Video Lecture # 23
Multi-threaded Programming

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

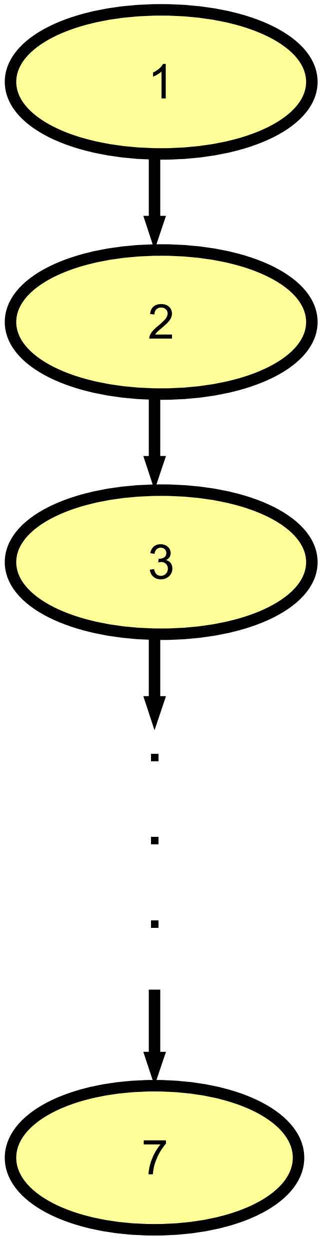
Source Code files available at: **`https://bitbucket.org/arifpucit/spvl-repo/src`**
Lecture Slides available at: **`http://arifbutt.me`**

Agenda

- Concurrent / Parallel Programming
- Overview of Threads
- Thread Implementation Models
- Linux Implementations of POSIX Threads
  - LinuxThreads
  - NPTL
- Creating and managing threads using pthread API (NPTL)
- Thread Attributes
- Threads and Signals
- Threads and `fork()`
- Thread Cancellation

Concurrent / Parallel Programming

Sequential Programming



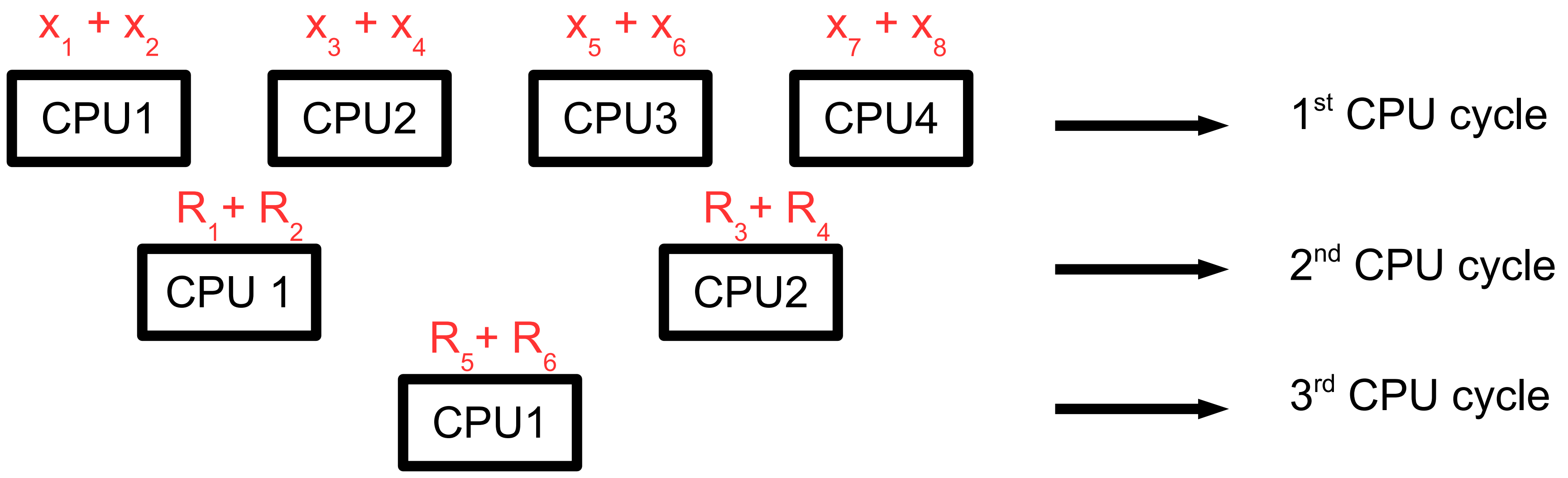
Suppose we want to add eight numbers x_1, x_2, x_3, ... x_8

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 4xCPUs 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**- process includes a virtual address space to hold the process image
- **Scheduling**- follows an execution path that may be interleaved with other processes

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

Processes and Threads (cont...)

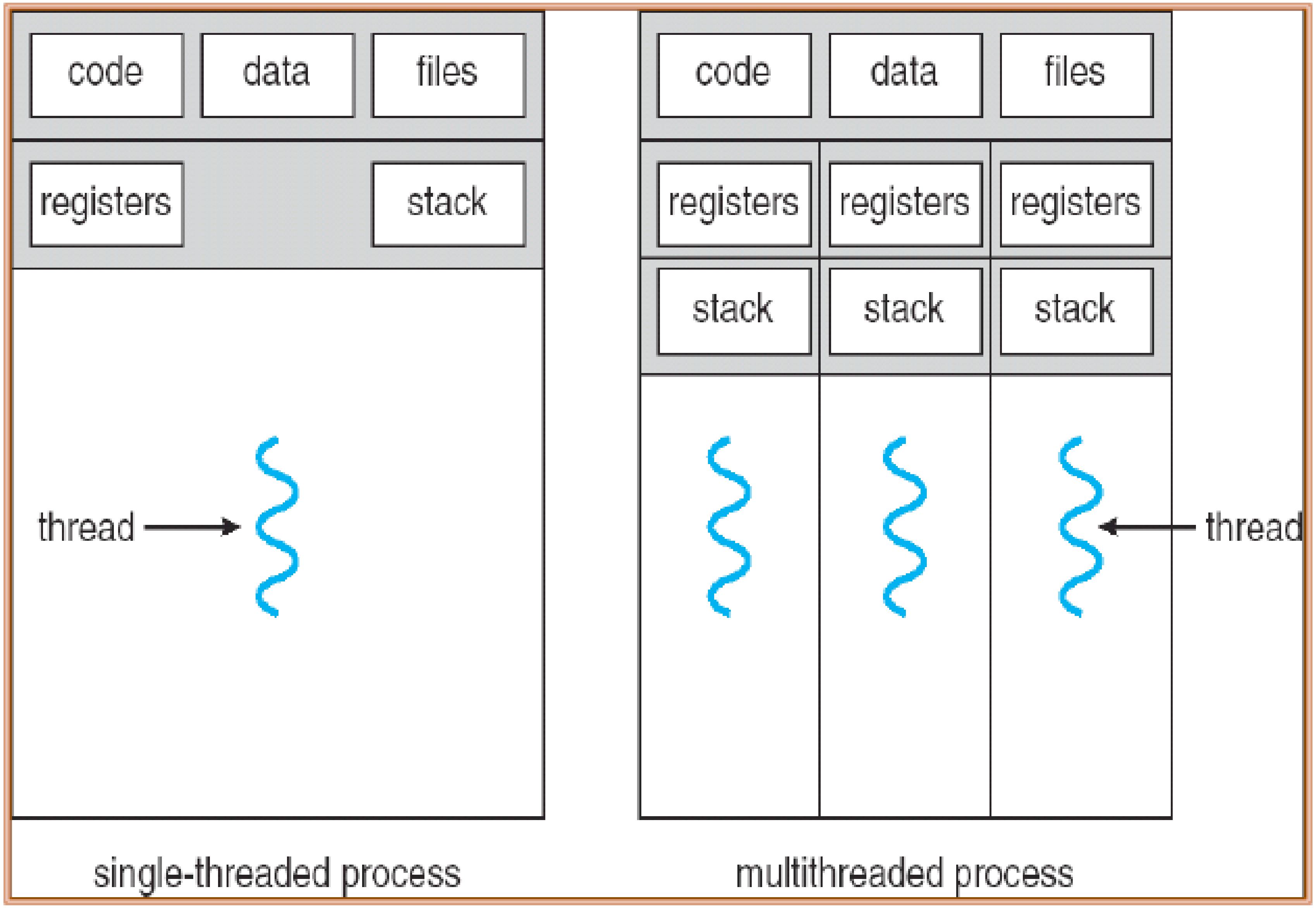
Similarities between Processes & Threads:

- 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

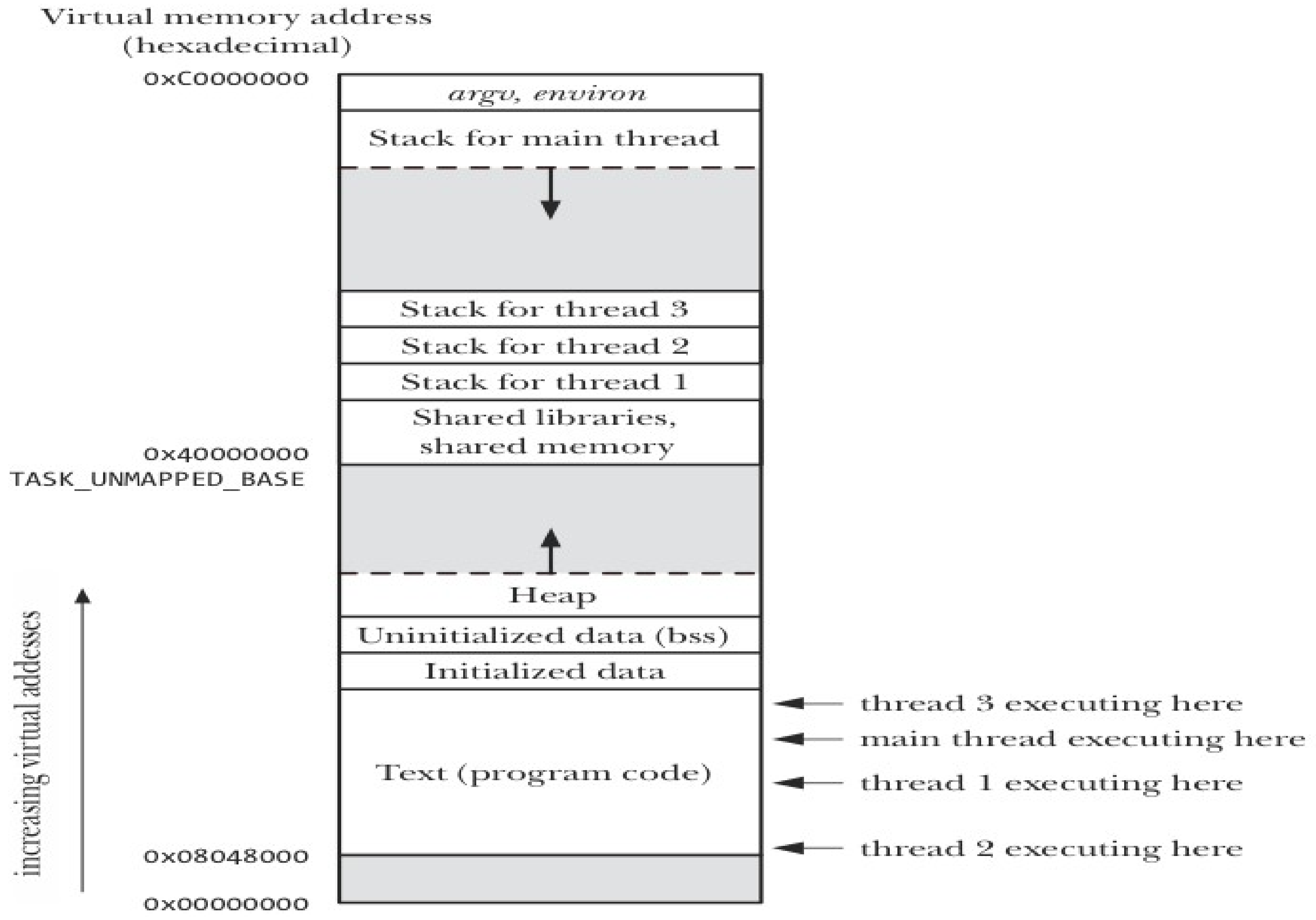
Differences between Processes & Threads:

- No automatic protection in threads
- Every process has its own address space, while all other threads within a process executes within the same address space

Single vs Multi-threaded Process



Pictorial View of a Multi-threaded Process



Temporal Multi-threading: Only one thread of instruction can execute in any given pipeline stage at a time
Simultaneous Multi-threading (SMT/HT): More than one thread of instruction can execute in any giver pipeline stage at a time. (SMT/HT is a multi-threading on a super scalar architecture)

Multi-Threaded Process

Threads within a process share :

- PID, PPID, PGID, SID, UID, GID
- Code and Data Section
- Global Variables
- Open files via PPFDT
- Signal Handlers
- Interval Timers
- CPU time consumed
- Resources Consumed
- Nice value
- Record locks (created using fcntl())

Threads have their own:

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

Thread Implementation Models

Thread Implementation Models (M:1)

In Many-to-one (M:1) threading implementation, all of the details of thread creation, termination, scheduling, synchronization, and so on are handled entirely within the user-space. Kernel knows nothing about the existence of multiple threads within the process

Advantages:

- Thread operations are fast as no mode switch is required
- User level threads can be used even if the underlying platform does not support multithreading

Disadvantages:

- When a user-level 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 the process, it CANNOT schedule separate threads to different CPUs on multiprocessor hardware

Thread Implementation Models (1:1)

In one-to-one (1:1) threading implementation, each thread maps onto a separate kernel scheduling entity (KSE). All of the details of thread creation, termination, scheduling, synchronization and so on are handled by system calls inside the kernel

Advantages:

- When a kernel-level thread makes a blocking system call, e.g., `read()`, only that thread is blocked
- Since the kernel is aware of the existence of multiple threads within the process, it can schedule separate threads to different CPUs on multiprocessor hardware

Disadvantages:

- Thread operations are slow as a switch into kernel mode is required
- Overhead of maintaining a separate KSE for each of the threads in an application place a significant load on the kernel scheduler, degrading overall system performance

Thread Implementation Models (M:N)

The many-to-many (M:N) threading implementation, aim to combine the advantages of the 1:1 and M:1 models, while eliminating their disadvantages. Each process can have multiple associated KSEs, and several threads may map to each KSE

Disadvantages:

- The major disadvantage of M:N model is its complexity. The task of thread scheduling is shared between the kernel and the user-space threading library, which must cooperate and communicate information with one another

The M:N model was initially considered for the NPTL threading implementation, but rejected as it required much changes to the Kernel. The Linux threading implementations **LinuxThreads** and **NPTL** employ the 1:1 model

Linux Implementation of POSIX Threads

LinuxThreads

LinuxThreads is the original Linux threading implementation, developed by Xavier Leroy. In addition to the threads created by the application, LinuxThreads creates an additional "manager" thread that handles thread creation and termination. Threads are created using a **`clone()`**, with the flags mentioned below: (threads share virtual memory, file descriptors, file system-related information (umask, root directory, pwd,...) and signal disposition)

`CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND`

Deviations from specified behavior

- **`getpid()`** returns a different value in each of the threads of a process
- **`getppid()`** returns the PID of the manager thread
- If one thread creates a child using **`fork()`**, then only the thread that created the child process can **`wait()`** for it
- If a thread calls **`exec()`**, then SUSv3 requires that all other threads are terminated. While this is not so in LinuxThreads
- Threads don't share PGIDs, and SIDs
- Threads don't share resource limits
- Some versions of `ps`(1) show all of the threads in a process (including the manager thread) as separate items with distinct PIDs
- CPU time returned by **`times()`** and resource usage information returned by `getrusage()` are per thread
- Threads don't share nice value set by **`setpriority()`**

NPTL Threads

The Native POSIX Threads Library (NPTL) is is the modern Linux Threading implementation, developed by Drepper and Ingo Molnar, designed to address most of the shortcomings of LinuxThreads. It adheres more closely to SUSv3 specification. Applications that employ large number of threads scale much better under NPTL than under LinuxThreads. NPTL threads does not require an additional manager thread. Supported by Linux 2.6 onwards. Threads are created using **`clone()`**, that specifies all the flags of LinuxThreads and more:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND |
CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |
CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```

To discover thread implementation on your system give following command:

```
$ getconf GNU_LIBPTHREAD_VERSION
$ getconf GNU_LIBC_VERSION
```

On systems that provides both NPTL and LinuxThreads, if you want to run a multithreaded program with LinuxThreads, you set the following environment variable to a kernel version that doesn't provided support for NPTL (e.g., 2.2.5)

```
$ export LD_ASSUME_KERNEL=2.25
```

Pthreads API

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

Pthreads 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

Pthreads 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()`

Pthreads 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()`

Thread Creation Proof of Concept

`t0.c - t4.c`

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 over write the static variable with its own return value and return value written by the first thread will be over written
- So the best solution is to create the variable to be returned in the heap instead of stack

Returning Value from a Thread Function
Proof of Concept
`rv1.c - rv2.c`

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 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

Creating Arrays of Threads
Proof of Concept

`array_threads1.c - array_threads3.c`

matrix_mul.c

Program should read a text file containing the size and data of two matrices X_{axb} and Y_{bxc}. Create two 2xD dynamic integer arrays, and populate them as per the matrices data in the file. Create the third 2xD dynamic integer array Z_{axc}, which will contain the product of two input matrices. Create n threads in heap where n=a*c. Each thread should compute one value of the product matrix. Once all the threads are done, the main thread should display the input matrices and the product matrix on stdout

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 to `gcc`

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` affects the include file in three ways:

- The `errno` refers to thread specific error location rather than global variable
- Library functions are no longer defined as macros; e.g. `getc()` and `putc()`. In multi-threaded programs, some standard I/O library functions require additional locking which macros don't perform, so we must call function instead
- The reentrant versions of the standard library functions are used, e.g., instead of `gethostbyname()`, its reentrant equivalent `gethostbyname_r()` is used

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 **`gettid()`**	TID returned by **`pthread_self()`**
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()`

Proof of Concept: `id_threads1.c, id_threads2.c`

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 crates 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()`

Pthreads 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

Pthreads 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();
```

Thread Attributes
Proof of Concept
`attr_threads.c`

Point to Ponder

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

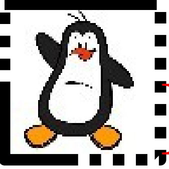
The UNIX signal model was designed with the UNIX process model in mind, so there are some significant conflicts between the signal and thread models. Combining signals and threads is complex and should be avoided whenever possible. Some key points to be kept in mind are:

- 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?

- When any thread calls one of the `exec()` functions, the calling program is completely replaced and all threads, except the one that called `exec()`, vanish immediately
- 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
- After an `exec()` the thread ID of the remaining thread is unspecified



Point to Ponder

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?

- The child process is created with a single thread – the one that called the `fork()`
- It is recommended that a `fork()`, in a multithreaded process should always be followed by an immediate `exec()` call, so that all the global variables as well as all pthread objects (mutexes and condition variables) disappear, as the child program overwrites the memory of the process
- If there is no `exec()` after the `fork()`, then the state of global variables as well as all pthread objects (mutexes and condition variables) are preserved in the child, which may cause problems in the child program. So for programs that uses `fork()` that is not followed by an `exec()`, the pthreads API provides a mechanism for defining fork handlers using the function `pthread_atfork()`. These fork handlers are preserved after a `fork()`, but not preserved after an `exec()`

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 un-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. (Places in a thread's execution where it can be cancelled are called cancellation points)

Things To Do

If you have problems visit me in counseling hours. . . .