com/arifpucit/OS-Codes>

Returning value from a Thread Function



- A thread function can return a pointer to its parent/calling thread, and that can be received in the 2nd argument of the `pthread_join()` function.
- The pointer returned by the `pthread_exit()` must not point to an object that is local to the thread, since that variable is created in the local stack of the terminating thread function.
- Making the local variable static will also fail. Suppose two threads run the same `thread_function()`, the second thread may overwrite the static variable with its own return value and return value written by the first thread will be overwritten.
- So the best solution is to create the variable to be returned in the **heap** instead of *stack*.

```
pthread_create(&tid1, NULL, f1, (void*)argv[1]);
pthread_create(&tid2, NULL, f1, (void*)argv[2]);
pthread_join(tid1, &rv1);
pthread_join(tid2, &rv2);
int count1 = *((int*)rv1);
int count2 = *((int*)rv2);
printf("Characters in %s: %d\n", argv[1], count1);
printf("Characters in %s: %d\n", argv[2], count2);
return 0;
}

void* f1(void* args){
    char* filename = (char*)args;
    int *result = (int*)malloc(sizeof(int));
    *result = 0;
    char ch;
    int fd = open(filename, O_RDONLY);
    while((read(fd, &ch, 1)) != 0){
        (*result)++;
    }
    close(fd);
    pthread_exit((void*)result);
}
```

Demonstration



GitHub Code Repository Link: <https://github.com/arifpucit/OS-Codes>

Creating Arrays of Threads



- You may need to create large number of threads for dividing the computational tasks as per your program logic.
- At compile time, if you know the number of threads you need, you can simply create an array of type `pthread_t` to store the thread IDs.
- If you do not know at compile time, the number of threads you need, you may have to allocate memory on heap for storing the thread IDs.
- The maximum number of threads that a system allow can be seen in `/proc/sys/kernel/threads-max` file. There are however, other parameters that limit this count like the size of stack the system needs to give to every new thread.

Demonstration

Array of Threads

Lec3.3/array_threads1.c
Lec3.3/array_threads2.c
Lec3.3/array_threads3.c

GitHub Code Repository Link: <https://github.com/arifpucit/OS-Codes>

Point to Ponder



errno is a global per-process variable used to store the error number occurred in the last failed system call. What problem can occur due to this shared variable in a multi-threaded program?



- The problem is that two or more threads can encounter errors, all causing the same `errno` variable to be set. Under these circumstances, a thread might end up checking `errno` after it has already been updated by another thread.
- Solution is to make `errno` local to every thread; so setting it in one thread does not affect its value in any other thread. This can be achieved by compiling with `-D_REENTRANT` flag of `gcc`.

Point to Ponder



In a multithreaded process, all threads have the same PID as returned by the `getpid()` system call. How to uniquely identify a thread within a multi-threaded process?



- We can use `gettid()` and `pthread_self()`.
- But must keep in mind the following difference between the values returned by these two calls

TID returned by <code>gettid()</code>	TID returned by <code>pthread_self()</code>
Assigned by kernel, similar to PIDs	POSIX TIDs maintained by thread implementation
May be reused after a very long time once the PID counter reach the max value	Reused after the completion of the thread
Unique across the system	Unique within the process only

- Since `gettid()` is Linux specific and therefore not portable. So to uniquely identify a thread, use combination of process ID as returned by `getpid()` and POSIX thread ID as returned by `pthread_self()`

Demonstration

Thread IDs

Lec3.3/id_threads1.c
Lec3.3/id_threads2.c

GitHub Code Repository Link: <https://github.com/arifpucit/OS-Codes>

Thread Attributes

Thread Attributes



Every thread has a set of attributes which can be set before creating it. If we pass a NULL as second argument to `pthread_create()` function, the default thread attributes are used. The default value of thread attributes are shown in table below:

Attribute	Default Value	Description
detachstate	PTHREAD_CREATE_JOINABLE	Joinable by other threads
stackaddr	NULL	Stack allocated by system
stacksize	NULL	2 MB
priority	---	Priority of calling thread is used
policy	SCHED_OTHER	Determined by system
inheritsched	PTHREAD_INHERIT_SCHED	Inherit scheduling attributes from creating thread

Detach State (Avoiding Zombie Threads)



Joinable Thread:

- A joinable thread (like a process) is not automatically cleaned up by GNU/LINUX when it terminates. The thread's exit status hangs around in system until another thread calls `pthread_join()` to obtain its return value. Only then its resources are released.
- For example whenever we want to return data from child thread to parent thread the child thread must be a joinable thread.

Detached Thread:

- A detachable thread is cleaned up automatically when it terminates. Since a detached thread is immediately cleaned up, another thread may not wait for its completion by using `pthread_join()` to obtain its return value.
- For example suppose the main thread creates a child thread to do back up of a file and the main thread continue its execution. When the backup is finished , the second thread can just terminate.
- There is no need for it to rejoin the main thread. A thread can detach itself using `pthread_detach(pthread_self())` call

Steps to Specify Customized Thread Attributes

- Create a `pthread_attr_t` object.
- Call `pthread_attr_init()`, passing it a pointer of above object.
- Modify the attribute object to contain the desired attribute value using the appropriate setters.
- Pass a pointer to the attribute object when calling `pthread_create()`
- Destroy pthread attribute object by calling `pthread_attr_destroy()`

pthread API (Cont...)



```
int pthread_attr_init(pthread_attr_t *attr);  
int pthread_attr_destroy(pthread_attr_t *attr);
```

- The `pthread_attr_init()` function initializes the thread attributes object pointed to by `attr` with default attribute values. After this call, individual attributes of the object can be set using various related functions (next slide), and then the object can be used in one or more `pthread_create()` calls.
- When a thread attributes object is no longer required, it should be destroyed using the `pthread_attr_destroy()` function. Destroying a thread attributes object has no effect on threads that were created using that object.

pthread API (Cont...)



```
int pthread_attr_setdetachstate( pthread_attr_t*attr,int detachstate);
```

- This function sets the detach state attribute of the thread attributes object referred to by attr to the value specified in the second argument detachstate, which can take following two values:
 - PTHREAD_CREATE_DETACHED
 - PTHREAD_CREATE_JOINABLE
- Associated getters and setters of thread attribute object

```
int pthread_attr_getdetachstate();  
int pthread_attr_setdetachstate();  
int pthread_attr_getstacksize();  
int pthread_attr_setstacksize();  
int pthread_attr_getstackaddr();  
int pthread_attr_setstackaddr();
```

```
int pthread_attr_getschedpolicy();  
int pthread_attr_setschedpolicy();  
int pthread_attr_getinheritsched();  
int pthread_attr_setinheritsched();
```

Demonstration



GitHub Code Repository Link: <https://github.com/arifpucit/OS-Codes>

Point to Ponder



If a signal is sent to a multi-threaded process. Which thread will receive that signal?



The UNIX signal model was designed with the UNIX process model in mind, so there are some significant conflicts between the signal and thread models. Combining signals and threads is complex and should be avoided whenever possible. Some key points to be kept in mind are:

- Signal handlers are per-process.
- Signal masks are per-thread.
- Sending a signal using `kill(1)` or `kill(2)` will terminate the process. You can use `pthread_kill(3)` to send a signal to another thread in the same process.
- If one thread ignores a signal, then that signal is ignored by all threads

Point to Ponder



If one of the threads executes the `exec()` system call, what happens?



- If any thread in a multithreaded Linux process executes one of the `exec()` functions, the entire process is completely replaced by the new program image.
- All the other threads are terminated, and only the calling thread continues as single thread of the new program.
- None of the threads executes destructors for thread-specific data or calls cleanup handlers.
- All the pthread objects (mutexes and condition variables) disappear as the new program overwrites the memory of the process.

Point to Ponder



Consider a multi-threaded process, if one thread executes the `fork()` system call, does the new process duplicate only the calling thread or all threads? Is the child process single threaded or multi-threaded?

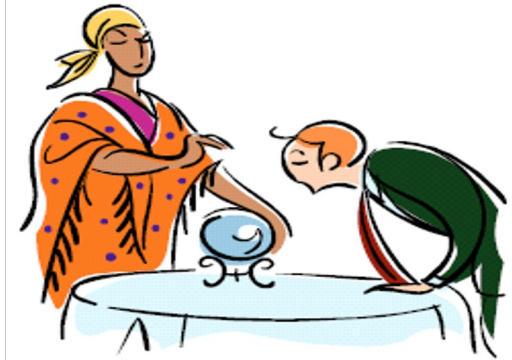


- When a thread in a multi-threaded process calls `fork()`, the operating system creates a child process that contains only the calling thread. The child process is created with a single thread – the one that called the `fork()`; none of the other threads are duplicated.
- This means the child process is single-threaded, even though the parent process may still be multi-threaded (all its other threads continue running normally).

Point to Ponder



What if the main thread want to cancel another thread or threads? Suppose multiple threads are searching through a database, if one thread returns data, remaining threads might need to be cancelled.



- A thread can call `pthread_cancel()` to request that another thread be cancelled by mentioning the TID of the target thread
- This cancellation may cause a problem if the target thread is holding some resources which it must free later
- To counter this possibility, it is possible for a thread to make itself cancellable or non-cancellable by calling a function `pthread_setcancelstate()`
- Moreover, a cancellable thread may also set its cancel type by calling a function `pthread_setcanceltype()`, which can be asynchronous, i.e., thread may be cancelled at any point in its execution or deferred, in which case the cancellation request is queued, until the target thread reaches next cancellation point.

Point to Ponder



Why all multi-threaded code must be compiled with **-D_REENTRANT** defined?
What difference does it make?



Compiling your multi-threaded code with **-D_REENTRANT** enables thread-safe behavior by affecting the standard headers and libraries in the following ways:

- Every thread has its local `errno` variable that refers to a thread-specific location, ensuring that error values are not overwritten by other threads.
- Library functions like `getc()` and `putc()` are redefined as real function calls instead of macros. This change allows the use of internal locking mechanisms necessary for thread-safe access to shared resources (e.g., `FILE` streams).
- The code becomes compatible with reentrant versions of library functions, such as `gethostbyname_r()` instead of `gethostbyname()`. These reentrant functions are explicitly designed to be safe in multi-threaded contexts, often by requiring the caller to supply buffers.

Example: race1.c



```
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, inc, NULL);
    pthread_create(&t2, NULL, dec, NULL);
    pthread_join(t1, NULL);    pthread_join(t2, NULL);
    printf("Value of balance is :%ld\n", balance);
    return 0;
}
```

```
void * inc(void * arg){
    for(long i=0;i<1000000000;i++)
        balance++;
    pthread_exit(NULL);
}
```

```
void * dec(void * arg){
    for(long i=0;i<1000000000;i++)
        balance--;
    pthread_exit(NULL);
}
```


- Watch SP video on Multi-Threaded Programming:
<https://youtu.be/OgnLaXwLC8Y?si=GeXtVLkLCTbCHgeI>



Coming to office hours does NOT mean that you are academically weak!