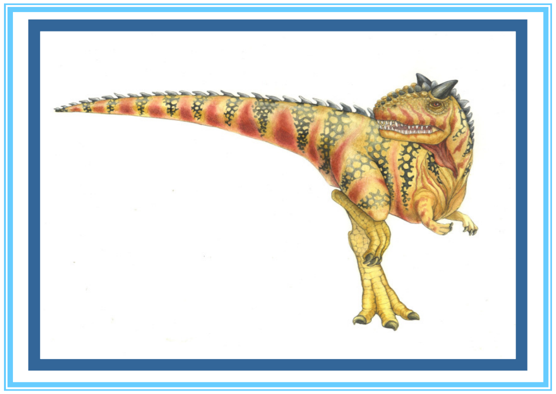
Lecture 6,7 : Thread

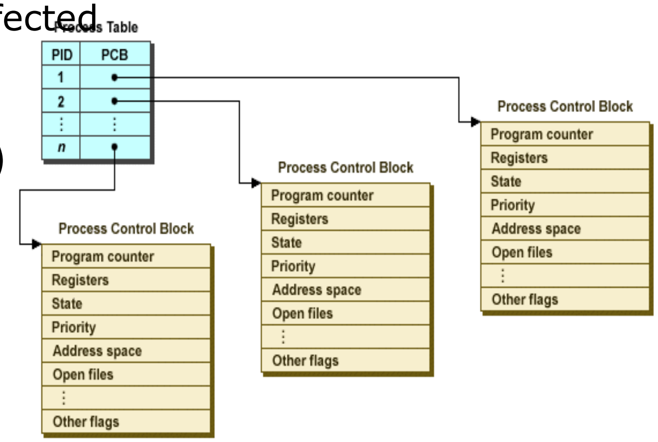
From Processes to Threads



The Soul of a Process

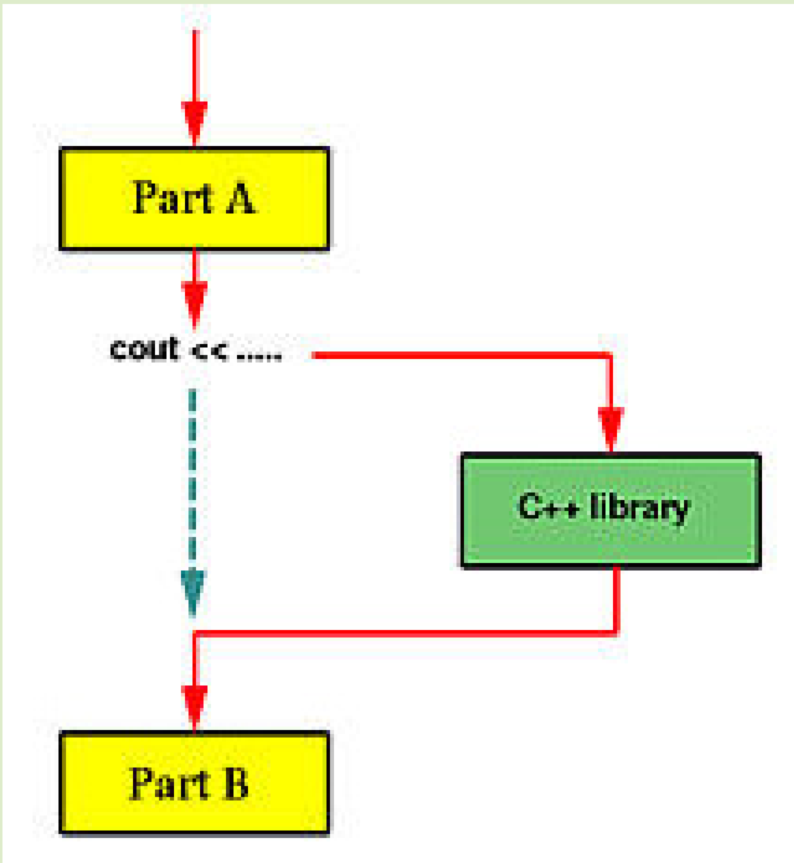
Shared data, has IPC, execution been affected

- What is similar in cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)

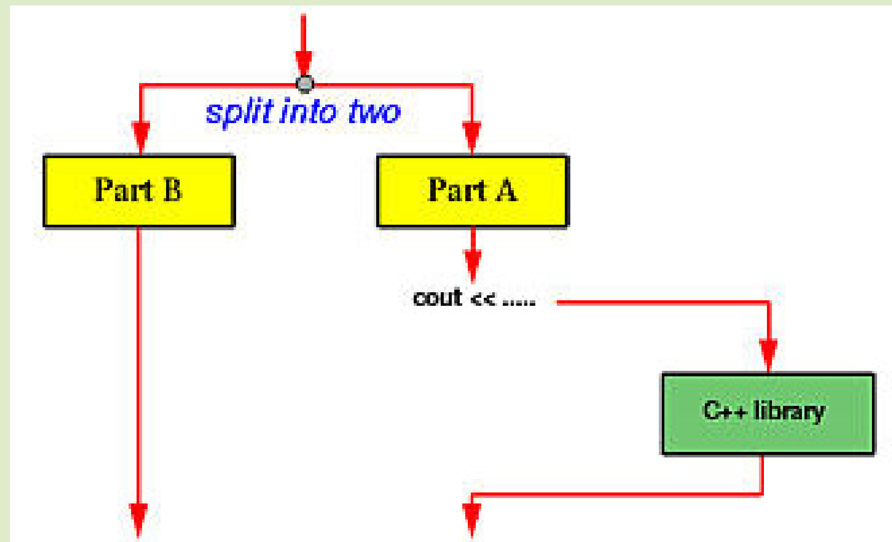


- What don't they share?
  - Each has its own execution state: PC, SP, and registers
  - Key idea: Why don't we separate the concept of a process from its execution state?
    - Process: address space, privileges, resources, etc.
    - Execution state: PC, SP, registers
  - Exec state also called thread of control, or thread

Why Threads?



Part A
cout << .....
C++ library
Part B



split into two
Part B
Part A
cout << .....
C++ library

Processes and Threads

- Process abstraction combines with two concepts
  - Concurrency
    - each process is a - sequential execution stream of instructions
  - Protection
    - each process defines an address space
    - address space identifies all addresses that can be touched by the program itself

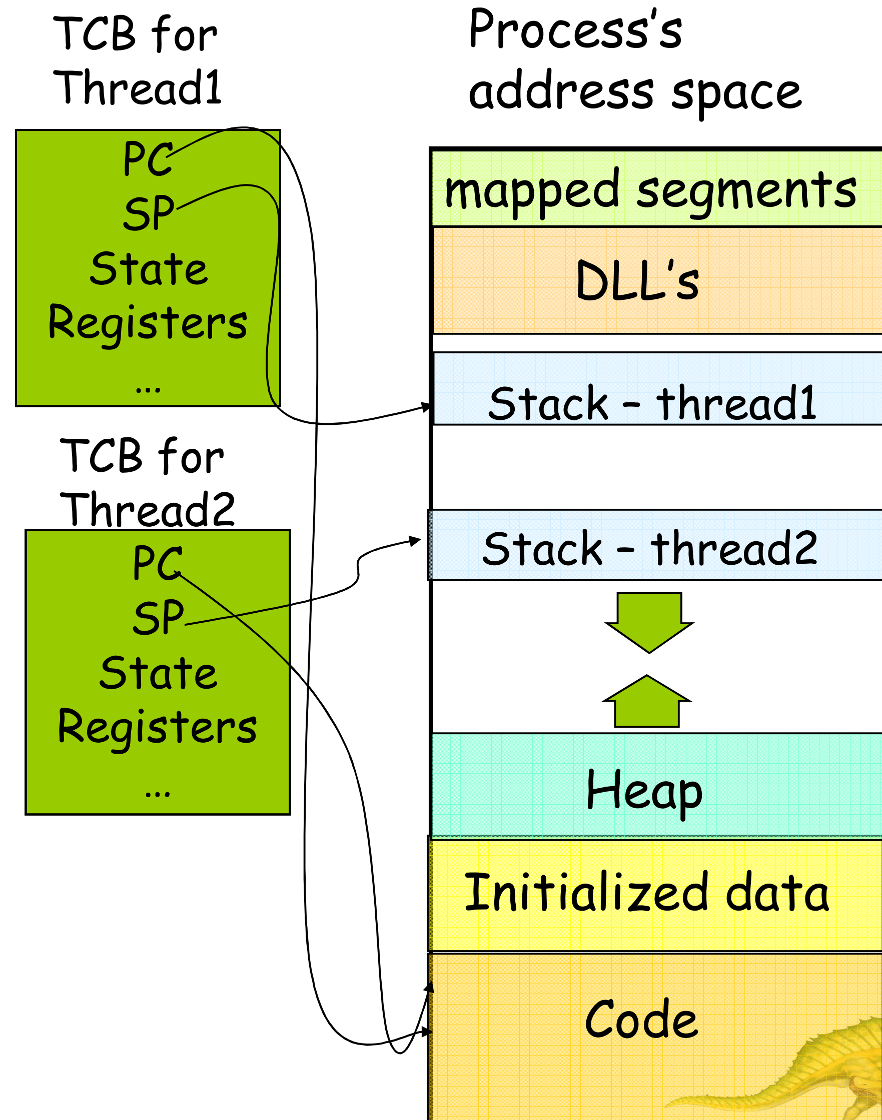
- Threads
  - Key idea: separate the concepts of concurrency from protection
  - A thread is a sequential execution stream of instructions
    - shared - process address space
  - Threads can execute on different cores on a multicore CPU
    - parallelism for performance and can communicate with other threads by sharing memory

Shared –map memory address space

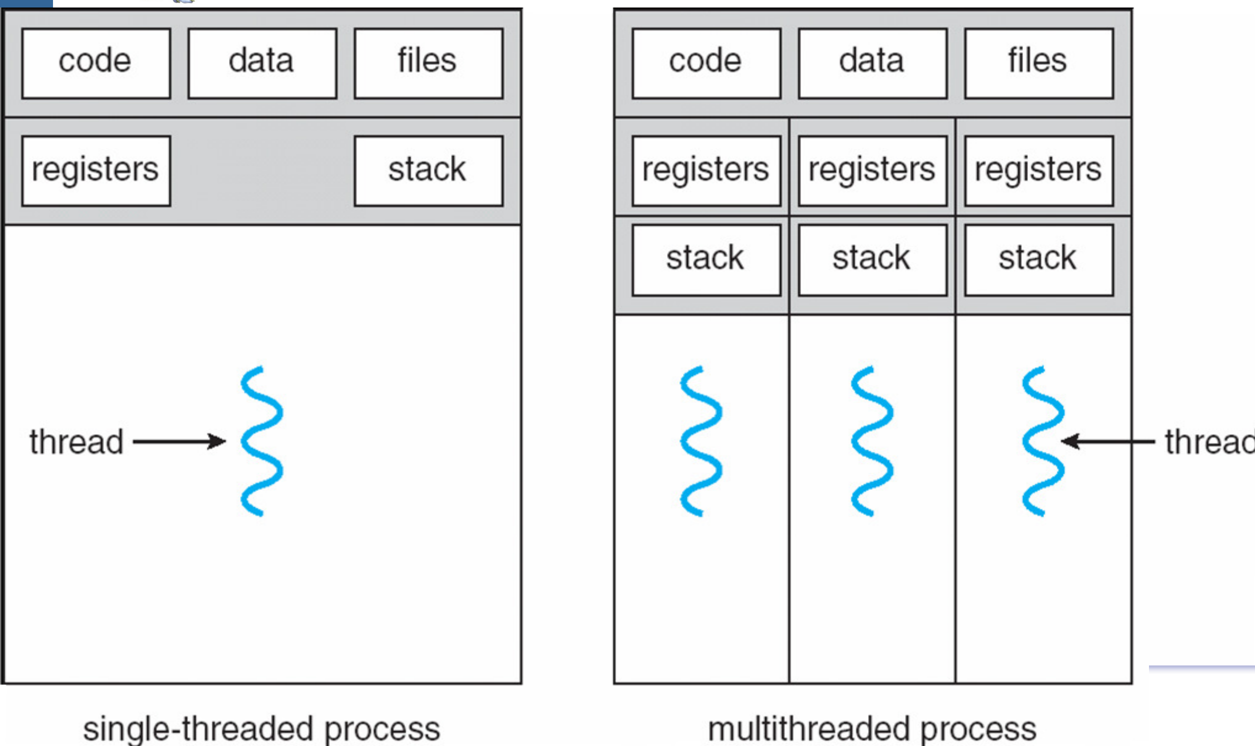
Code execution different

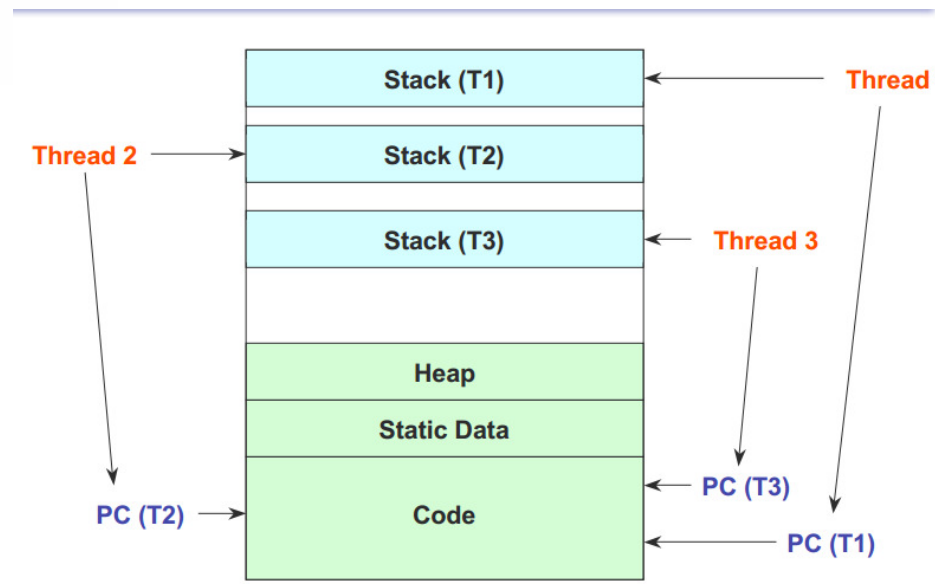
Processes and Threads

- Processes define an address space; threads share the address space
- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, …), register values, a pointer to PCB, …



Single and Multithreaded Processes





The Case for Threads

Consider the following code fragment

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

Is there a missed opportunity here? On a Uni-processor?

On a Multi-processor?

The Case for Threads

Consider a Web server

- get network message (URL) from client
- get URL data from disk
- compose response
- send response

How well does this web server perform?

Introducing Threads

- A thread represents an abstract entity that executes a sequence of instructions
  - It has its own set of CPU registers
  - It has its own stack
  - There is no thread-specific heap or data segment (unlike process)
- Threads are lightweight
  - Creating a thread more efficient than creating a process.
  - Communication between threads easier than processes.
  - Context switching between threads requires fewer CPU cycles and memory references than switching processes.
  - Threads only track a subset of process state (share list of open files, pid, …)
- Examples:
  - OS-supported: Windows' threads, Sun's LWP, POSIX threads
  - Language-supported: Modula-3, Java

Context switch time for which entity is greater?

1. Process
2. Thread

How Can it Help?

- How can this code take advantage of 2 threads?

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

- Rewrite this code fragment as:

```
do_mult(l, m) {
    for(k = l; k < m; k++)
            a[k] = b[k] * c[k] + d[k] * e[k];
}
main() {
  CreateThread(do_mult, 0, n/2);
  CreateThread(do_mult, n/2, n);
```

- What did we gain?

Multithreads vs processor

	T1	C1+I1	TT=C1+I1+C2	Case1: Simple
P1			TT=C1+ Max (I1, C2)	Case2: Multithreaded process@uniprocessor
	T2	C2	TT=Max(C1+I1, C2)	Case2: Multithreaded process@multiprocessors

How Can it Help?

- Consider a Web server

  Create a number of threads, and for each thread do

  - get network message from client
  - get URL data from disk
  - send data over network

- What did we gain?

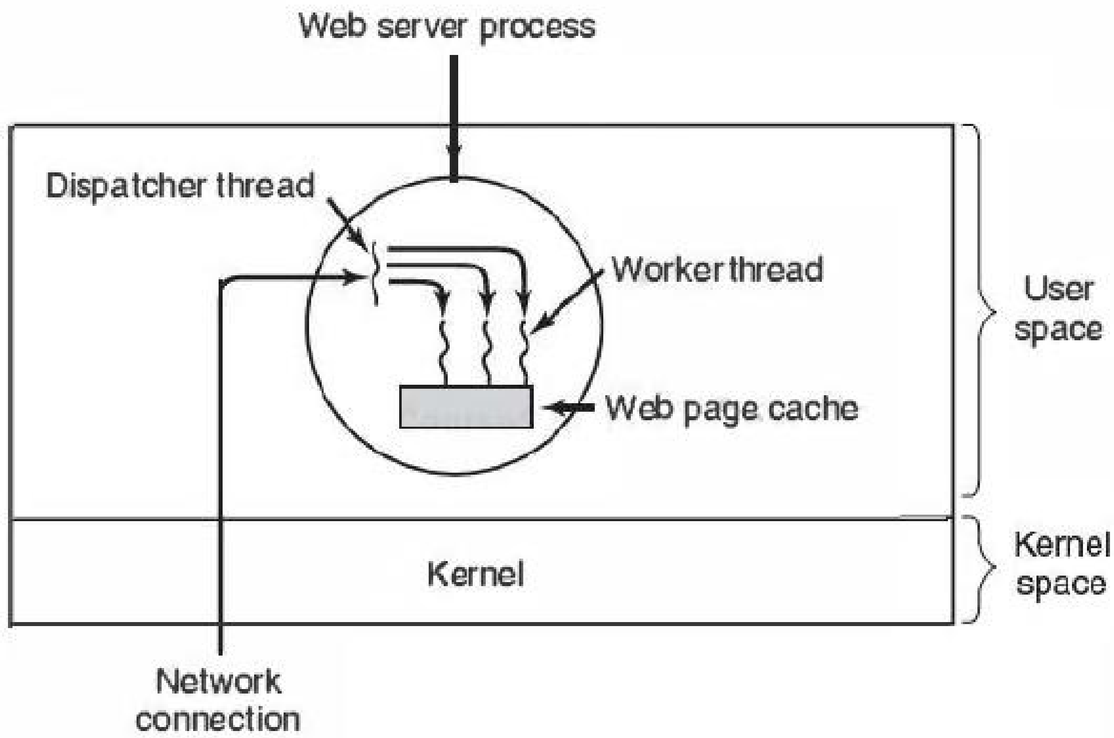


Figure 2. A multithreaded Web server

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Thread Pool

Figure 3. A rough outline of the code for Fig. 2. (a) Dispatcher thread. (b) Worker thread

Overlapping Requests (Concurrency)

Request 1
Thread 1

- get network message (URL) from client
- get URL data from disk

(disk access latency)

- send data over network

Request 2
Thread 2

- get network message (URL) from client
- get URL data from disk

(disk access latency)

- send data over network

Time

- Total time is less than request 1 + request 2

Threads vs. Processes

Threads

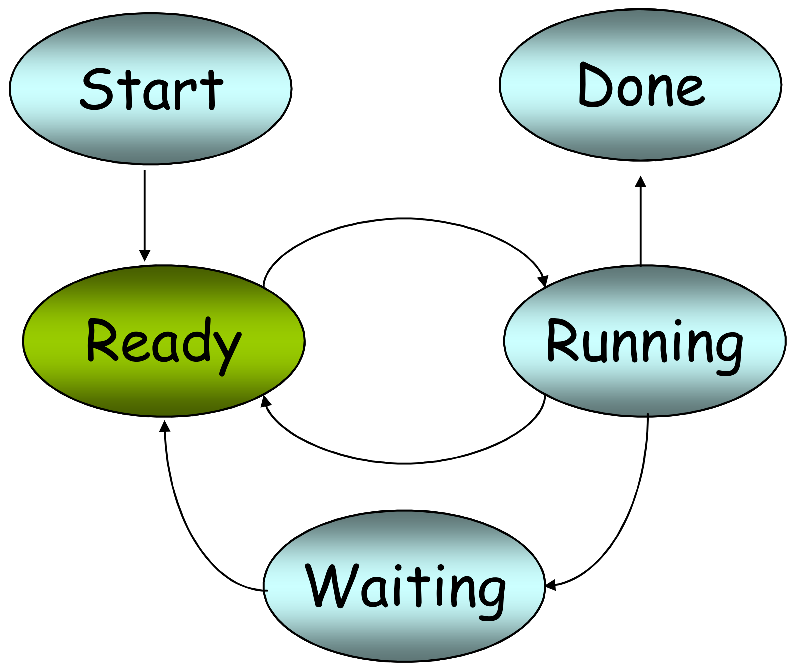
- A thread has no separate data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one threads in a process, the first thread calls main & has the process's stack
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory.
- Each thread can run on a physical processor
- Inexpensive creation and context switch

Processes

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- If a process dies, its resources are reclaimed & all threads die
- Inter-process communication via OS /data copying/message passing.
- Each process can run on a physical processor
- Expensive creation and context switch

Threads' Life Cycle

- Threads (just like processes) go through a sequence of *start*, *ready*, *running*, *waiting*, and *done* states



Threads have the same scheduling states as processes

1. True
2. False

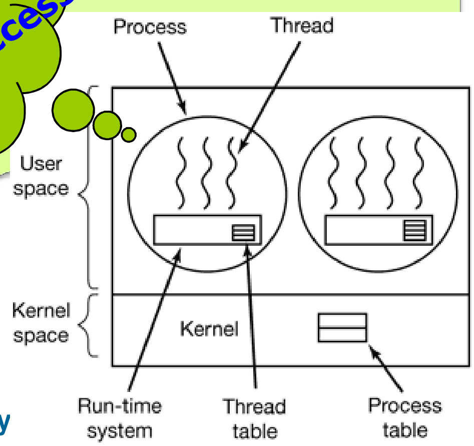
Implementing Threads

POSIX Pthreads

User Level

- Threads package entirely in user space
- Kernel knows nothing about threads
- Fast to create and switch-scheduler as local procedure

· Blocking System Call
· CPU access



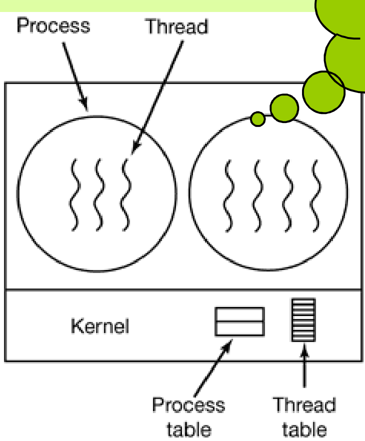
Windows NT
Windows 2000

Kernel Level

- No runtime system
- Global Thread table, updated by kernel call
- Do not block process for systemcall
- Thread switching: same or another process
- Not so fast as runtime system

Thread Recycling
Thread pool

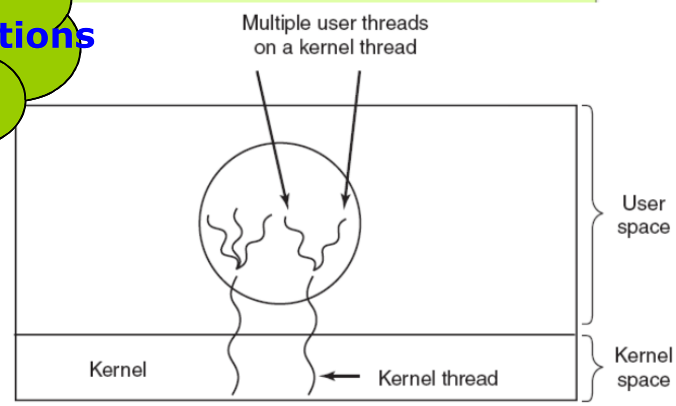
Frequent thread operations
Need many System calls



Solaris

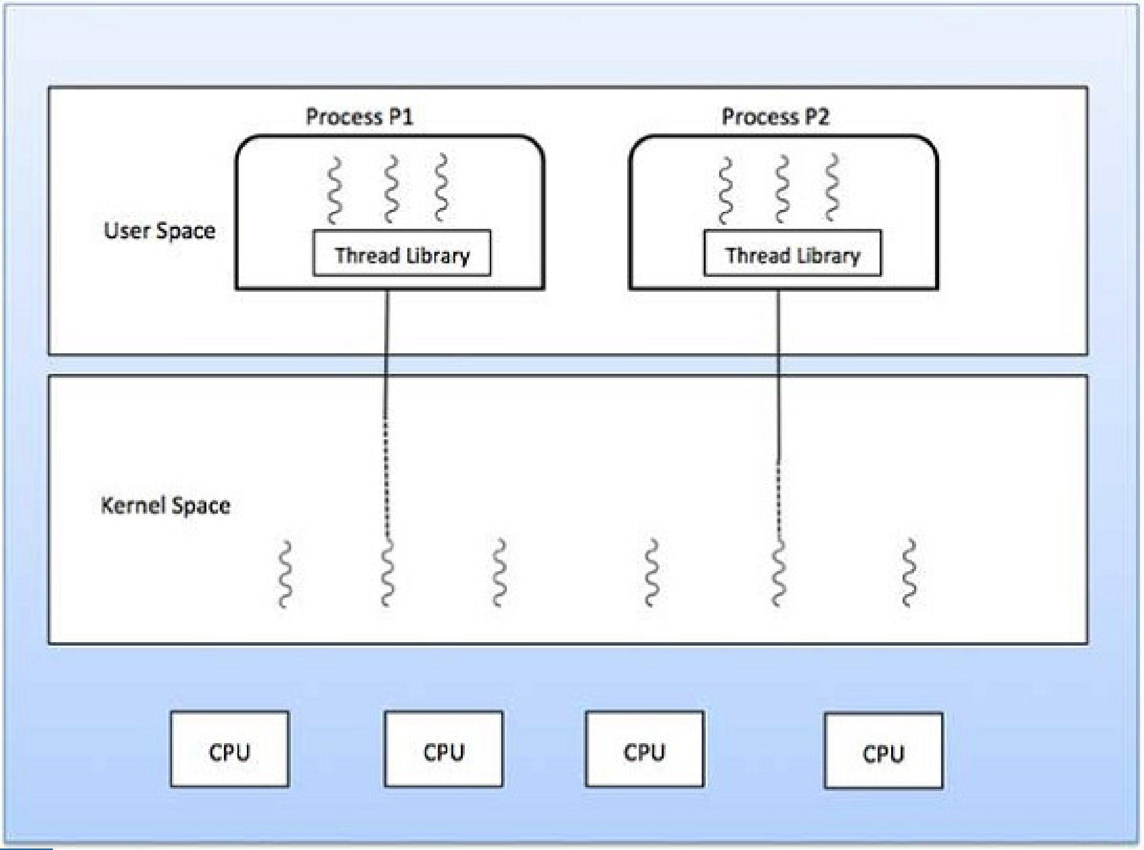
Hybrid

- Use kernel-level threads and then multiplex user-level threads onto same or all kernel threads.



Multithreading Models

- Many to one Model



•Thread management is done in user space.

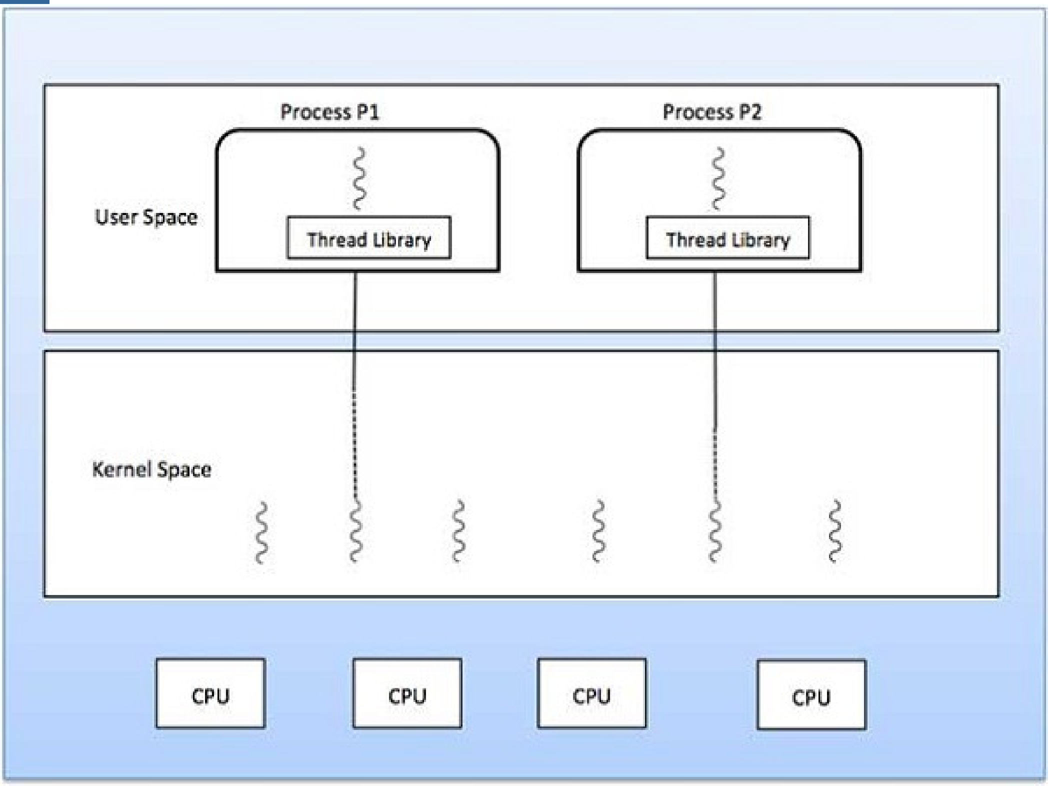
•Entire process is block if a thread makes blocking system call.

•No used in multi processors system. No concurrency.

•Green threads – Solaris thread library.

Multithreading Models

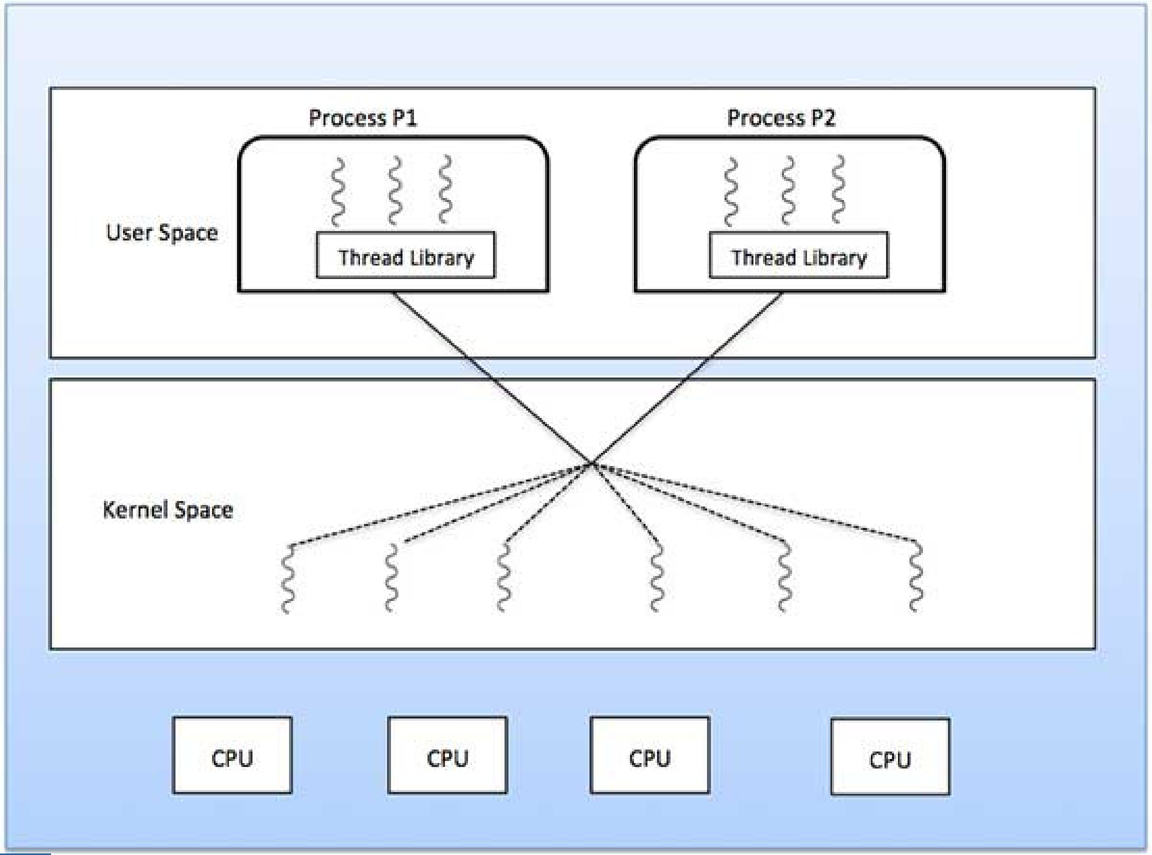
- One to one Model



- Provide more concurrency than many to one.
  - another thread will run when a thread makes a blocking system call.
- Multiple threads to run in parallel on microprocessors.
- Drawback: creating a user thread, requires to create a kernel thread.
- Restriction has # User (or kernel) threads are supported by the system.
- Windows NT, Windows 2000

Multithreading Models

- Many to Many Model



•Many users level threads to a smaller or equal number of kernel threads.

•Solve one-to-one, Many-to-one Model's restrictions.

•Users can create as many user threads as need

- corresponding kernel threads can run parallel @ multiprocessor

•Solaris 2, IRIX, HP-UX

Thread Issues

- The fork and exec System Calls
  - If one thread calls fork, does the process duplicate all threads/ single thread.
    - Two versions fork call-
      - one duplicates all threads, another duplicates only thread that invokes.
  - If any thread calls exec system call - replace the entire process.
  - If any thread calls exec system call after fork- duplicating all threads is unnecessary.
- Generating signals to a thread:
  - Signals are given to process not threads
  - Threads need to register signals
  - However, what if two processes register same signal??

Observation

- Creating threads is faster than creating process >> Multithreaded server is better?
- Multi-threaded server has some problematic issues
  - amount of time requires to create a thread before process request
  - Process request -> new thread creation
    - No bound on thread creation
- Solution: thread pools
  - Prior creates some threads before process startup and place in pool
    - faster and limit on thread creation
  - Size of the pool limited to # of CPUs, memory, expected # of client requests
  - Dynamic pool architecture
    - adjust # of threads according to usage patterns

POSIX Thread Program-1

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

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
```

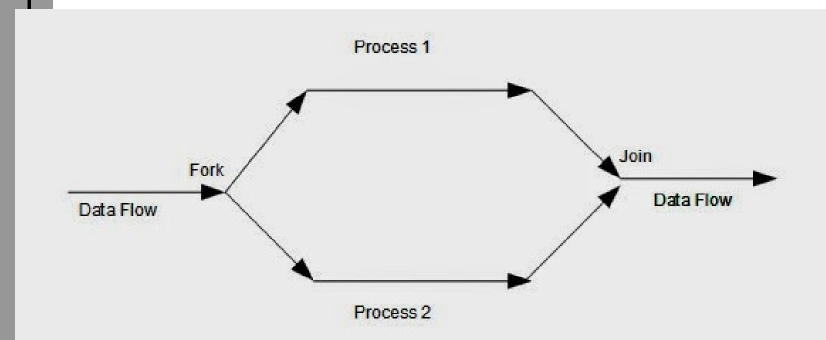
```
int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code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

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Opaque Object

POSIX Thread Program-2

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

/* Global variable:  accessible to all threads */
int thread_count;

void* Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
}  /* main */

void* Hello(void* rank) {
   long my_rank = (long) rank
         /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank,
         thread_count);

   return NULL;
}  /* Hello */
```

Program 4.1: A Pthreads "hello, world" program



gcc –o pth_hello pth_hello.c –lpthread

./pthread 2

Course materials:
Galvin book 5.1-5.2,5.3.1

End of Lecture 6,7