Operating Systems

4. Multithreaded Programming

Recall the Main Concepts Behind Processes

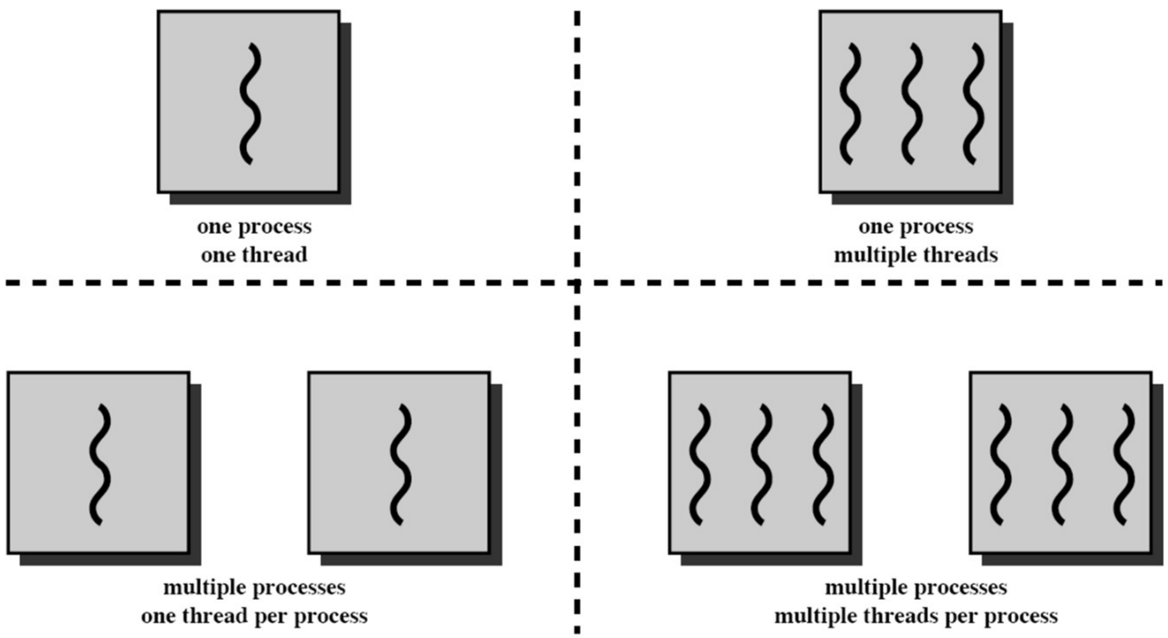
1. Resource Ownership
   - Virtual address space to hold the process image
     - Program
     - Data
     - Stack
     - Attributes
   - OS shields processes from interfering with each others resources
     - Protection

2. Scheduling/ Dispatching
   - Execution may be interleaved with other processes
   - Maintain execution states, etc.

- Can we decouple this functionality?

Processes and Threads

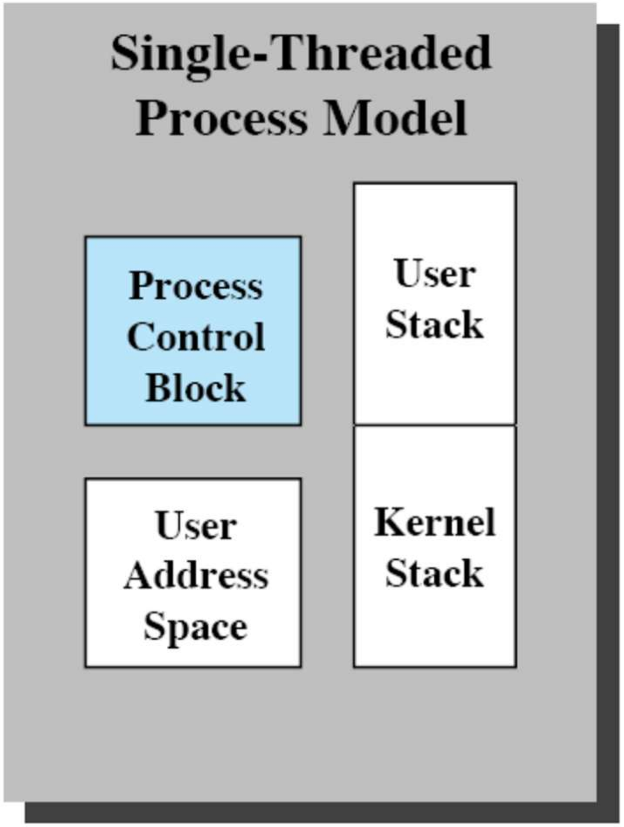
Multithreading: More than one entities can possibly execute in the same resource- (i.e., process-) environment (and collaborate better)



- Unit of dispatching
  - Referred to as a thread
- Unit of resource ownership
  - Referred to as a process or task

Processes vs. Threads

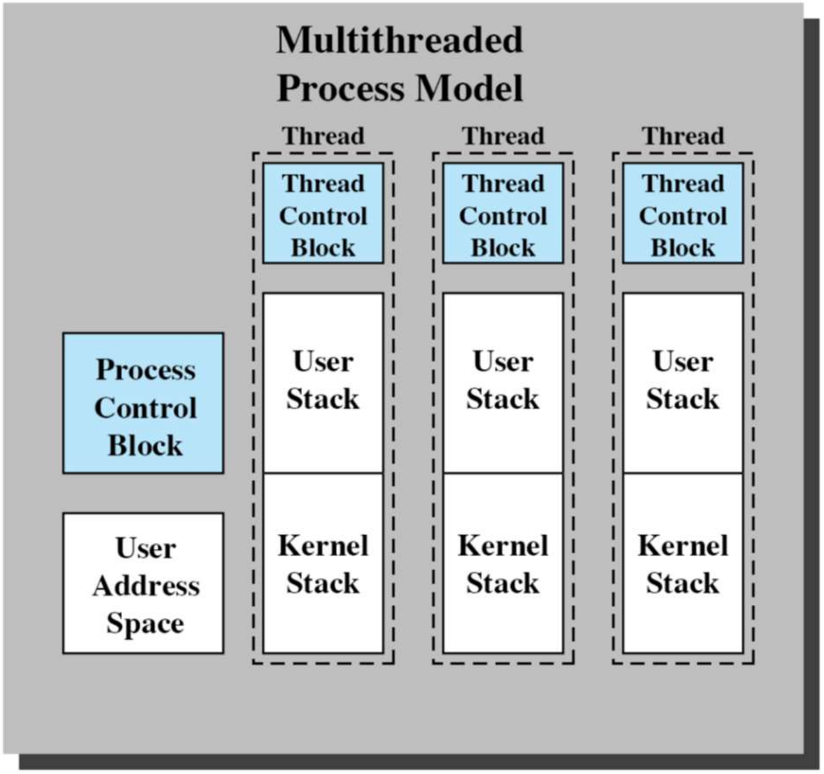
- Process characteristics
  - A virtual address space
    - ➢ Holds the process image
  - Global variables, files, child processes, signals and signal handlers



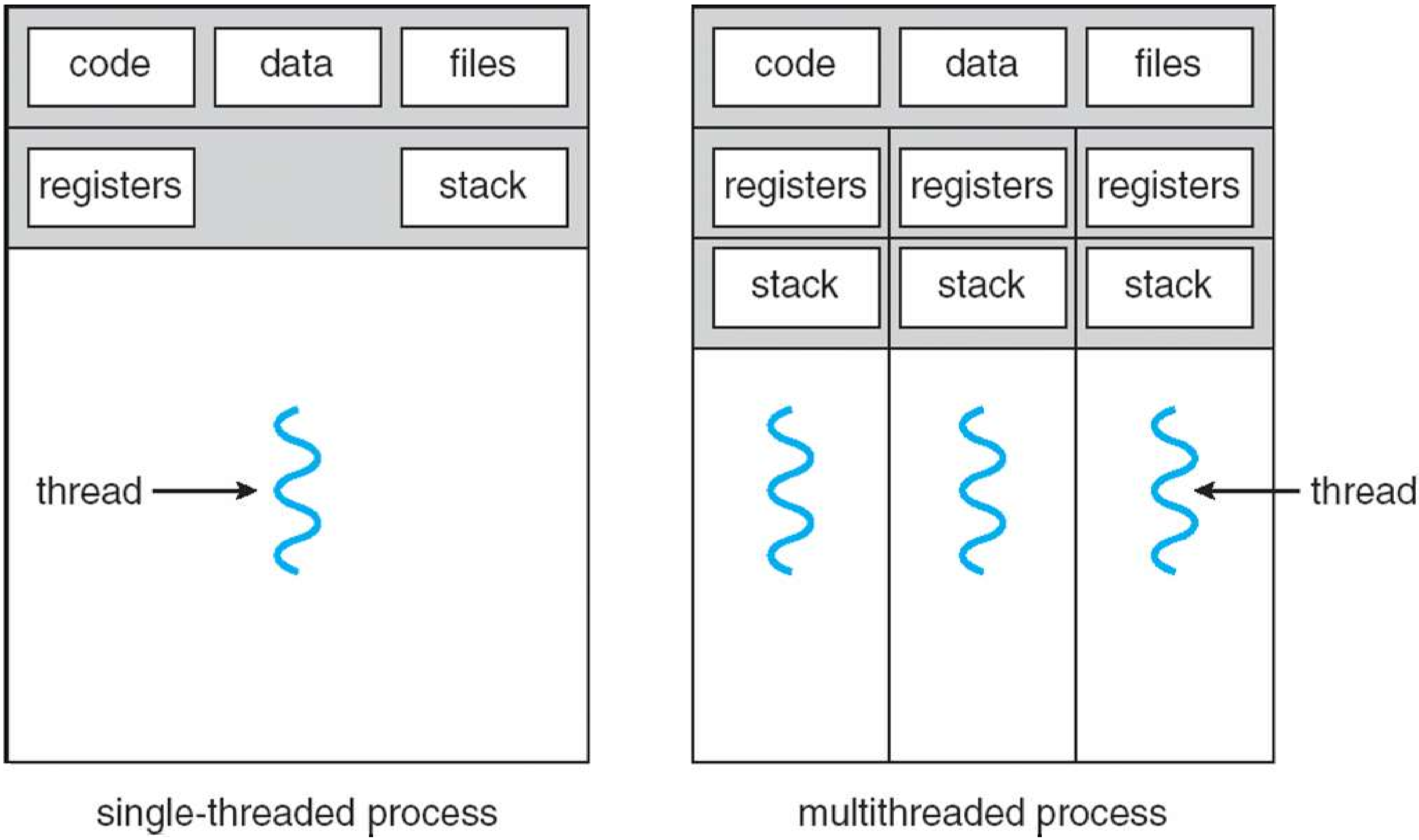
Processes vs. Threads

- Process characteristics
  - A virtual address space
    - ➢ Holds the process image
  - Global variables, files, child processes, signals and signal handlers

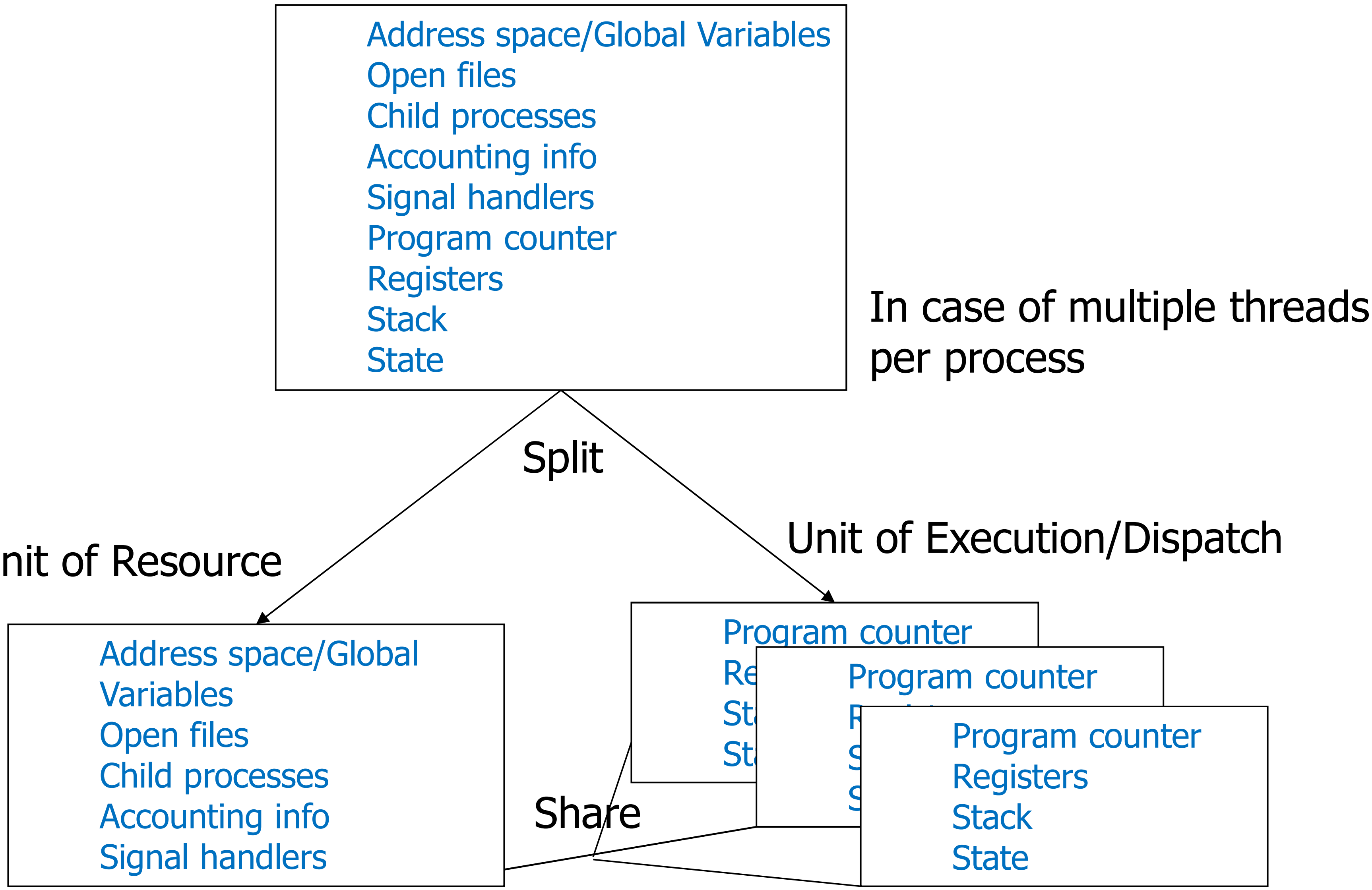
- Thread characteristics
  - An execution state, stack and context (saved when not running)
  - Access to the memory and resources of its process
    - ➢ All threads of a process share this
  - Some per-thread static storage for local variables



Processes vs. Threads



Processes vs. Threads



Benefits of Threads

- Easy/Lightweight Communication
  - Threads within the same process share memory and files
  - Communication does not invoke the kernel

- May allow parallelization within a process
  - I/O and computation to overlap
    - Recall historical evolution from uni-programming to multiprogramming
  - Concurrent execution in multiprocessors

- Takes less time to
  - Create/terminate a thread than a process
  - Switch between two threads within the same process

Uses of Threads

1. Overlap foreground with background work
   – Decouple interactions from processing
   – For instance processing message requests
2. Asynchronous processing
   – Backup while editing
3. Speed up execution
   – Parallelize independent actions
4. Modular program structure
   – Must be careful here, not to introduce too much extra overhead

Thread I Wait for message Store result in buffer

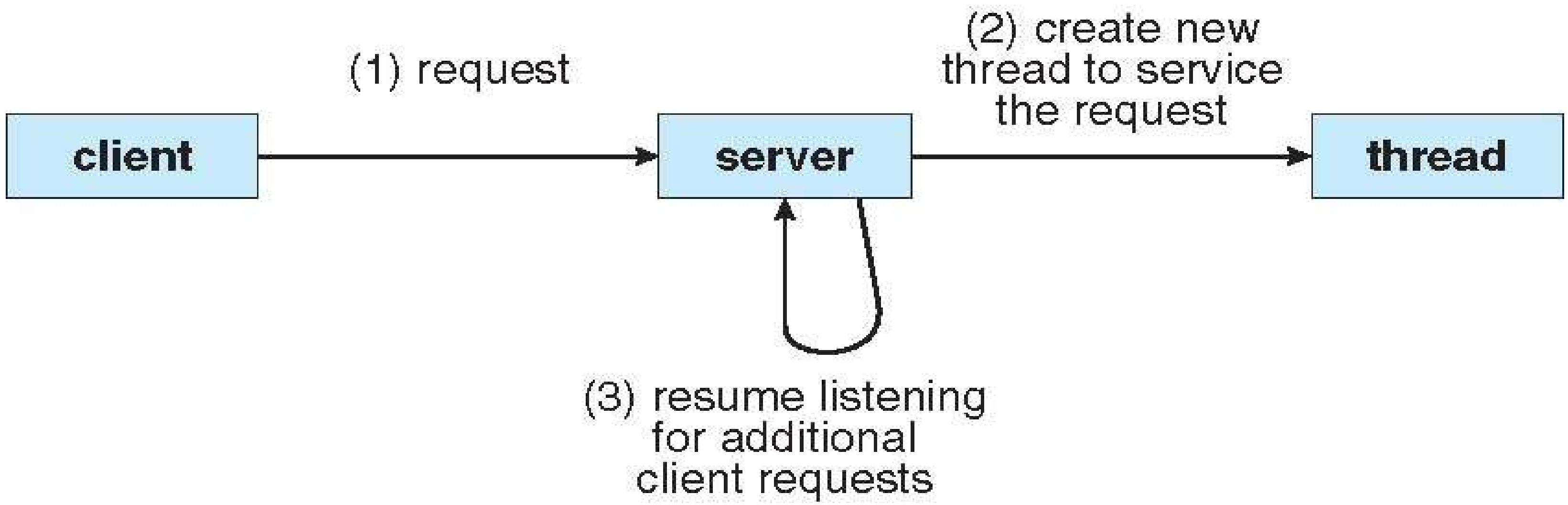
Thread II Wait for message Store result in buffer

Thread III Consume messages in buffer

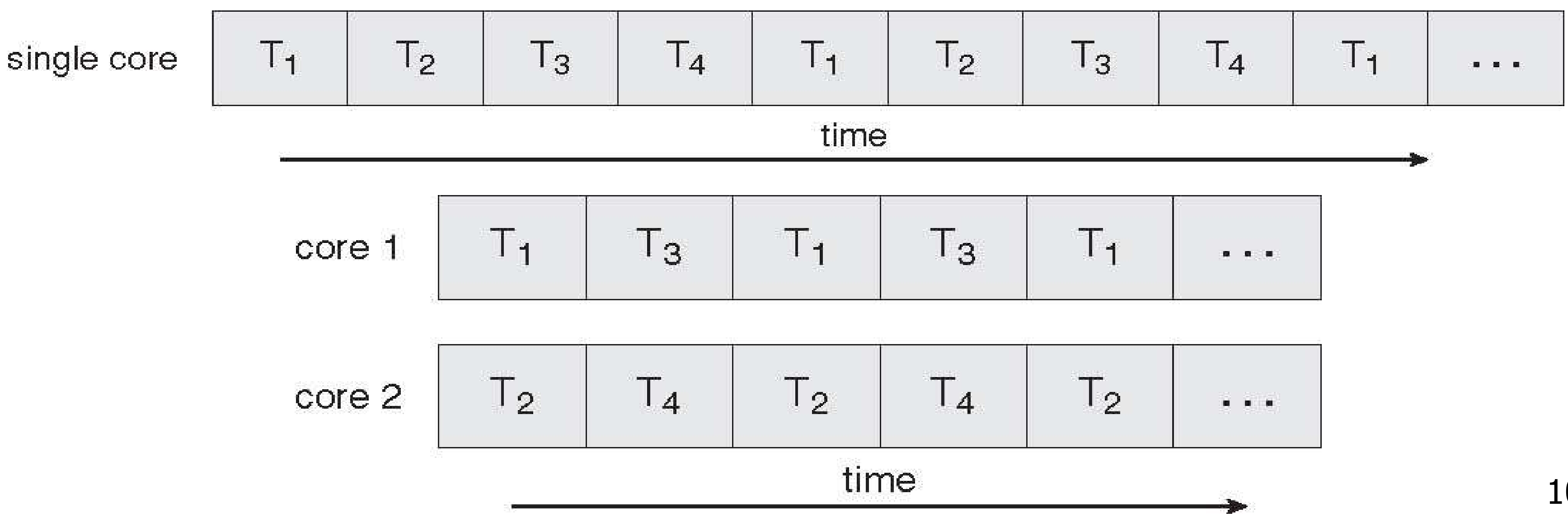
Thread IV Consume messages in buffer

⇒Increase throughput in message processing

Multithreaded Server Architecture – Example



Parallel execution on single core and multicore



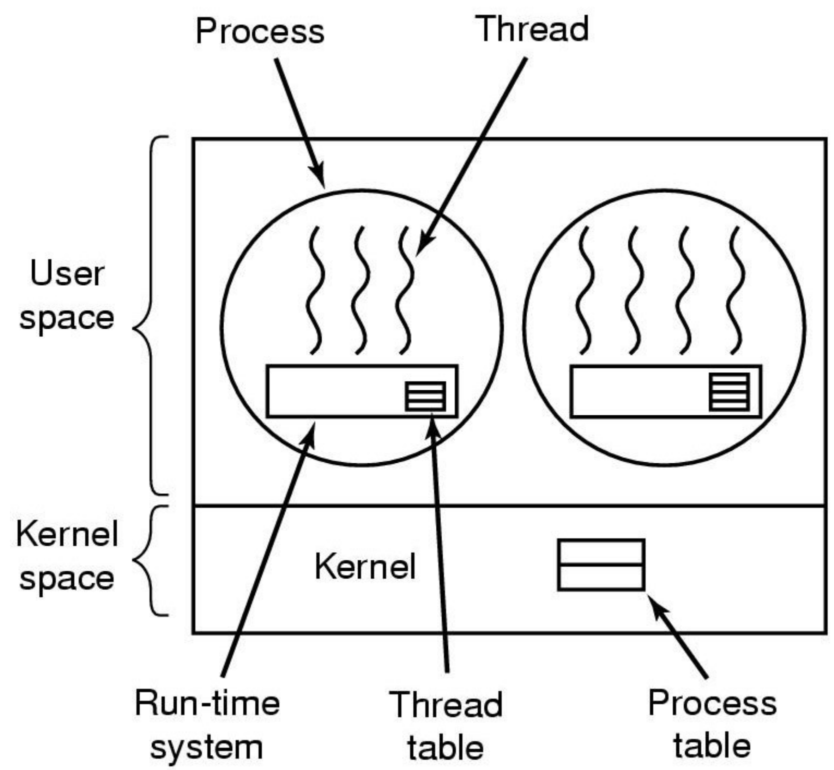
Effect of Suspension/Termination

- Suspending a process involves suspending all threads of the process since all threads share the same address space

- Termination of a process, terminates all threads within the process

Implementing Threads in User Space

- The kernel is not aware of the existence of threads
- Run-time system (thread-library in execution)
  - Responsible for bookkeeping, scheduling of threads
- Allows for customized scheduling
- Support for any OS
- But: problem with blocking system calls
  - Blocking of a thread causes blocking of the process
  - Threads cannot execute if a thread of the same process is blocked

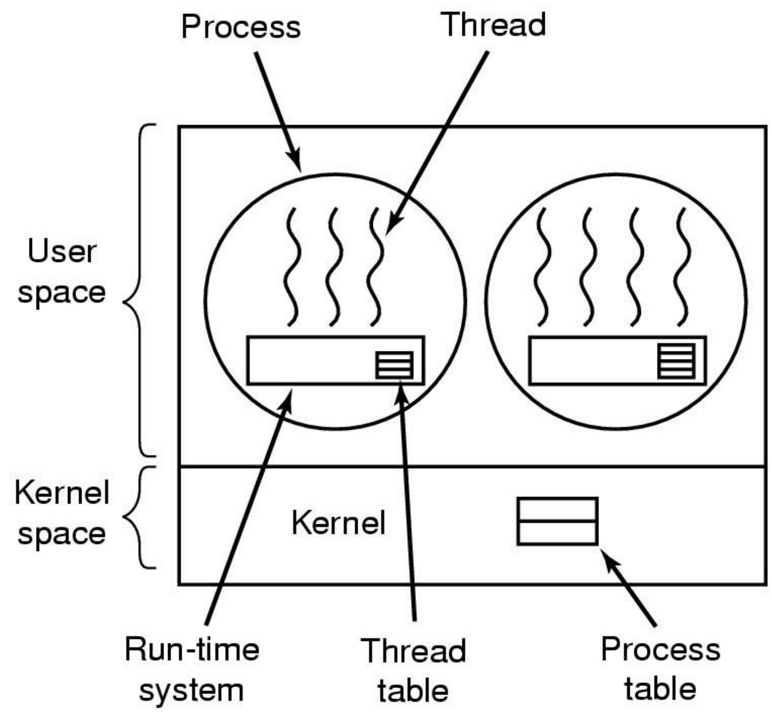


Implementing Threads in User Space

The thread library contains code for

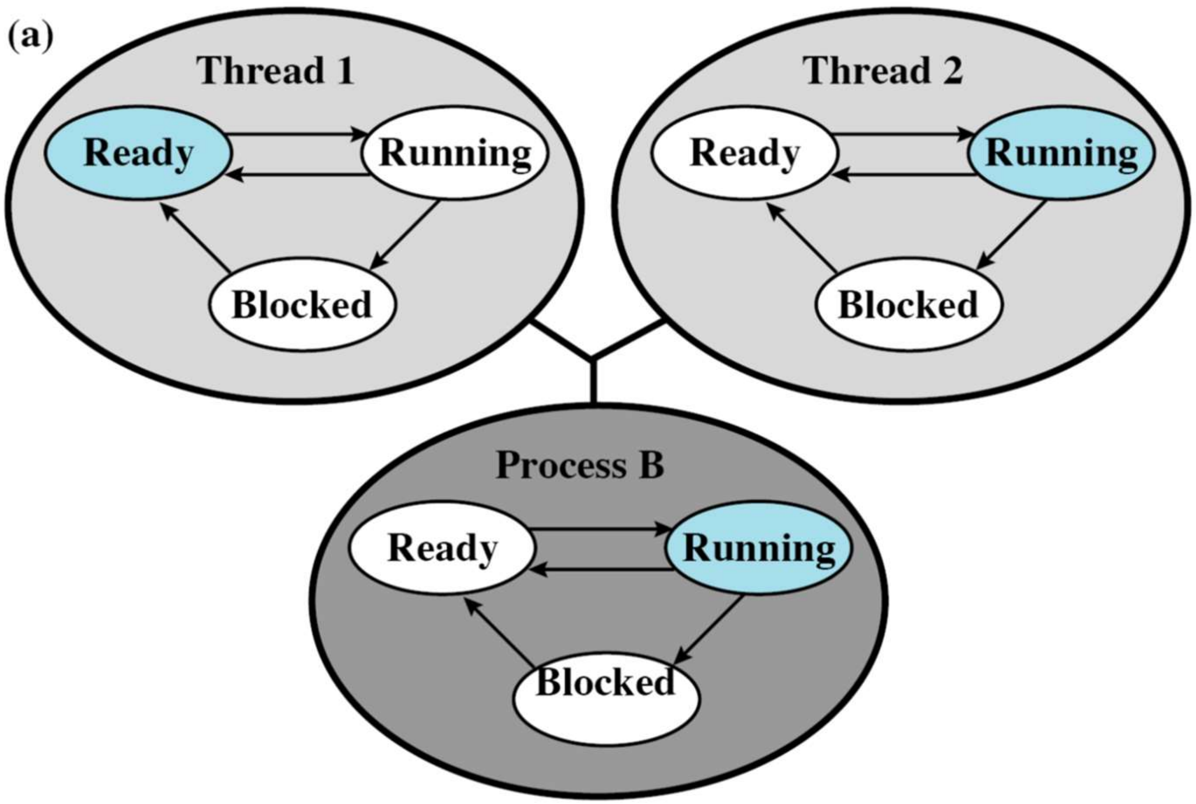
- Creating and destroying threads
- Passing messages and data between threads
- Scheduling thread execution
  - Pass control from one thread to another
- Saving and restoring thread contexts

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads



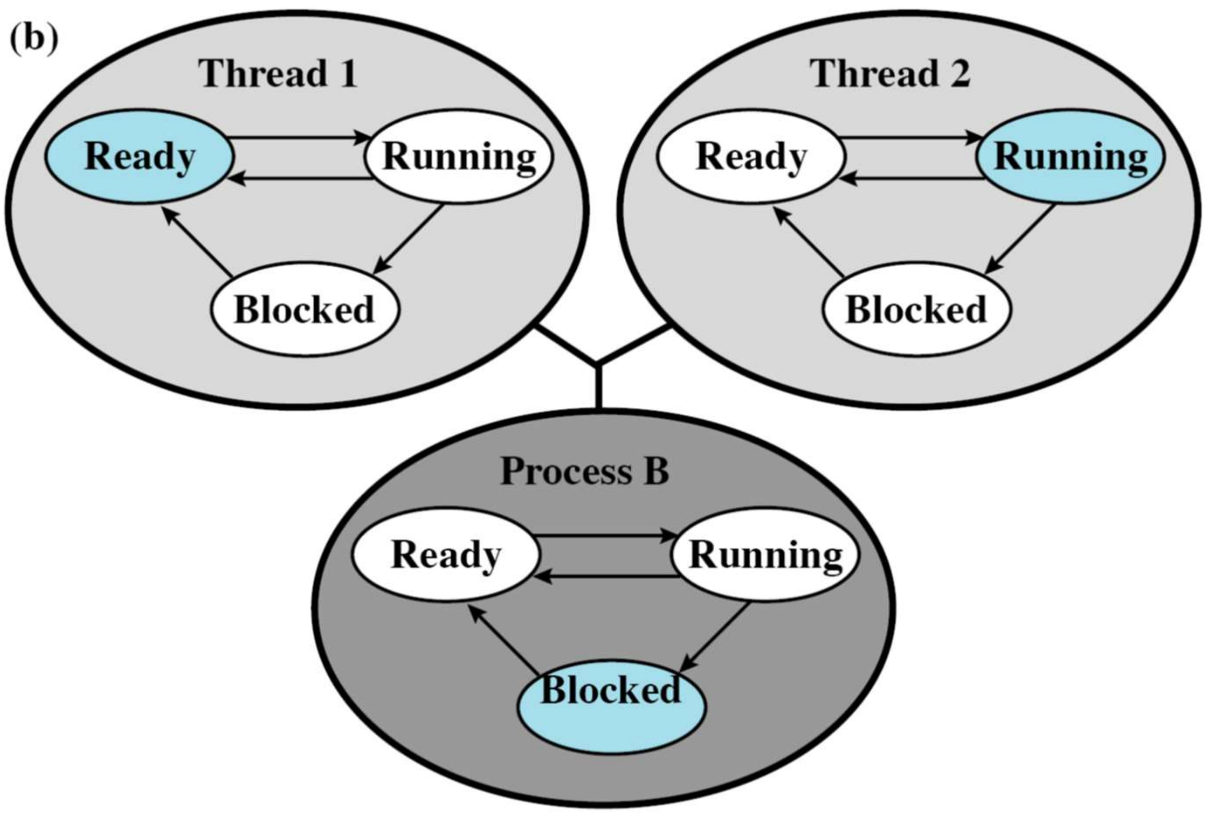
Example: User-Level Thread Execution

- Consider a thread makes a system call



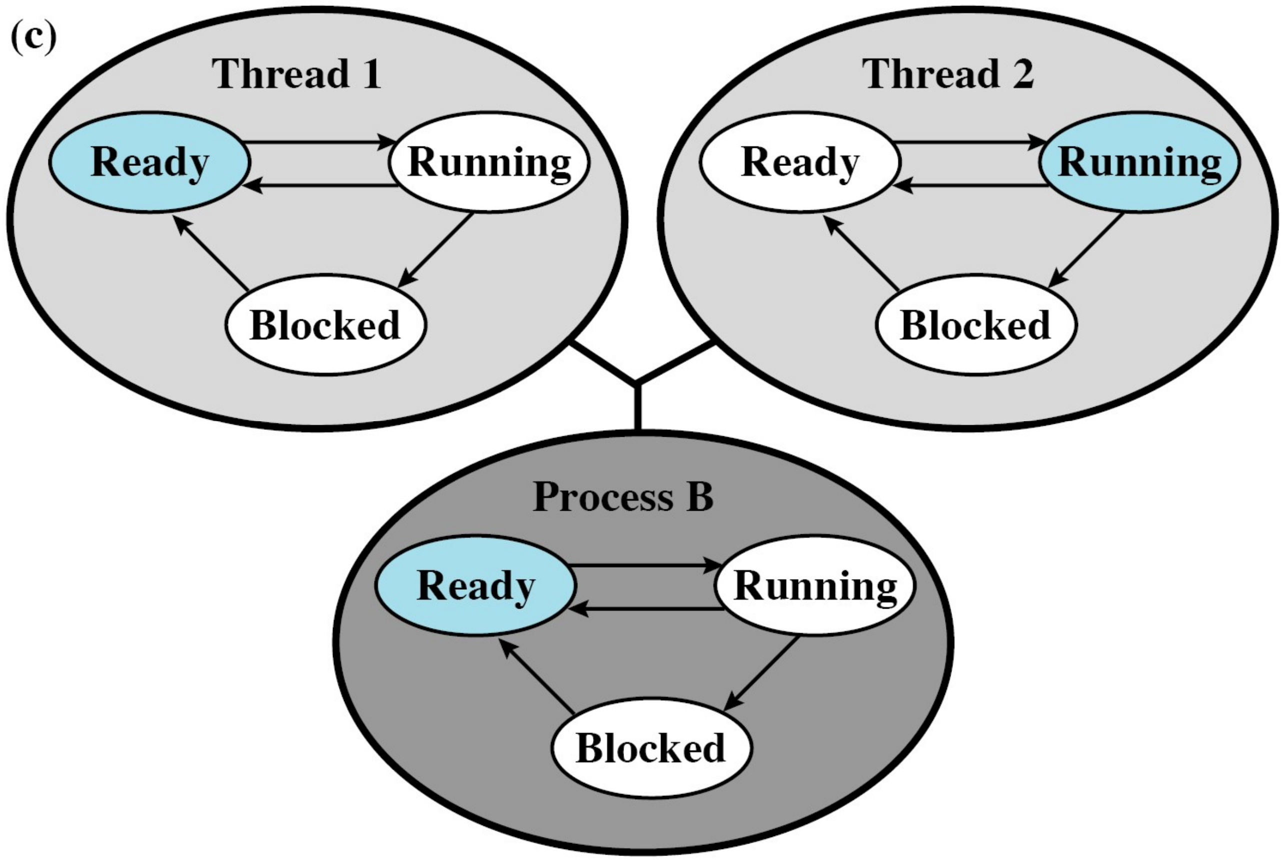
Example: User-Level Thread Execution

- Process switches to blocked state
- State of Thread 2 remains running, but Thread 2 cannot execute instructions



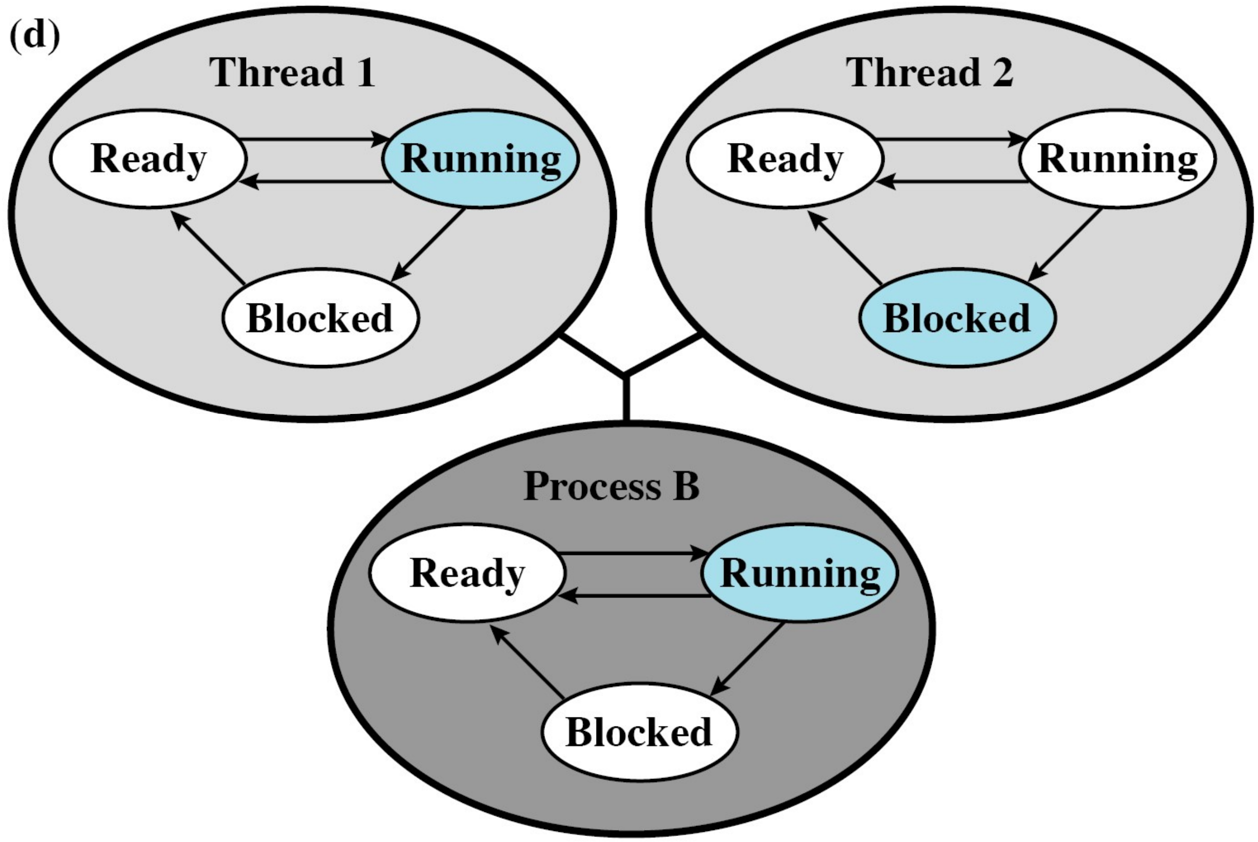
Example: User-Level Thread Execution

- Process switches to ready state
  - Returned from the system call



Example: User-Level Thread Execution

- Thread 2 needs to wait for some action of Thread 1



User-Level Thread

Advantages

- Fast
  - Do not need context switching to kernel mode to perform thread management
- Scheduling can be application specific
  - Choose the best algorithm for the situation
- Can run on any OS
  - Only a thread library is required

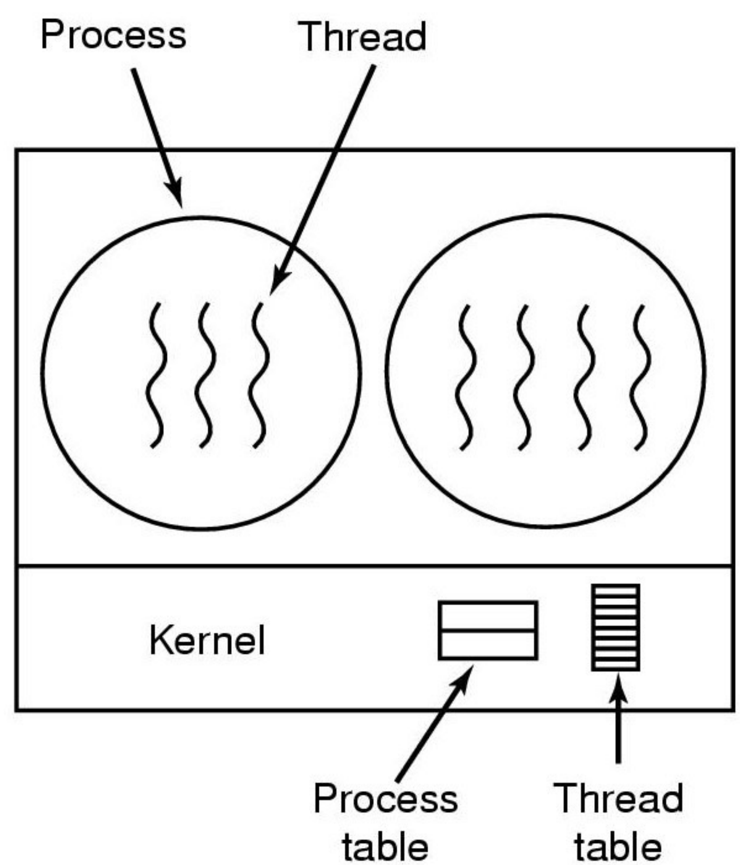
Disadvantages

- Most system calls are blocking for processes
  - All threads within a process will be implicitly blocked
- Cannot benefit from the multiprocessor system
  - Kernel assign one process to only one CPU

Implementing Threads in Kernel/Lightweight Process

- Scheduling
  - Happens on a thread basis
- Does not suffer from the "blocking problem"
- Threads of a process can be scheduled on multiple processors
- Less efficient than user-level threads
  - Kernel is invoked for thread creation,
  - Termination
  - Switching

Kernel maintains context information for the process and the threads



Kernel-Level Thread Support

- Windows XP/2000
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X

Kernel-Level Thread

Advantages

- In multiprocessor system, mutiple threads from the same process can be scheduled on multiple processors
- Blocking at thread level, not process level
  - If a thread blocks, the CPU can be assigned to another thread in the same process
- Even the kernel routines can be multithreaded

Disadvantages

- Thread switching always involves the kernel
  - Two context switches per thread switch
- Slower compared to User Level Threads
  - Faster than a full process switch

Some Performance Data

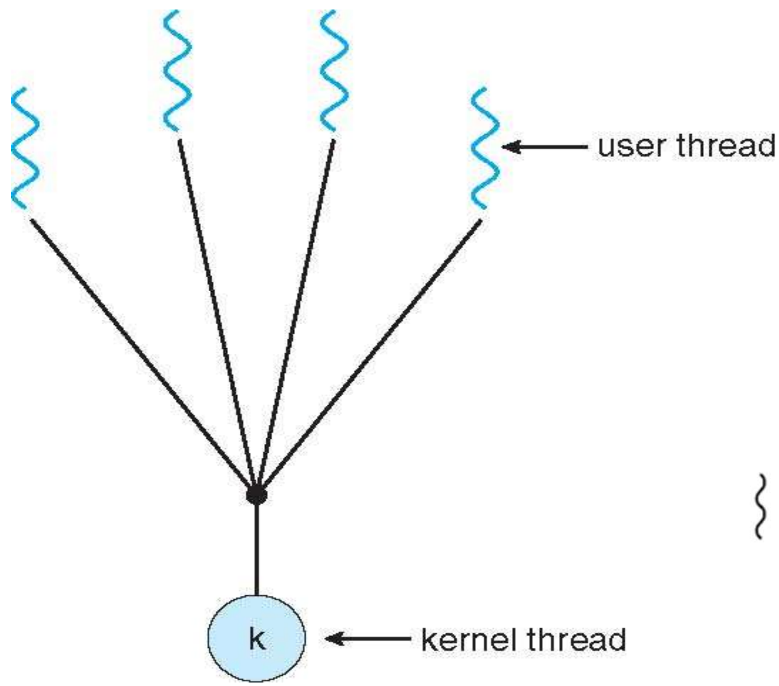
Table 4.1 Thread and Process Operation Latencies (μs) [ANDE92]

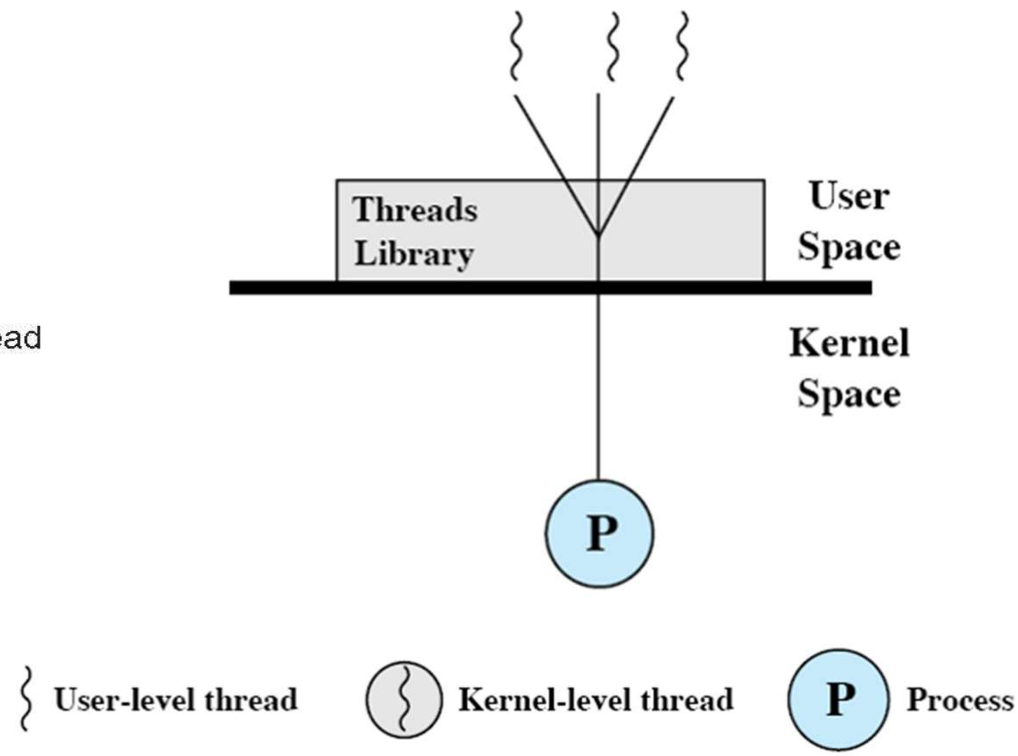
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

- Null Fork: Overhead of creating a process or thread
- Signal Wait: Overhead of synchronizing two processes or threads together

Multithreading Models: Many-to-one Model

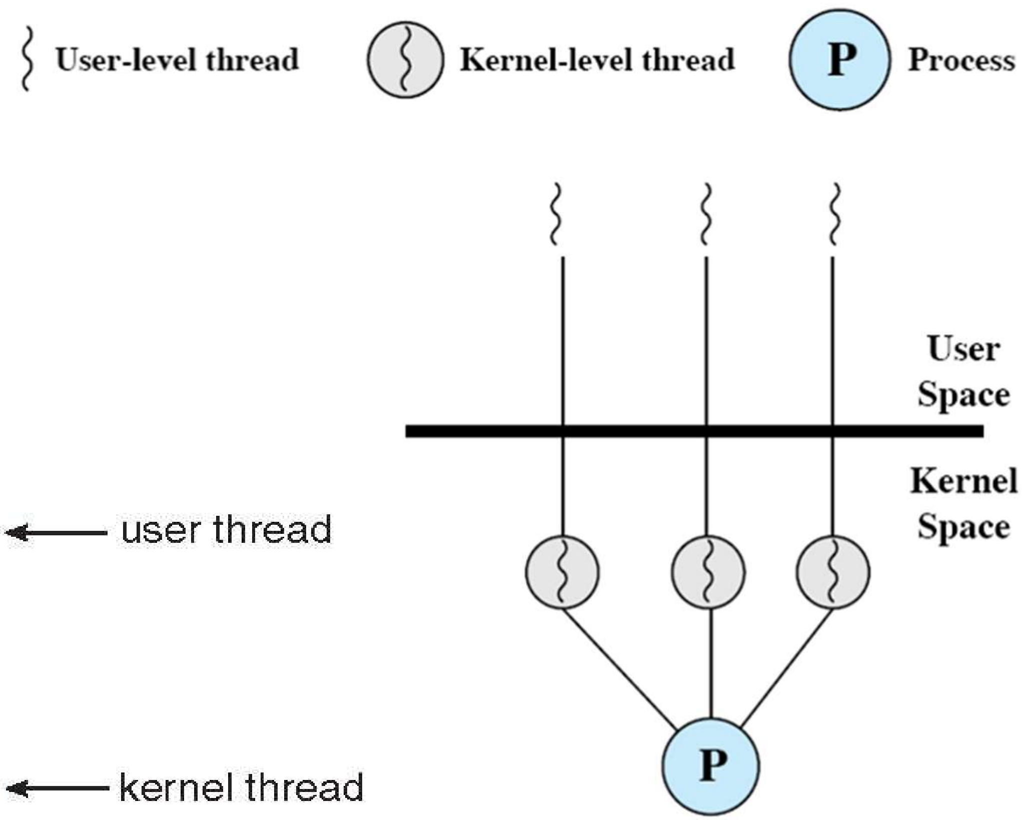
- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

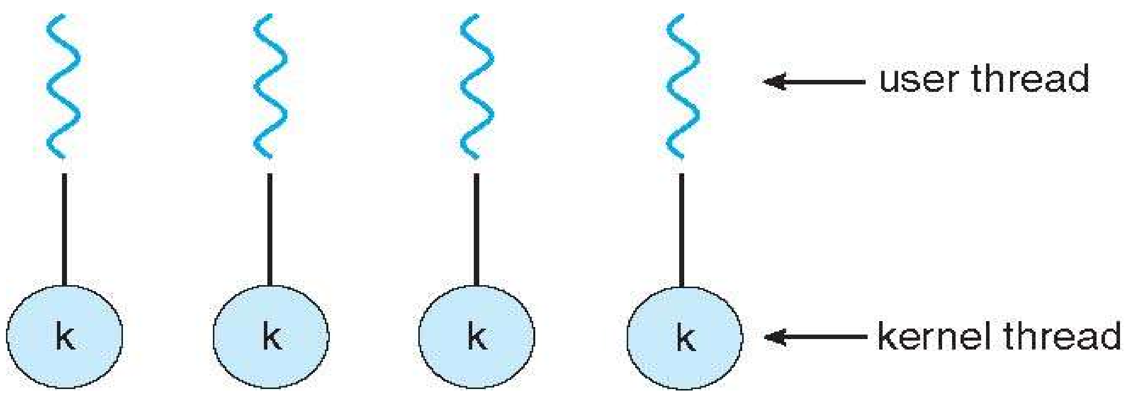




Multithreading Models: One-to-one Model

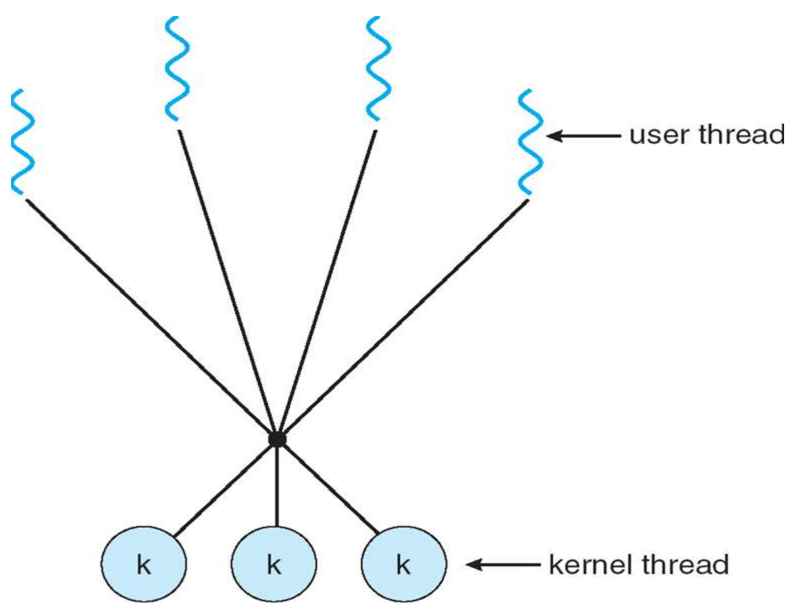
- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

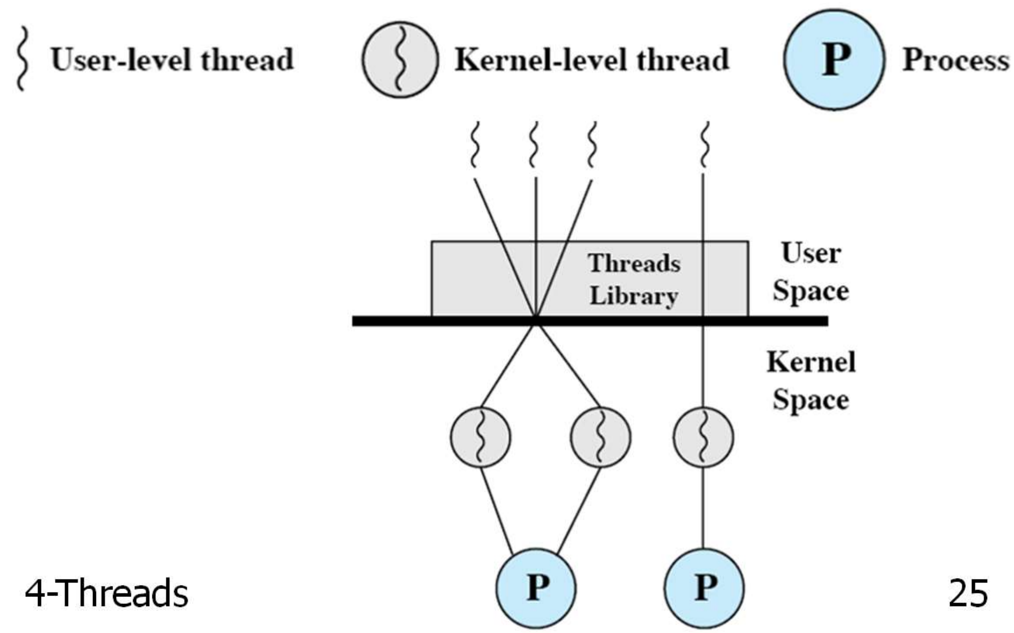




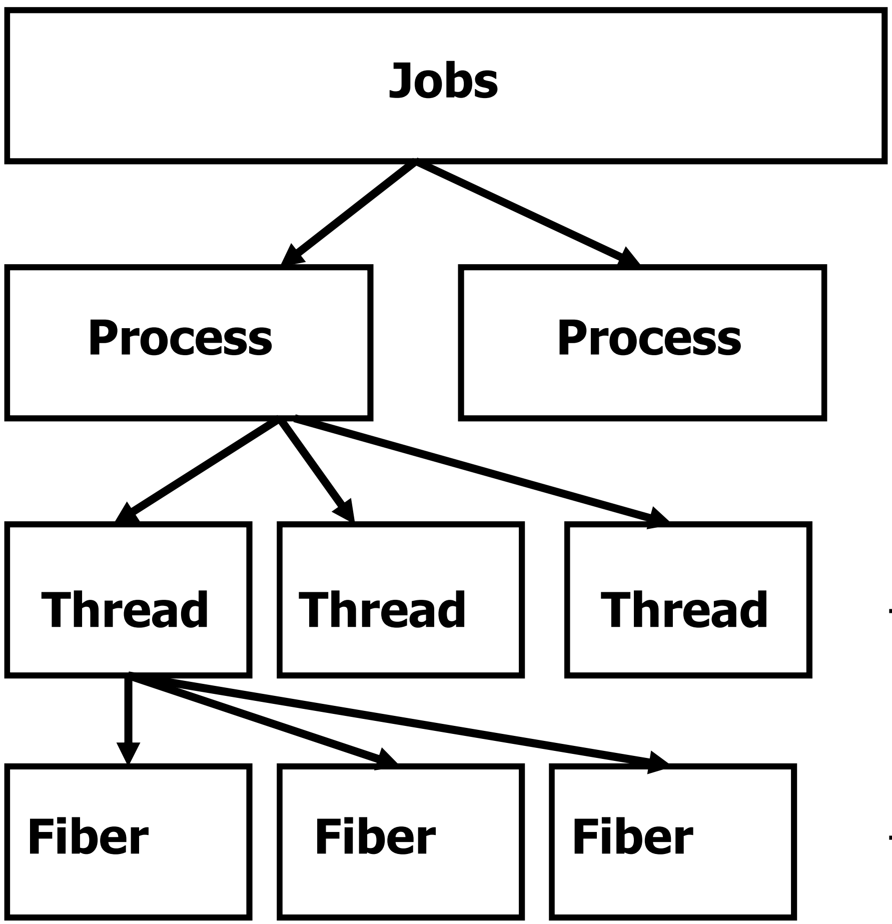
Multithreading Models: Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Example:
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package





Windows: Threads and Processes



Collection of processes

Kernel-level threads

User-level threads

WIN32 API

CreateProcess
ExitProcess

CreateThread
ExitThread

CreateFiber
ExitFiber
SwitchToFiber

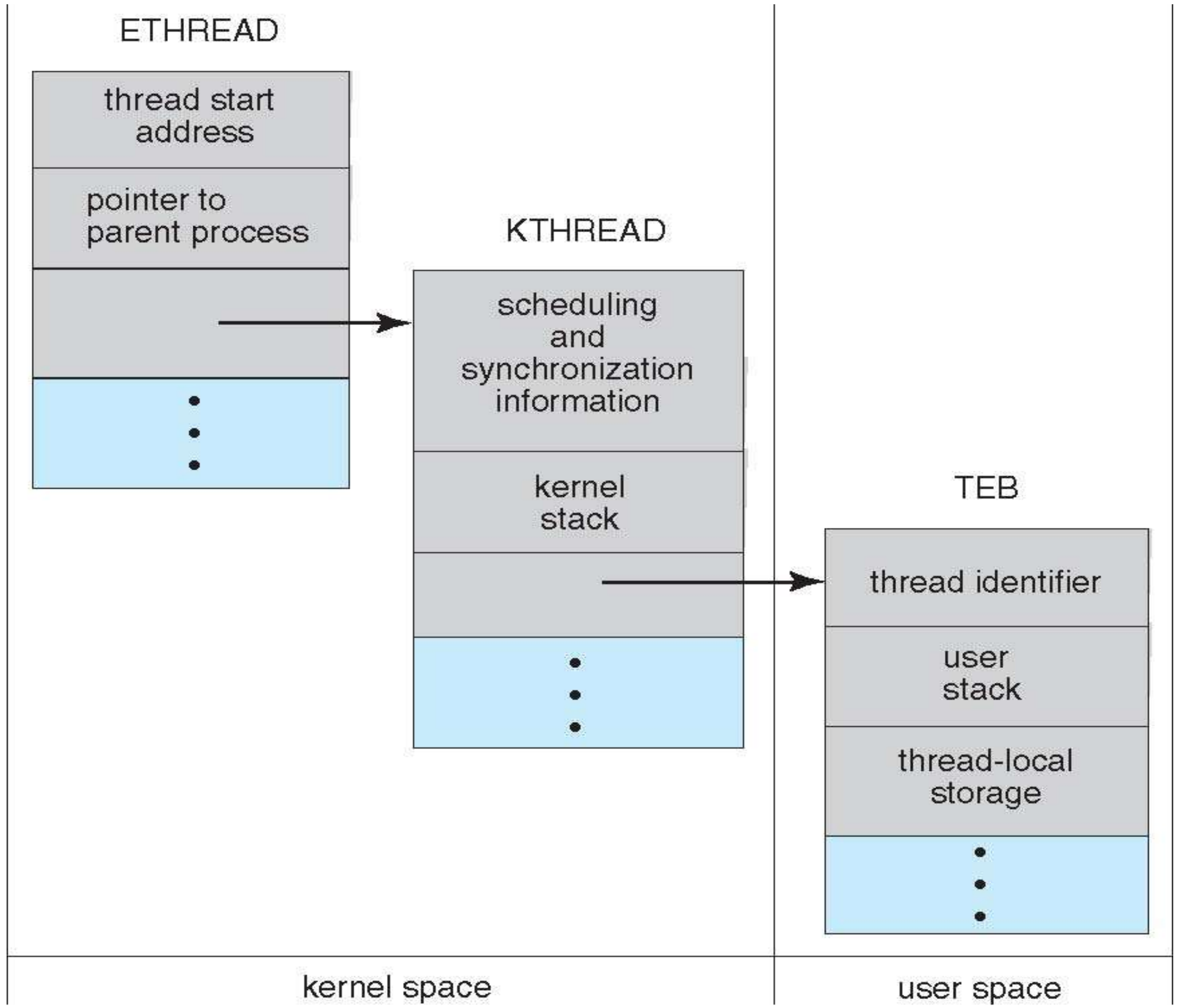
Windows XP: Threads and Processes

- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area

- The register set, stacks, and private storage area are known as the context of the threads

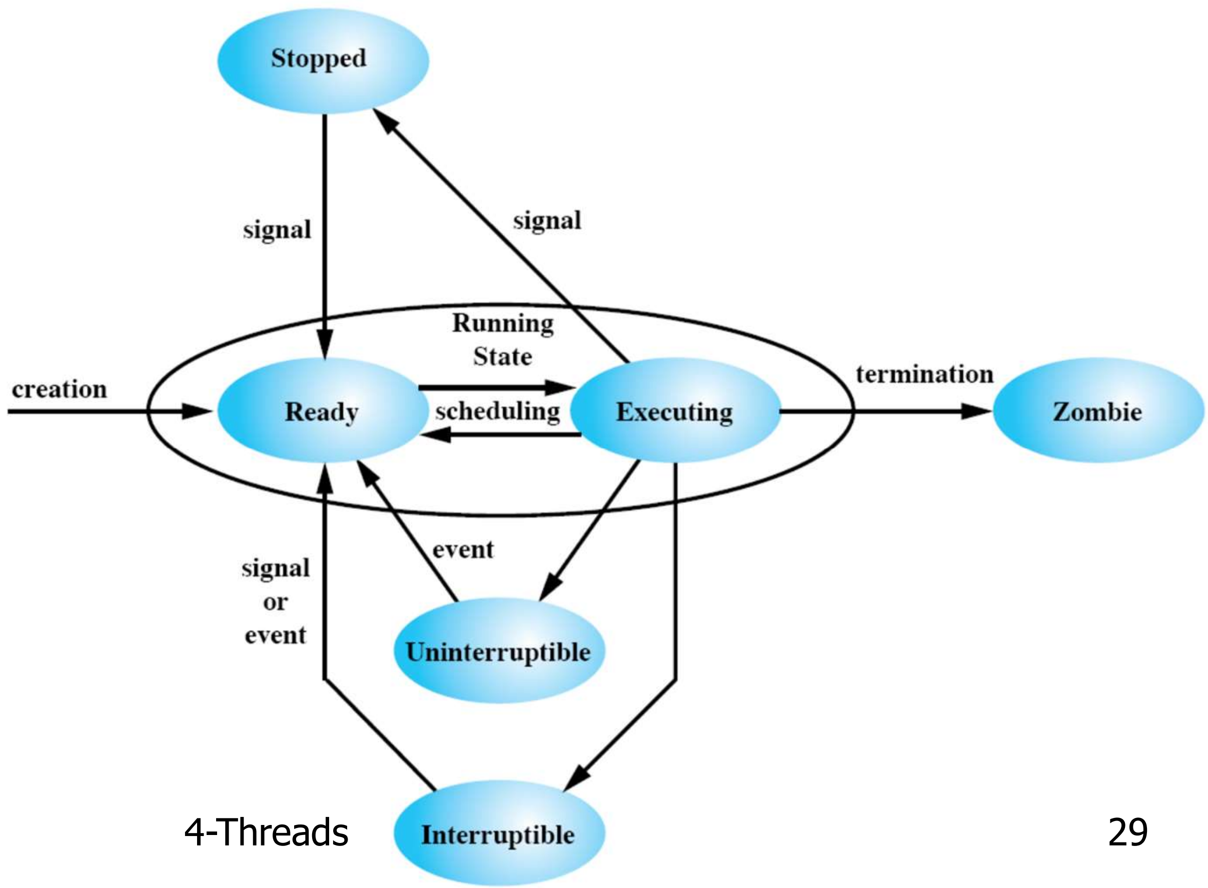
- The primary data structures of a thread include
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP: Threads and Processes



Linux Task Management

- Linux uses task as a placeholder for processes and threads
- Task creation
  - Specify resources to be inherited from the parent
    - ➢ Share file system information
    - ➢ File descriptors
    - ➢ Memory space
    - ➢ …



Linux Task Management

- `fork()` and `clone()` system calls
- `clone()` takes options to determine sharing between parent and child tasks

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- No sharing takes place, if none of above flags is set when `clone()` is invoked
  - Functionality similar to that provided by `fork()` system call

Linux Task Management

- Unique kernel data structure (`struct task_struct`) for each task
  - Instead of storing data, contains pointers to other data structures
  - For example, data structure that represent
    - open files
    - Signal handlers
    - Virtual memory
- When `fork()` is invoked
  - New task is created with copy of all associated data structures of the parent process (task)
- When `clone()` is invoked
  - New task points to the data structure of the parent process (task)
  - Depending on the set of flags passed to clone( )

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Thread Libraries

- POSIX threads (Pthreads)
  - Execution model for threads (specified by IEEE)
  - Specifies application programming interface
  - Implementation (using user-/kernel-level threads) is up to developers
  - More common in UNIX systems

- Win32 thread library
  - Kernel-level library, windows systems

- Java threads
  - Supported by the JVM
  - Java thread API is implemented using a thread library available on the host operating system, e.g., Pthreads, Win32

Pthreads API Overview

Thread creation:

```
int pthread_create (pthread_t *thread_id,
          const pthread_attr_t *attributes,
           void *(*thread_function)(void *),
           void *arguments);
```

Thread termination:

```
int pthread_exit (void *status);
int pthread_cancel (pthread_t thread):
```

Waiting for status of a thread:

```
int pthread_join (pthread_t thread,
                    void **status_ptr);
```

Modify thread attributes:

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

Thread Creation

```
int pthread_create (pthread_t *thread_id,
                    const pthread_attr_t *attributes,
                    void *(*thread_function)(void *),
                    void *arguments);
```

- This routine creates a new thread and makes it executable
- Thread stack is allocated and thread control block is created
- Once created, a thread may create other threads
  - When process starts an "initial thread" exists by default and is the thread which runs main
- Function `pthread_create` returns
  - zero, if success
  - Non-zero if error

Thread Creation

```
int pthread_create (pthread_t *thread_id,
                    const pthread_attr_t *attributes,
                    void *(*thread_function)(void *),
                    void *arguments);
```

`pthread_t *thread_id`

- Hold the identifier of the newly created thread
- Caller can use this thread ID to perform various operations

Thread Creation

```
int pthread_create (pthread_t *thread_id,
                    const pthread_attr_t *attributes,
                    void *(*thread_function)(void *),
                    void *arguments);
```

`pthread_attr_t *attributes`

- Used to set thread attributes
- To create thread with default attributes
  - Attribute object is not specified, i.e., NULL is used
- Default thread is created with following attributes
  - It is non-detached (i.e., joinable)
    - Detached: On termination all thread resources are released by OS
  - It has a default stack and stack size
  - It inherits the parent's priority

Thread Creation

```
int pthread_create (pthread_t *thread_id,
                    const pthread_attr_t *attributes,
                    void *(*thread_function)(void *),
                    void *arguments);
```

`void *(*thread_function)(void *)`

- The C routine that the thread will execute once it is created

Thread Creation

```
int pthread_create (pthread_t *thread_id,
                    const pthread_attr_t *attributes,
                    void *(*thread_function)(void *),
                    void *arguments);
```

`void *arguments`

- Arguments to be passed to `thread_function`
- Arguments must be passed by reference and cast to `void*`
- These pointers must be cast as pointers of type void

Thread Creation: Example

```
#include <pthread.h>
#define NUM_THREADS 5

void * PrintHello(void *threadid) {
    printf("\n%d: Hello World!\n", threadid);
}

int main() {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS; t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
}
```

Possible Output Sequence

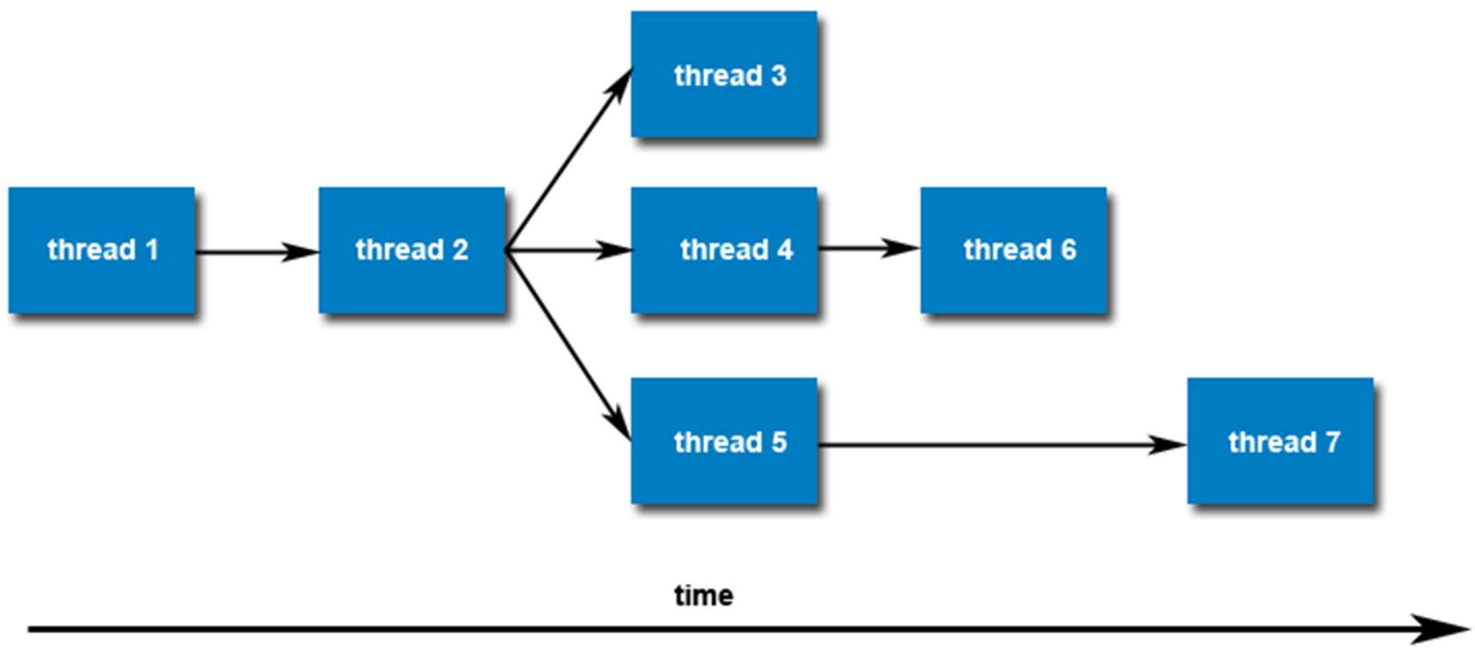
- No deterministic execution!
  - Thread execution can interleave in multiple ways

Possible Output

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

Thread Creation

- Once created, threads are peers, and may create other threads
  - No implied hierarchy or dependency between threads



Thread Termination

Several ways of termination

- The thread makes a call to the `pthread_exit()` routine
- The thread is canceled by another thread via `pthread_cancel()` routine
- The entire process is terminated due to a call to the `exit()`
- The thread returns from its starting routine/function
  - Same as if there was an implicit call to `pthread_exit()` using the return value of thread function as the exit status
  - Behavior of thread executing `main()` differs
    - Effect: Implicit call to `exit()` using the return value of `main()` as the exit status

Thread Termination

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

- The `pthread_exit()` routine does not close files
  - Any files opened inside the thread will remain open after the thread is terminated
- If the "initial thread" exits with `pthread_exit()` instead of `exit()`, other threads will continue to execute
- The programmer may specify a termination status
  - Stored as a void pointer for any thread that may join the calling thread i.e., wait for this thread

Thread Termination: Example

```
#include <pthread.h>
#define NUM_THREADS 5

void * PrintHello(void *threadid) {
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}
int main() {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS; t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Passing Arguments to Threads

- The `pthread_create` permits the programmer to pass one argument to the thread start function
  - Argument must be passed by reference and cast to `(void *)`
- What if we want to pass multiple arguments?
  - Create a structure which contains all of the arguments
  - Pass a pointer to the structure in the `pthread_create`
- Once created threads have non-deterministic start-up & scheduling
  - How to safely pass data/argument to newly created threads?
  - Make sure that all passed data is thread safe
    - i.e., it cannot be changed by other threads
  - The calling function must ensure that argument remains valid for the new thread throughout its lifetime
    - At least until thread has finished accessing it

Passing Arguments to Threads

- Incorrectly passed arguments

```
for(t=0;t < NUM_THREADS;t++) {
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, printHello,(void *) &t);
   ...
 }
```

- Correctly passed arguments

```
int *tids[NUM_THREADS];
for(t=0;t < NUM_THREADS;t++) {
   tids[t] = new int;
   *tids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t],NULL,PrintHello,(void*)tid[t]);
   ...
}
```

Passing Structure as Argument

```
struct thread_data {
   int thread_id;
   int sum;
};
thread_data thread_data_array[NUM_THREADS];
int main() { ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   rc = pthread_create(&threads[t], NULL, PrintHello,(void )&thread_data_array[t]) ;
   ...
}
void *PrintHello(void *threadarg) {
   thread_data *my_data;
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   ...
}
```

Thread Identifier

```
pthread_t pthread_self(void);
```

- Returns the unique thread ID of the calling thread

```
int pthread_equal(pthread_t t1, pthread_t t2)
```

- Compares two thread IDs
  - If the two IDs are different 0 is returned
  - Otherwise a non-zero value is returned

Changing Thread Attributes

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

To change the (default) attributes of the thread, the following steps can be performed

- Initialize attributes object with `pthread_attr_init()` function
- Afterwards, individual attributes of the object can be set using various related functions
  - `pthread_attr_setdetachstate()` to set the detached state of thread
  - `pthread_attr_setstacksize()` function sets the stack size
- When a thread attributes object is no longer required, free library resources using the `pthread_attr_destroy()` function

Changing Thread Attributes: Example

```
#define NUM_THREADS 5
#define MEGEXTRA 1000000
#define N 1000

Pthread_attr_t attr;
int main() {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   size_t stacksize
   pthread_attr_init( &attr);
   pthread_attr_getstacksize(&attr, &stacksize);
   printf("Default stack size - %li\n", stacksize);
   stacksize = sizeof(double) * N * N + MEGEXTRA;
   pthread_attr_setstacksize (&attr, stacksize);
   for(t=0;t < NUM_THREADS; t++) {
      printf("Creating thread with stacksize - %li\n", stacksize);
      rc = pthread_create(&threads[t], &attr, PrintHello, (void *)&t);
       . . .
   }
   pthread_attr_destroy(&attr):
. . .
```

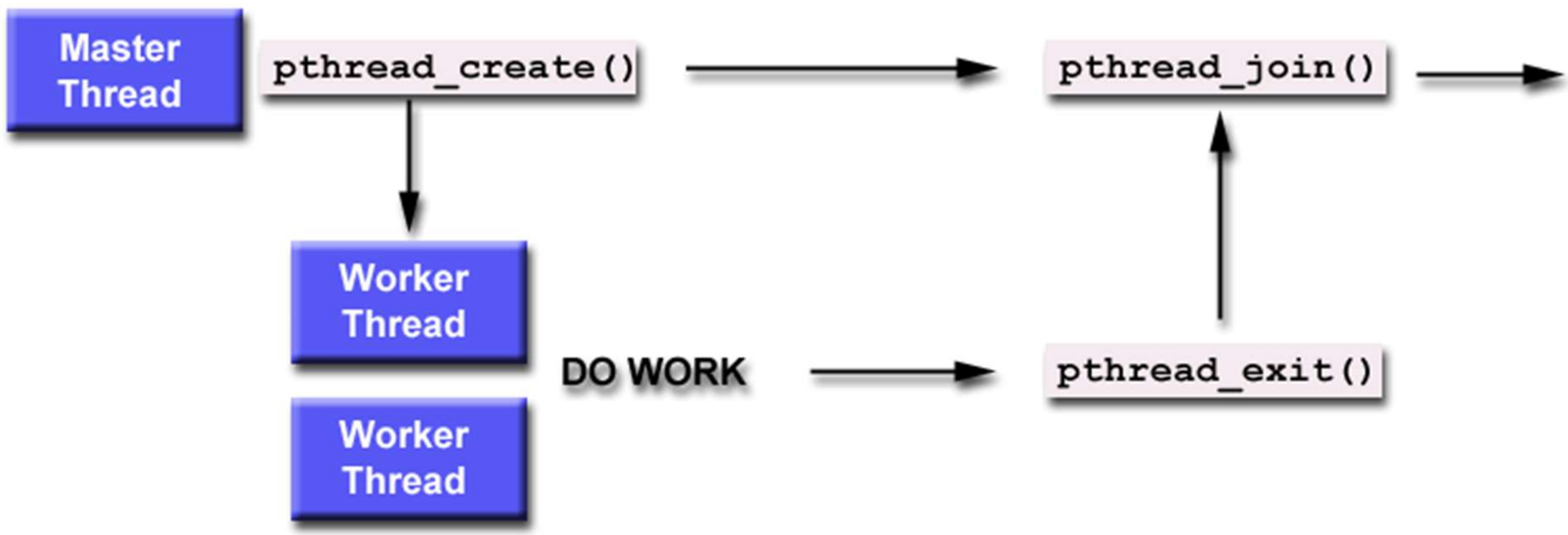
Thread Suspension and Termination

- Similar to UNIX processes, threads have the equivalent of the `wait()` and `exit()` system calls

```
int pthread_join (pthread_t thread,
                  void **status_ptr);
```

- Use to block threads
- To instruct a thread to block and wait for a thread to complete
- Any thread can call join on (and hence wait for) any other thread
- Serves as a mechanism to get a return value from a thread

Thread Suspension and Termination



Joinable Threads

- When a thread is created, one of its attributes defines whether it is joinable or detached
  - Only threads that are created as joinable can be joined
  - If a thread is created as detached, it can never be joined

- Detached Thread
  - On termination all thread resources are released by the OS
  - A detached thread cannot be joined
  - No way to get at the return value of the thread

- Joinable Thread
  - On thread termination thread ID and exit status are saved by the OS
  - Joining a thread means waiting for a thread with a certain ID
    - ➢ One way to accomplish synchronization between threads

Joinable Threads

- Multiple threads cannot wait for the same thread to terminate
- If multiple thread simultaneously try to join, the results are undefined
  - For example, only one thread returns successfully
  - The other threads fail with an error of ESRCH or EINVAL
- After `pthread_join()` returns, any stack storage associated with the thread can be reclaimed by the process
- Threads which have exited but have not been joined are equivalent to zombie processes
  - Their resources cannot be fully recovered

Joinable Threads: Example

```
#include <stdio.h>
#include <pthread.h>

int x = 5;

int child_code()
{
  while (x>0){
    x=x-1;
    printf("x has changed to %d\”
          ,x);
  }
  return 0;
}

int main(int argc, char *argv[])
{
  int rc;
  void** status=0;
  pthread_t child_thread;
  pthread_attr_t attr;

  /* Set threads properties */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
   PTHREAD_CREATE_JOINABLE);

  /* Create new thread */
  rc = pthread_create(&child_thread,
   &attr, (void*) &child_code, NULL);
  /* Free attributes object */
  pthread_attr_destroy(&attr);

  /* concurrent computation */
  x=x+5;
  printf("x has change to %d\n",x);

  /*wait for thread to terminate */
  pthread_join(child_thread, status);
  pthread_exit(NULL);
}
```

Possible Outcome

No deterministic execution!

- Two threads can interleave in multiple ways
- Requires further synchronization

Child: x=5

```
while (x>0){
  x=x-1;
  printf("x has changed to %d \n",x);
}
```

Parent: x=3

```
x=x+5;
printf("x has changed to %d\n",x);
```

Child: x=3

```
while (x>0){
  x=x-1;
  printf("x has changed to %d \n",x);
}
```

- x has changed to 4
- x has changed to 3

- x has changed to 8
- x has changed to 7

…

- x has changed to 1
- x has changed to 0

Joinable Threads: Example

```c
#include <pthread.h>
#define NUM_THREADS 4

void *BusyWork(void *t){
  long tid = (long)t;
  double result=0.0;
  printf("Thread %ld starting...\n",tid);
  for (int i=0; i<1000000; i++){
    result = result + sin(i) * tan(i);
  }
  printf("Thread %ld done.
          Result = %e\n",tid, result);
  pthread_exit((void*) t);
}
int main (int argc, char *argv[]) {
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc;
  long t;
  void *status;
  /* Initialize & set thread attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
    PTHREAD_CREATE_JOINABLE);
  for(t=0; t<NUM_THREADS; t++) {
    printf("Main:creating thread %ld\n", t);
    pthread_create(&thread[t], &attr,
         BusyWork, (void *)t);
  }
  /* Free attribute & wait for threads */
  pthread_attr_destroy(&attr);
  for(t=0; t<NUM_THREADS; t++) {
    pthread_join(thread[t], &status);
    printf("Main: completed join with thread
            %ld with status %ld\n", t,
            (long)status);
  }
  printf("Main: program completed. \n");
  pthread_exit(NULL);
}
```

Possible Output

Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 1 done. Result = -3.153838e+06
Thread 0 done. Result = -3.153838e+06
Main: completed join with thread 0 with status of 0
Main: completed join with thread 1 with status of 1
Thread 3 done. Result = -3.153838e+06
Thread 2 done. Result = -3.153838e+06
Main: completed join with thread 2 with status of 2
Main: completed join with thread 3 with status of 3
Main: program completed. Exiting.

Making a Thread Detached

```
int pthread_detach (pthread_t thread_id);
```

- Function marks the thread identified by `thread_id` as detached
  - Return 0 on success

- To avoid memory leaks a thread should
  - Either be joined
  - Or detached by a call to `pthread_detach()`

- Once a thread is detached using this function, it cannot be make joinable again

- Threads can detach themselves by calling `pthread_detach()` with an argument of `pthread_self( )`
  - i.e., `pthread_detach( pthread_self())`

Any Question So Far?