CMP325
Operating Systems
Lecture 08

Thread Management

Muhammad Arif Butt, PhD

Note:
Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:

OS with Linux:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8

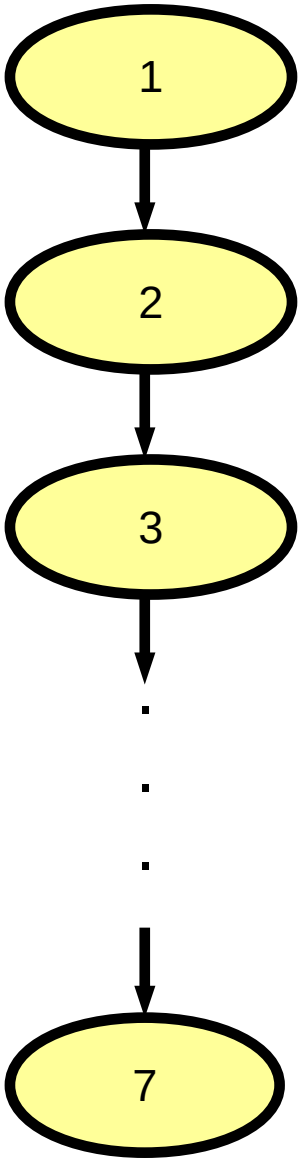
System Programming:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW

Today's Agenda

- Review of previous lecture
- Concurrent and Parallel Programming
- Introduction to Threads
- Multi threading
- Merits and Demerits of Threads
- User Level vs Kernel Level Threads
- Threading Models
  - Many to One
  - One to One
  - Many to Many
- Using POSIX pthread library calls to create multi-threaded programs

Concurrent / Parallel Programming

Sequential Programming



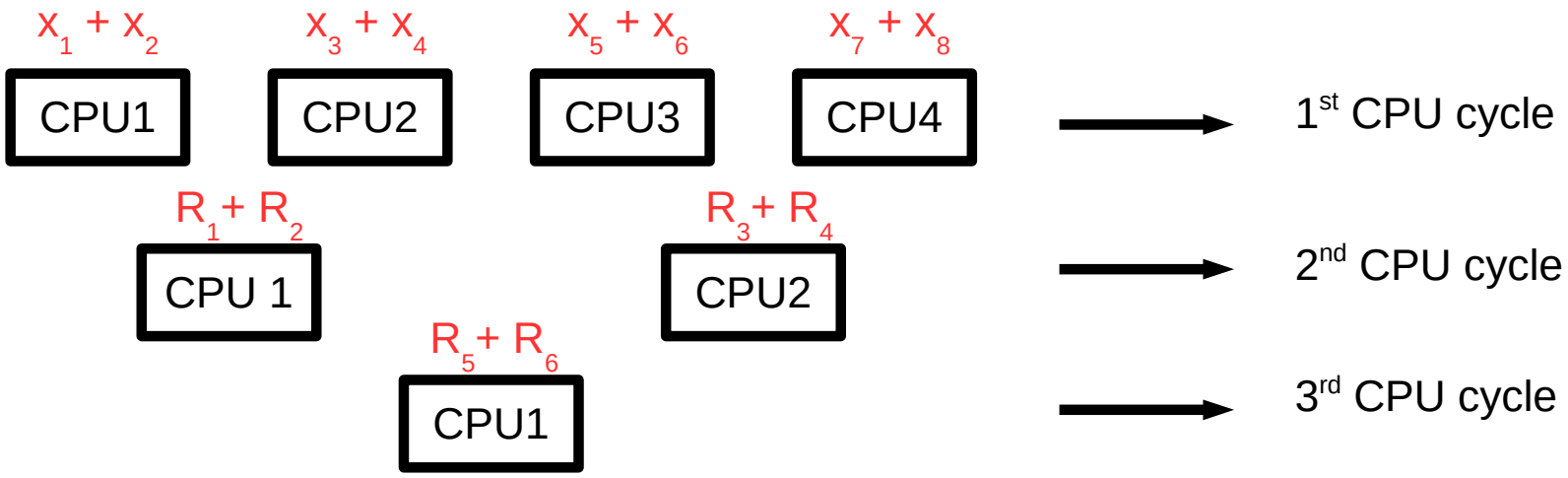
Suppose we want to add eight numbers x_1, x_2, x_3, ... x_8

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:

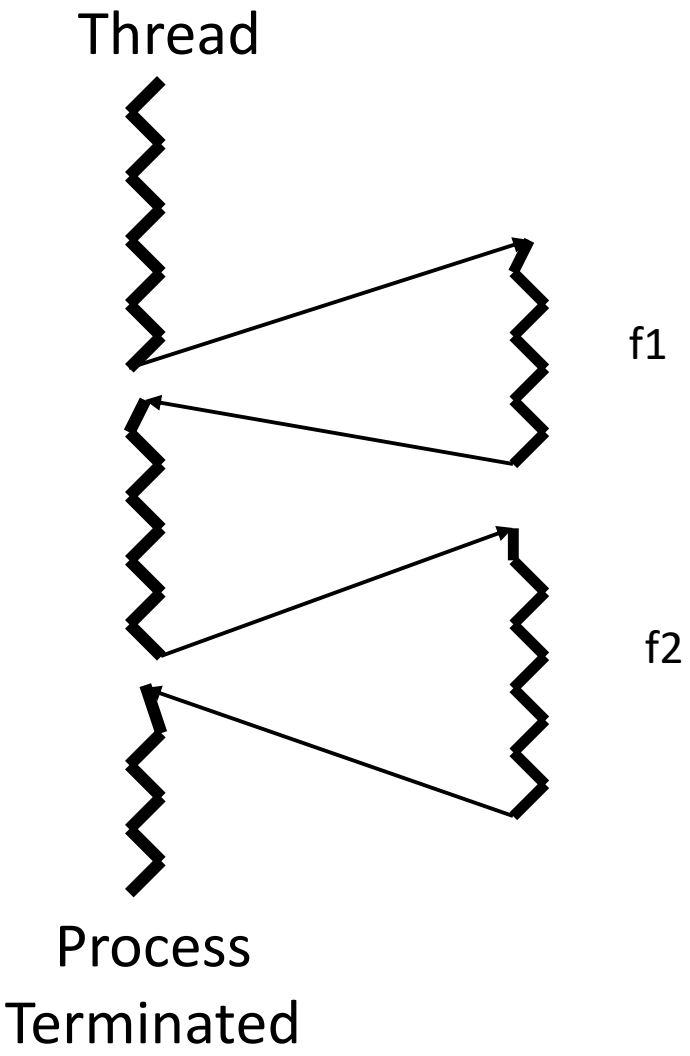
- **Resource ownership-** process includes a virtual address space to hold the process image
- **Scheduling-** follows an execution path that may be interleaved with other processes

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

Single Threaded Process

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

f1(…)
{  …  }

f2(…)
{  …  }
```

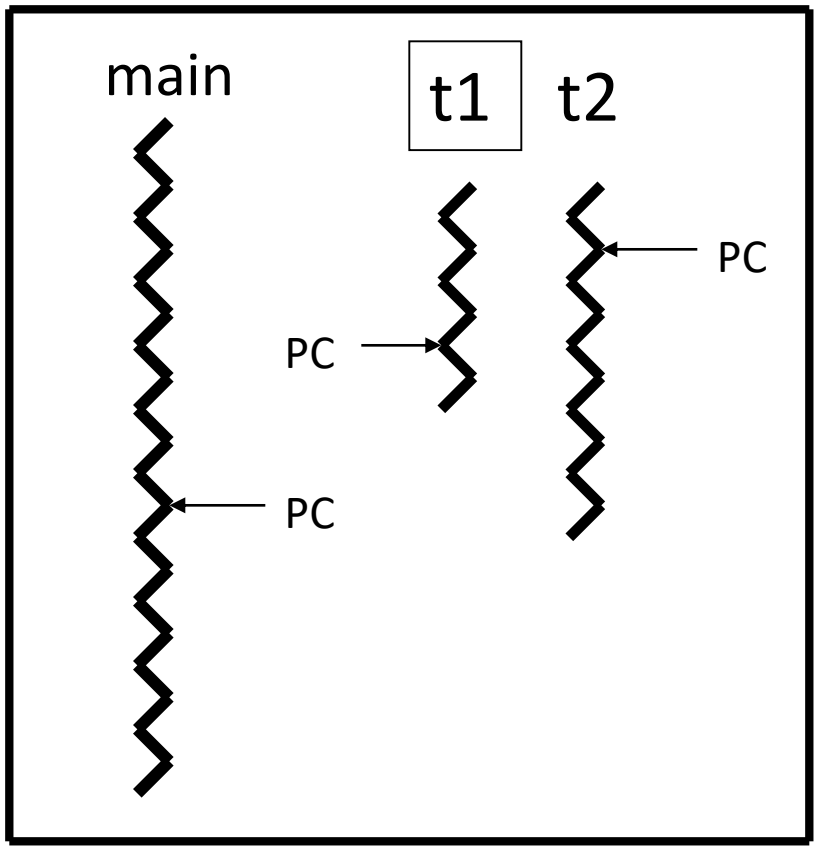


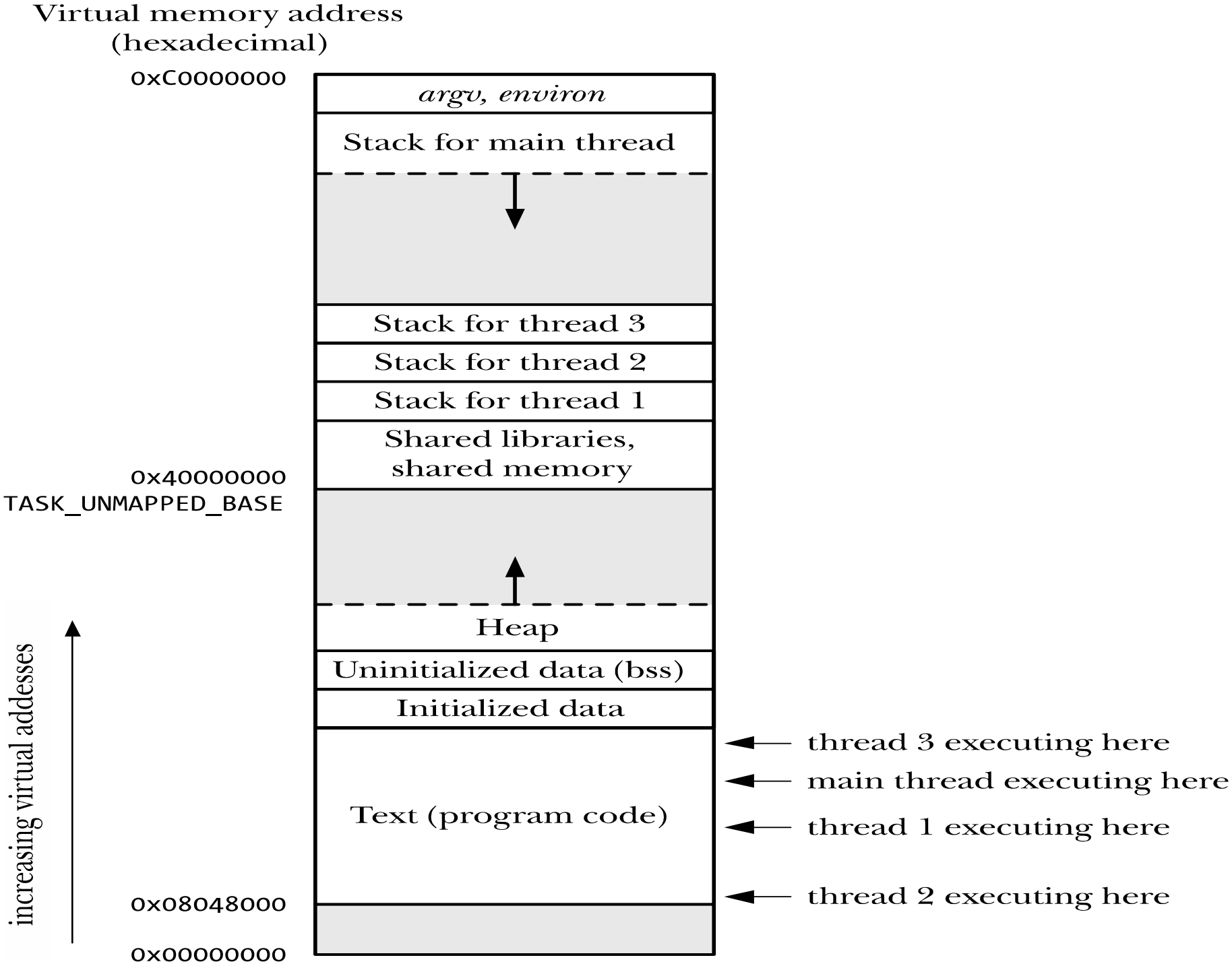
Thread Concept

- Previous slide is an example of a process with a single thread
- Suppose we want that 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.** A single program made up of a number of different concurrent activities; sometimes called multitasking, as in Ada
- Multithreading works similar to multiprogramming. The CPU switches rapidly back and forth among threads providing the illusion that threads are running in parallel

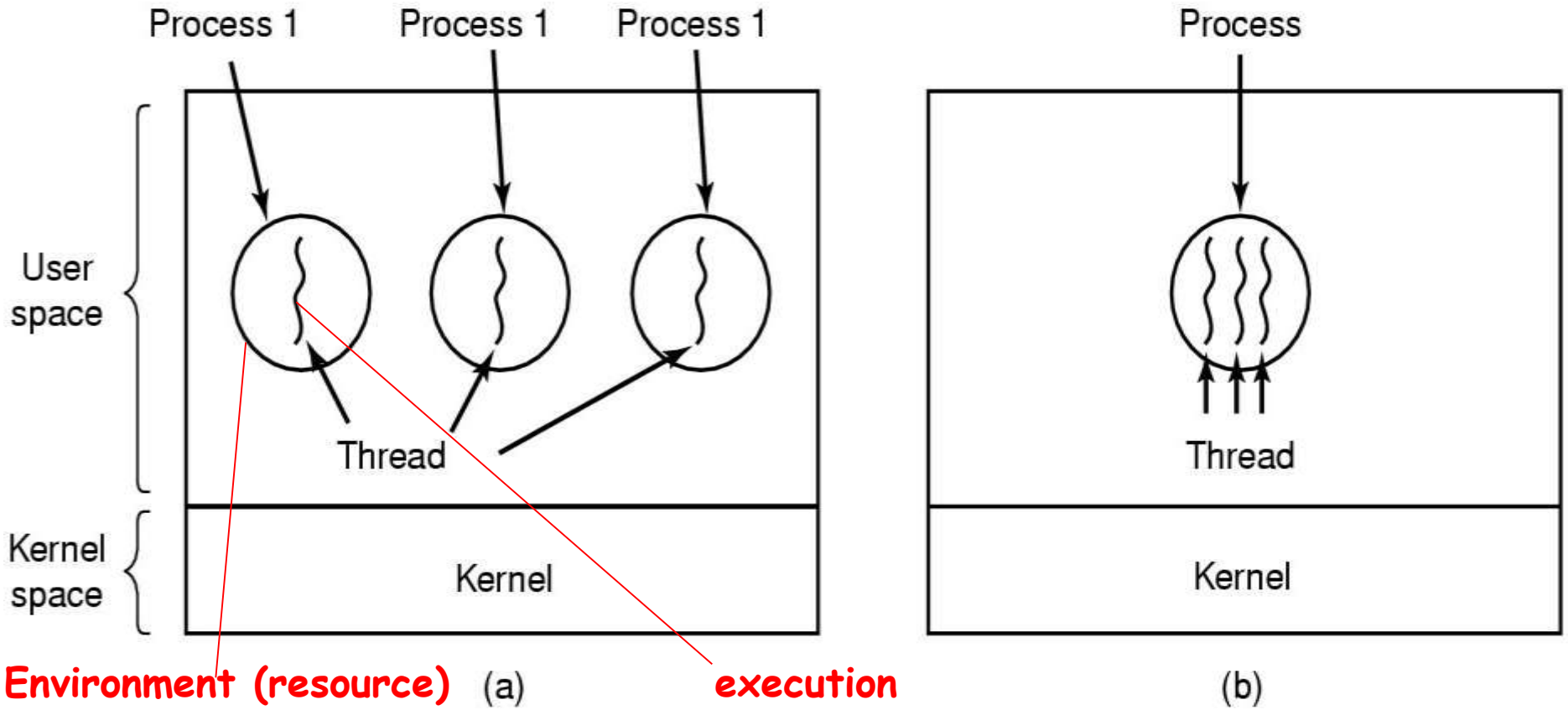
Multi Threaded Process

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

f1(…)
{  …  }

f2(…)
{  …  }
```




Virtual memory address
(hexadecimal)
0xC0000000
argv, environ
Stack for main thread
Stack for thread 3
Stack for thread 2
Stack for thread 1
Shared libraries,
shared memory
0x40000000
TASK_UNMAPPED_BASE
Heap
Uninitialized data (bss)
Initialized data
Text (program code)
0x08048000
0x00000000
increasing virtual addesses
thread 3 executing here
main thread executing here
thread 1 executing here
thread 2 executing here

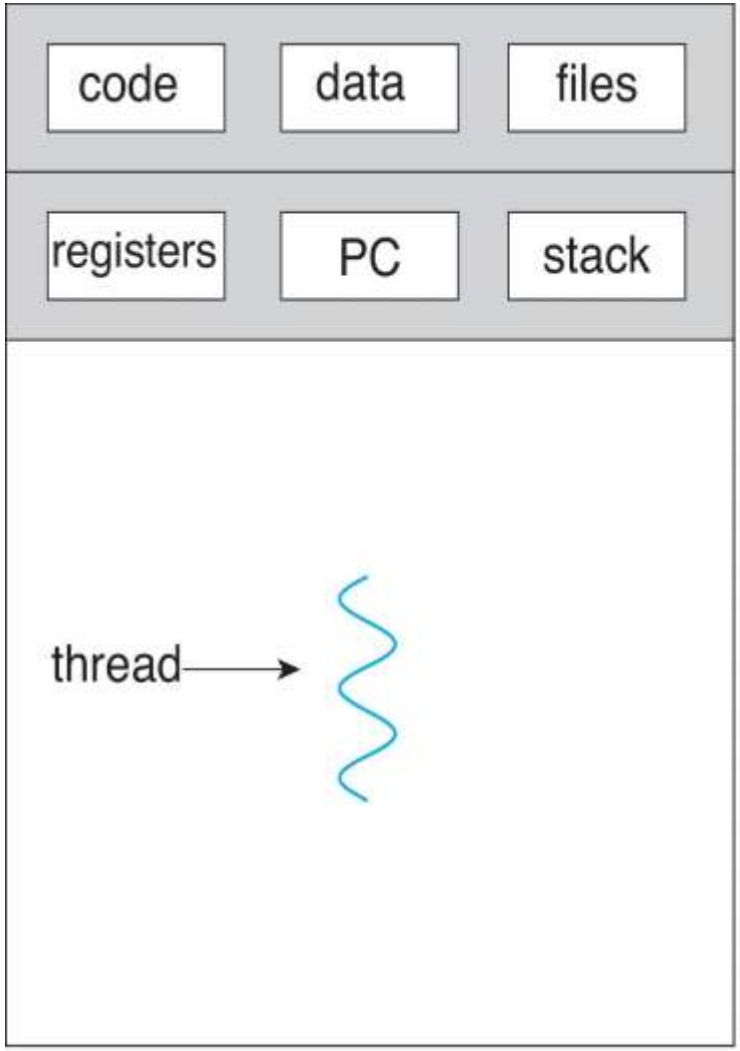
Single and Multithreaded Processes



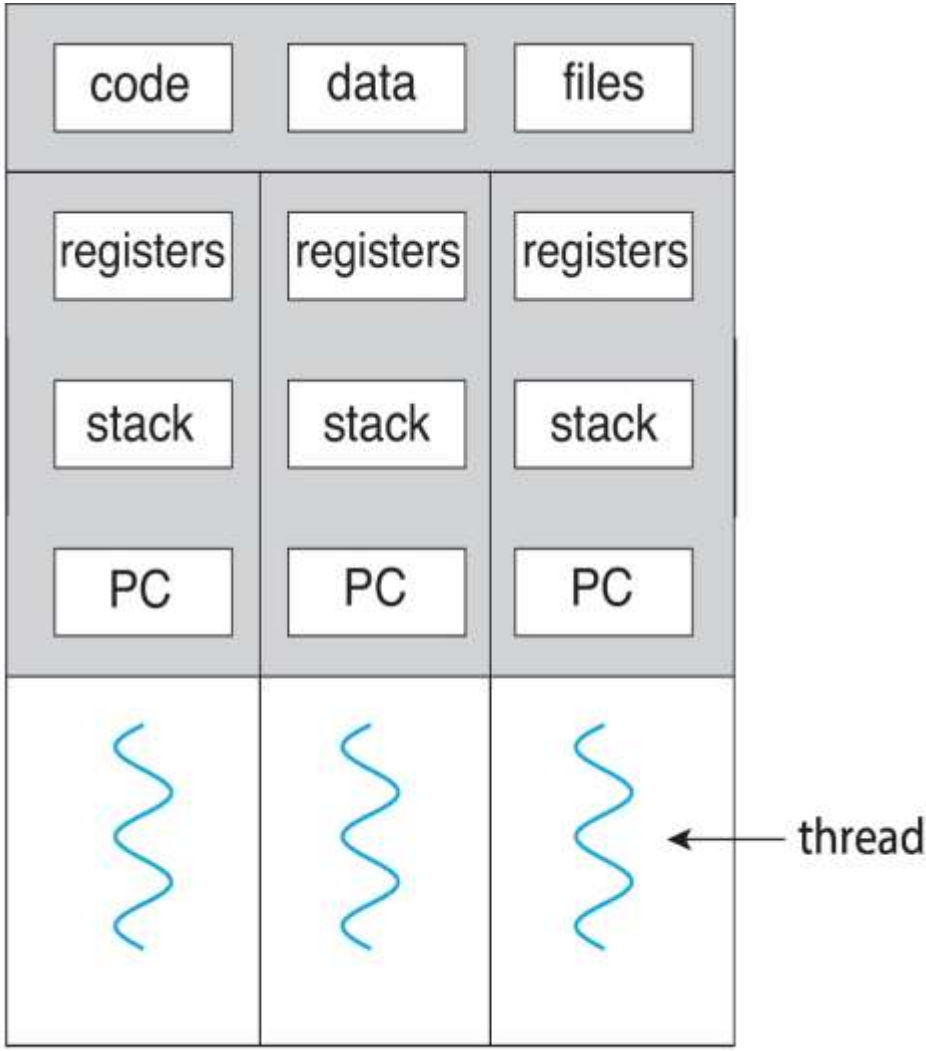
(a) Three processes, each with one thread
(b) One process with three threads

Processes are used to group resources together; Threads are the entities scheduled for execution on the CPU.

Single and Multithreaded Processes



single-threaded process



multithreaded process

Multi-Threaded Process

Threads within a process share :

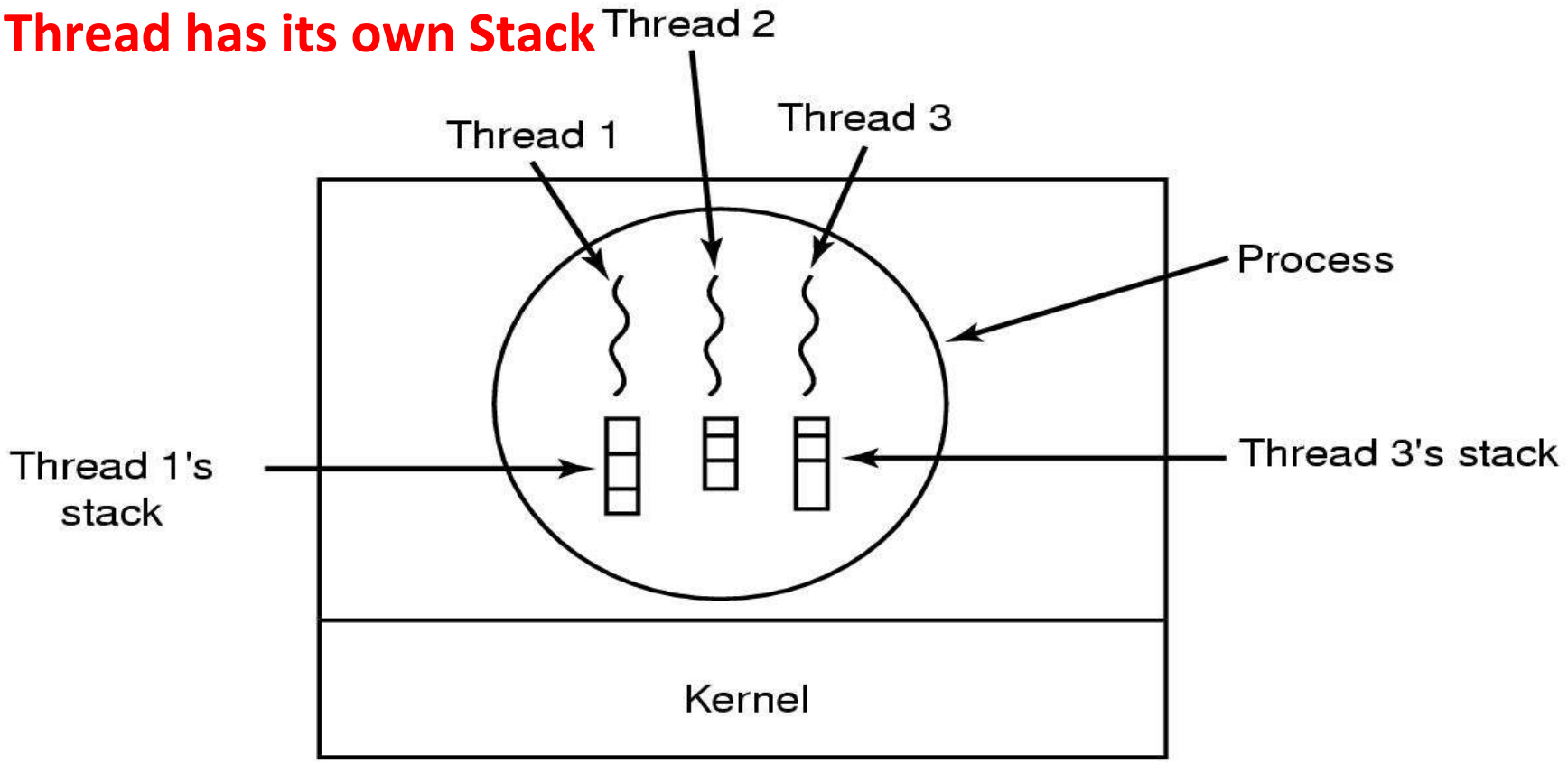
- PID, PPID, PGID, SID, UID, GID
- Controlling Terminals
- Code and Data Section
- Global Variables
- Open files via PPFDT
- Signal Dispositions
- Umask value
- Current Working Directory
- Interval Timers
- CPU time consumed
- Resource Limits
- Nice value
- Record locks (using `fcntl()`)

Threads have their own:

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

Single and Multithreaded Processes

Each Thread has its own Stack



- Each thread stack contains one frame for each procedure that has been called but not yet returned from
- This frame contains the procedure's local variables and their return address to use when the procedure call has finished
- For example, if procedure X calls procedure Y and this one calls procedure Z, while Z is executing the frames for X, Y, Z will be on the stack
- Each thread will generally call different procedures and thus has a different execution history

Threads vs Processes

Similarities

- A thread can also be in one of many states like new, ready, running, blocked, terminated
- Like processes only one thread is in running state (single CPU)
- Like processes a thread can create a child thread

Differences

- No "automatic" protection mechanism is in place for threads—they are meant to help each other
- Every process has its own address space, while all threads within a process operate within the same address space

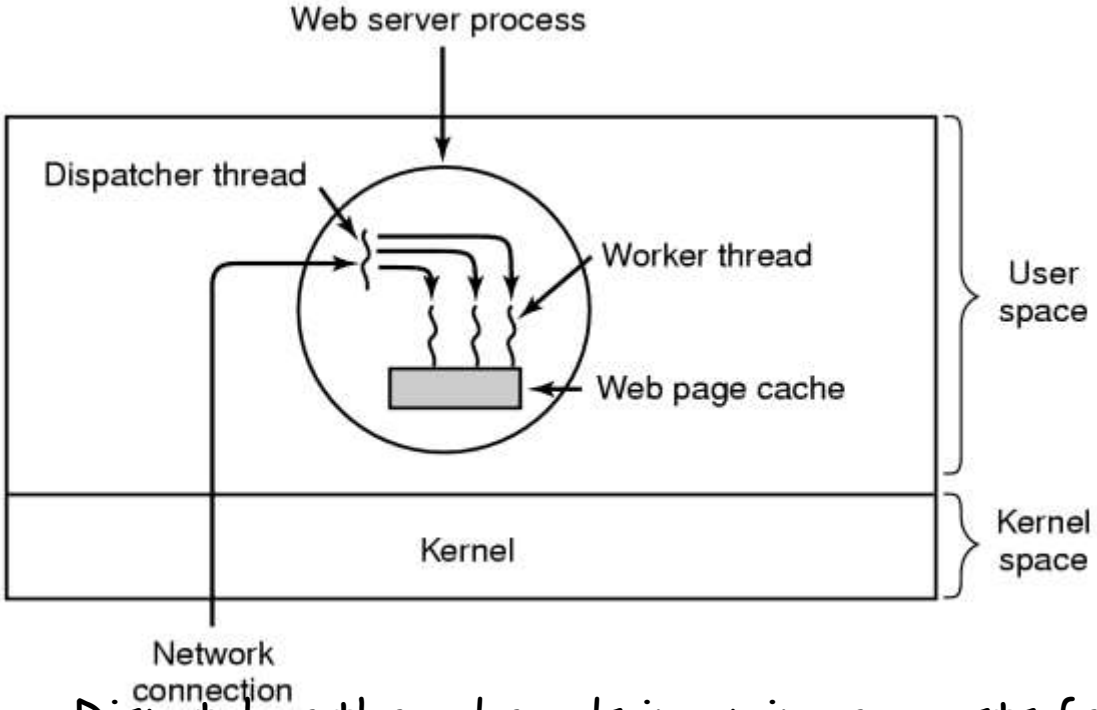
Benefits of Threads

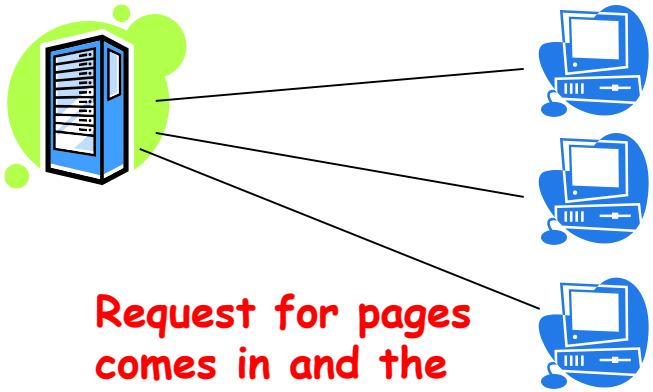
- **Responsiveness.** Multi-threaded servers (e.g., browsers) can allow interaction with user while a thread is formulating response to a previous user query (e.g., rendering a web page)
- **Resource sharing.** Process resources (code, data, etc.) are shared by all threads. OS resources (PCB, PPFDT, etc.) are also shared by all threads
- **Economy.** Take less time to create, schedule, and terminate a thread. **Solaris 2**: Thread Creation is thirty times faster than Process Creation and Thread Switching is five times faster than process switching
- **Performance**. In multi-processor and multi-threaded architectures (e.g., Intel's P-IV HT) each thread may run on a different processor in parallel

Disadvantages of Threads

- **Problems due to Shared memory**
  - Race Problem
  - Mutual Exclusion needs to be implemented on shared memory to allow multiple threads to access data for write/update
- **Many library functions are not Thread Safe**
  - For resource sharing, synchronization is needed between threads
- **Lack of Robustness**
  - A severe error caused by one thread (e.g. segmentation fault) terminates the whole process

Thread Usage: Web Server





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

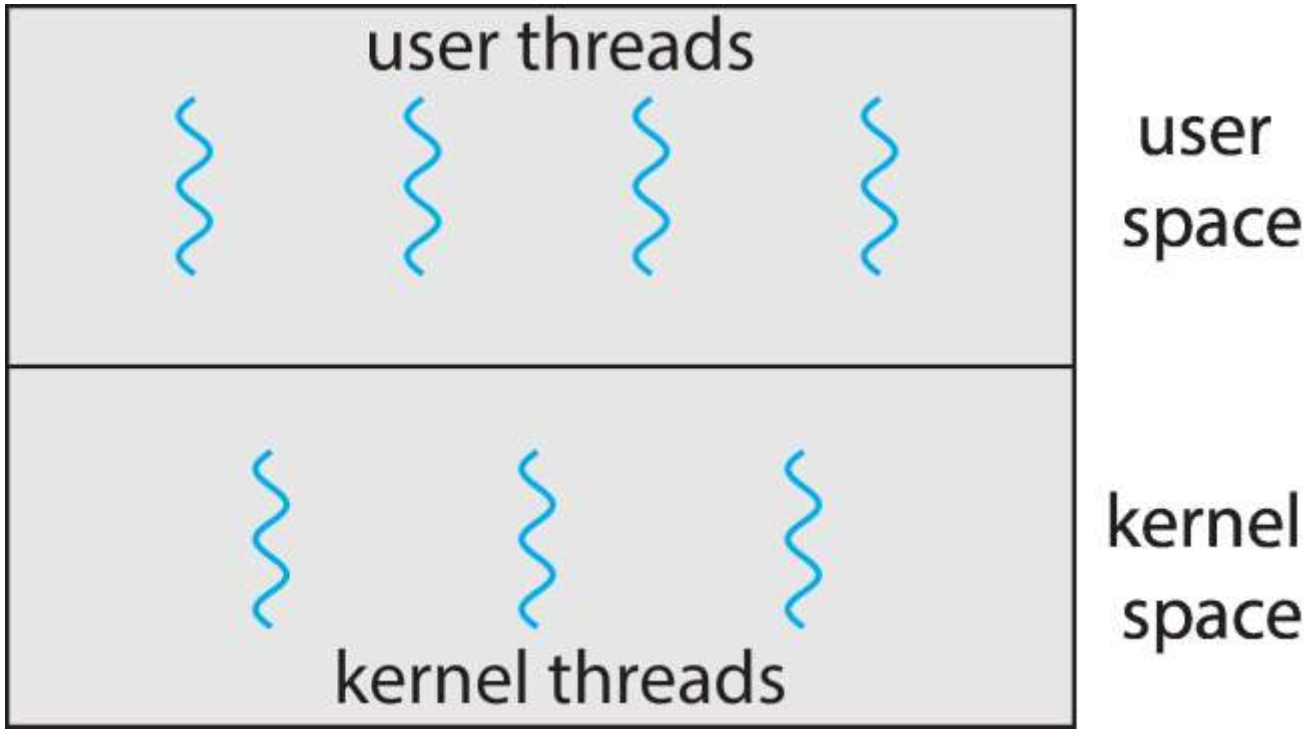
- Dispatcher thread reads incoming requests from the NW.
- After examining the request, it chooses an idle worker thread and hands it the request. It also wakes up the worker from blocked state to ready state.
- Worker now checks to see if the request can be satisfied from the Web page cache, to which 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.

User Threads vs Kernel Threads

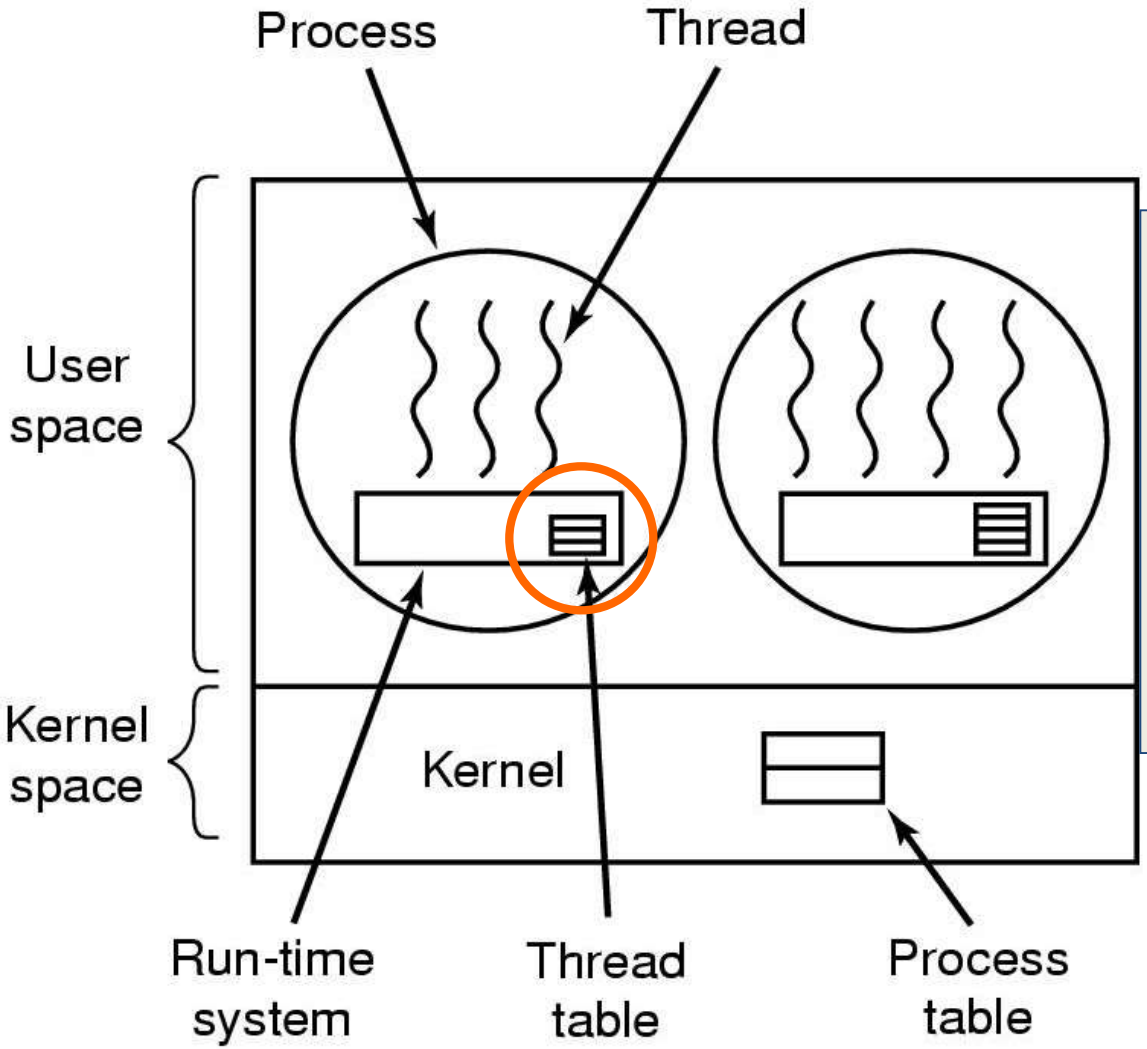
User Threads

- Thread management is done by user-level threads libraries (eg, POSIX Pthread, Win32 threads, Java threads, Solaris2 Threads, Mach C Threads) and Kernel is not aware of the existence of threads
- An application can be programmed to be multithreaded by using user level thread library, which contains code for:
  - Thread creation and termination
  - Thread scheduling
  - Saving and restoring thread context
  - Passing messages and data between threads
- By default an application begins with a single thread, within a process managed by Kernel. Later the application may spawn a new thread within the same process by invoking spawn utility in the thread library

User Threads vs Kernel Threads



Implementing Threads in User Space



- Thread library used.
- **Thread table** maintained in user space.
- All thread management is done in user space by library
- Kernel knows nothing about threads.

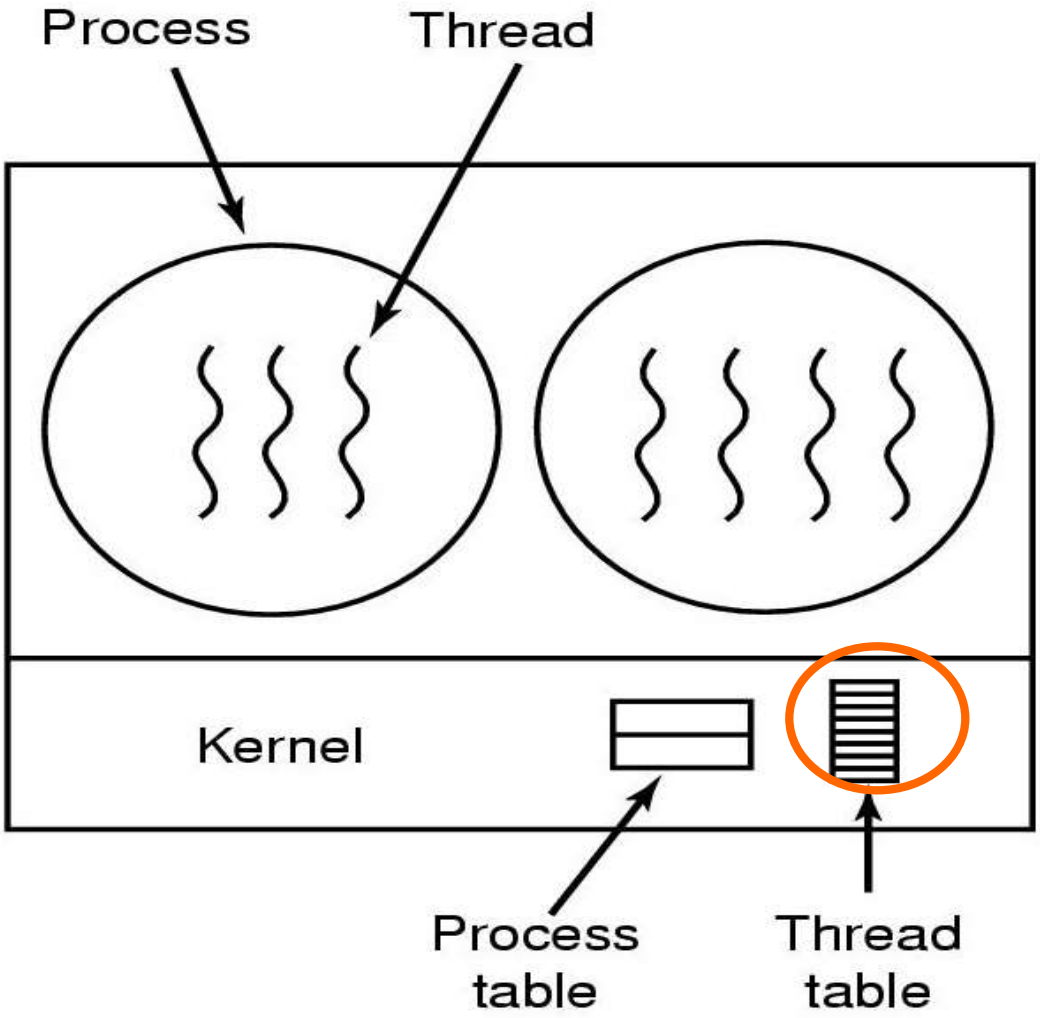
E.g.: pthread library

User-level Threads

Kernel Threads

- Thread management done by kernel and Kernel is aware of threads
- Kernel level threads are supported in almost all modern operating systems:
  - Windows NT/XP/2000
  - Linux
  - Solaris
  - Tru64 UNIX
  - Mac OS X
- There must be a relationship between user threads and kernel threads. There exist three common models of this interaction (1:1, M:1 and M:N)

Implementing Threads in the Kernel



- OS knows about individual threads within each process.
- Thread table maintained by kernel.
- E.g.: Windows 2K/XP

Kernel-level Threads

Thread Implementation Models

Thread Implementation Model (M:1)

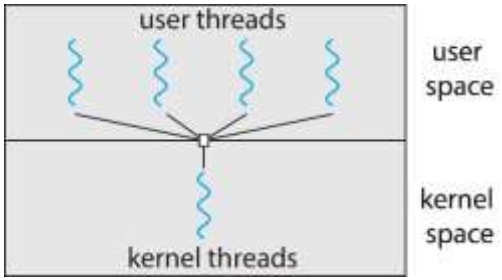
In Many-to-one (M:1) threading implementation, all of the details of thread creation, termination, scheduling, synchronization, and so on are handled entirely within the user-space. Kernel knows nothing about the existence of multiple threads within the process

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 Model (1:1)

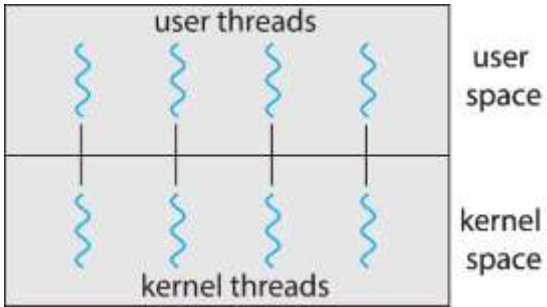
In one-to-one (1:1) threading implementation, each thread maps onto a separate kernel scheduling entity (KSE). All of the details of thread creation, termination, scheduling, synchronization and so on are handled by system calls inside the kernel

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

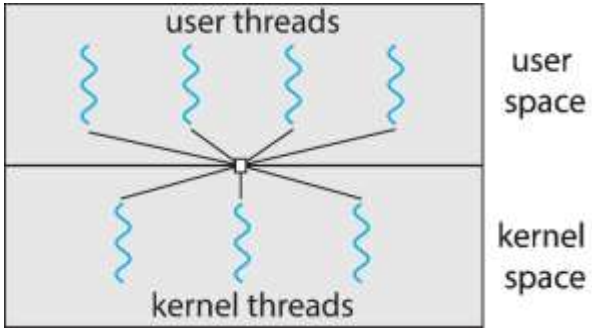


Thread Implementation Model (M:N)

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




LIVE FREE OR DIE
UNIX*
TRADEMARK OF BELL LABS*

Thread Libraries

- Thread libraries provides programmers with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by OS
- **Pthreads** may be provided either as user-level or kernel-level. Pthread is a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- **Java Threads** managed by the JVM. Typically implemented using the threads model provided by underlying OS. Java threads may be created by: Extending Thread class, or by implementing the Runnable interface. If underlying OS is Windows, then implemented using Win32 API; if it is Linux, then Pthreads.

Important POSIX System Calls

Thread Management

Call	Description
pthread_create()	Similar to fork()
pthead_join()	Similar to waitpid()
pthread_exit(void *status)	To terminate a thread

- Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.

Thread Creation

```
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

Thread Termination

```
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()`

Joining a Thread

```
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 0

```
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 1

```
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;
}
```

Compiling a multi-threaded Program

- Multi-Threaded program need to be linked with the thread library /usr/lib/libpthread.so

```
$ gcc -c t1.c -D_REENTRANT
$ gcc t1.o -o t1 -lpthread
$./t1
PUCITARIFARIFPUCITARIFPUCITPUCITARIFPUCITARIF
Bye Bye from main thread
```

Question: What are the advantages of compiling multi-threaded programs using the -D_REENTRANT flag?

Example 2

```
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<1000;i++)
        fprintf(stderr, "%c",'X');
   pthread_exit(NULL);
}
```

```
void * f2(void * arg){
   for(int i=0; i<800;i++)
        fprintf(stderr, "%c",'O');
   pthread_exit(NULL);
}
```

Example 3

```
int main(int argc, char* argv[]){
   int countofX = atoi(argv[1]);
   int countofO = atoi(argv[2]);
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, f1, (void*)&countofX);
   pthread_create(&tid2, NULL, f2, (void*)&countofO);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
   printf("\nBye Bye from main thread\n");
   return 0;}
```

```
void * f1(void * arg){
   int ctr = *((int*)arg);
   for(int i=0; i<ctr; i++)
       fprintf(stderr, "%c", 'X');
   pthread_exit(NULL);
}
```

```
void * f2(void * arg){
   int ctr = *((int*)arg);
   for(int i=0; i<ctr; i++)
       fprintf(stderr, "%c", 'O');
   pthread_exit(NULL);}
```

Example 4

```
struct mystruct{
   char character;   int count;
};
void * f1(void *);
int main(){
   pthread_t tid1, tid2;
   struct mystruct t1_args, t2_args;
   t1_args.character = 'X';   t1_args.count = 1000;
   pthread_create(&tid1, NULL, f1, (void*)&t1_args);
   t2_args.character = 'O';   t2_args.count = 800;
   pthread_create(&tid2, NULL, f1, (void*)&t2_args);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
   printf("\nBye Bye from main thread.\n");
   return 0;}
```

```
void * f1(void * args){
   struct mystruct p = *(struct mystruct*)args;
   for (int i = 0; i < p.count; i++)
       putc(p.character,stdout);
   pthread_exit(NULL);
}
```

Example 5

```
int main(int argc, char* argv[]){
if(argc != 3){
        printf("Must pass two file names.\n");
        exit(1);
   }
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, func, (void*)argv[1]);
   pthread_create(&tid2, NULL, func, (void*)argv[2]);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
   printf("Bye Bye from main thread\n");
   return 0;
}
```

```
void* func(void* args){
   char* filename = (char*)args;
   int ctr = 0;   char ch;
   int fd = open(filename, O_RDONLY);
   while((read(fd, &ch, 1)) != 0)
        ctr++;
   close(fd);
   printf("Characters in %s: %d\n", filename, ctr);
   pthread_exit(NULL);}
```

Example 6 (Race Condition)

```
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<100000000;i++)
       balance++;
   pthread_exit(NULL);
}
```

```
void * dec(void * arg){
   for(long i=0;i<100000000;i++)
       balance--;
   pthread_exit(NULL);
}
```

Points to Ponder

Points 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

Points 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

Points 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()`

Points 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)

SUMMARY

We're done for now, but Todo's for you after this lecture…

- Go through the slides and Book Sections: 4.1, 4.2, 4.3, 4.4
- Write down the programs discussed in class in vim editor and execute them. Make variations in the programs and understand the underlying details
- Try to understand why program in Example 6 gives different results when run multiple times
- Understand the difference between fork() and clone() system call
- Study books and understand the concept of
  - Thread Pool
  - Thread Cancellation
  - Signal handling in multithreaded programs

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