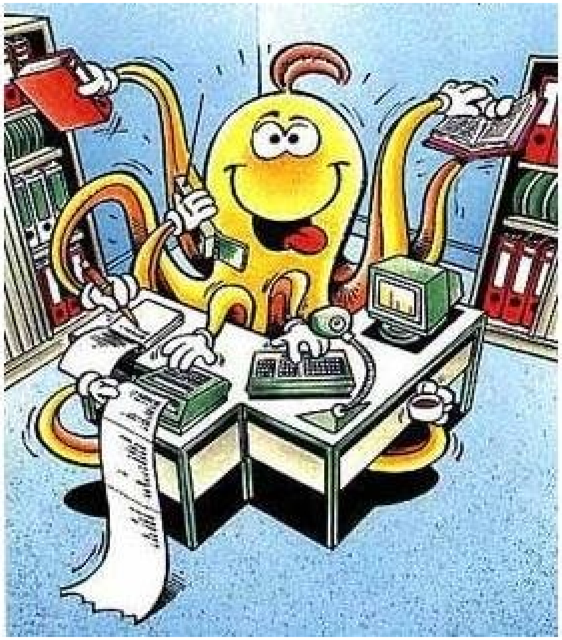
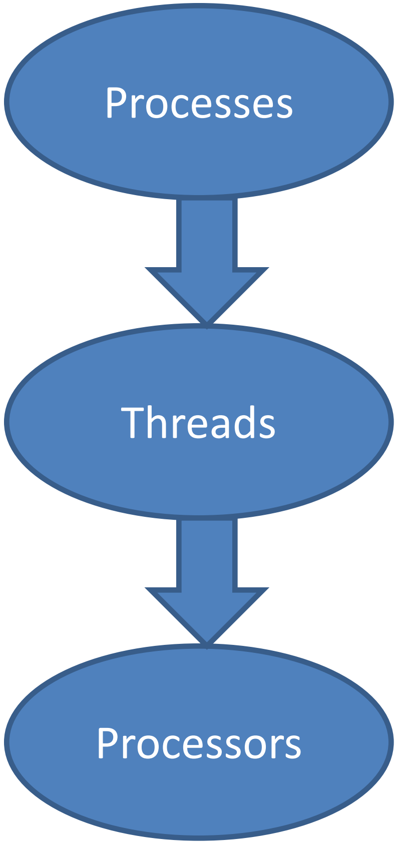
Operating Systems

Concurrency I

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Multithreading





Processes Vs Threads

- Process has its own address space.
- Threads are created within a process.
  - Sequential program = one process that contains one thread.
- A thread has:
  - an execution state (Running, Ready, etc.)
  - saved thread context when not running
  - access to the memory and resources of its process (all threads of a process share this)

Benefits of Threads

Takes less time to create a new thread than a process

Less time to terminate a thread than a process

Switching between two threads takes less time than switching between processes

A Thread

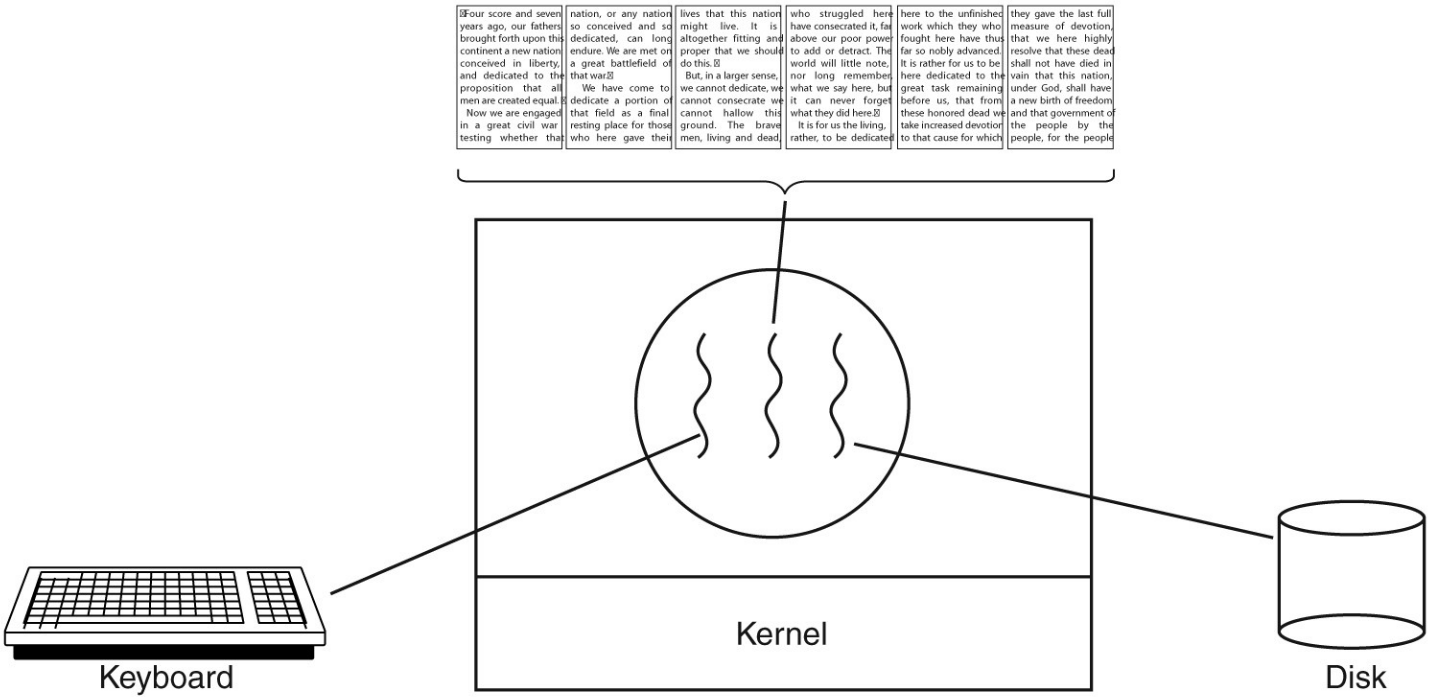
- **Definition**: sequence of related instructions executed independently of other instruction sequences from the same process.
- A thread can create another thread.
- Threads within the same process share:
  - Heap
  - Data
  - Any opened files
- Each thread has its own stack.

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Why Use Threads?

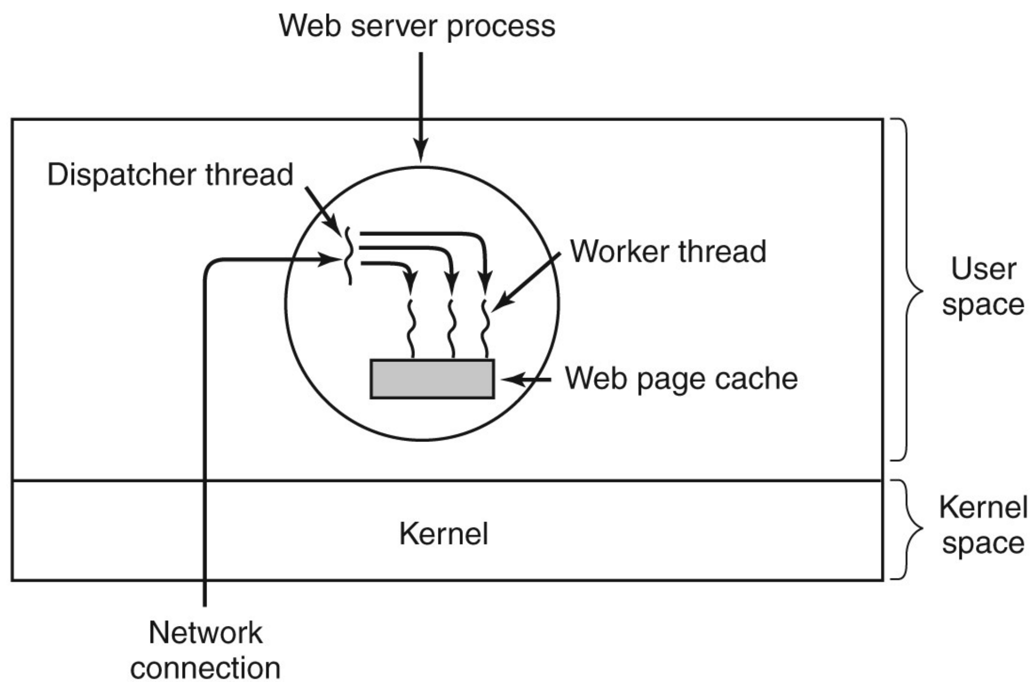
- Parallelism
- Lighter than processes
- Avoid reduce process blocking due to I/O

Example 1: A Word Processor



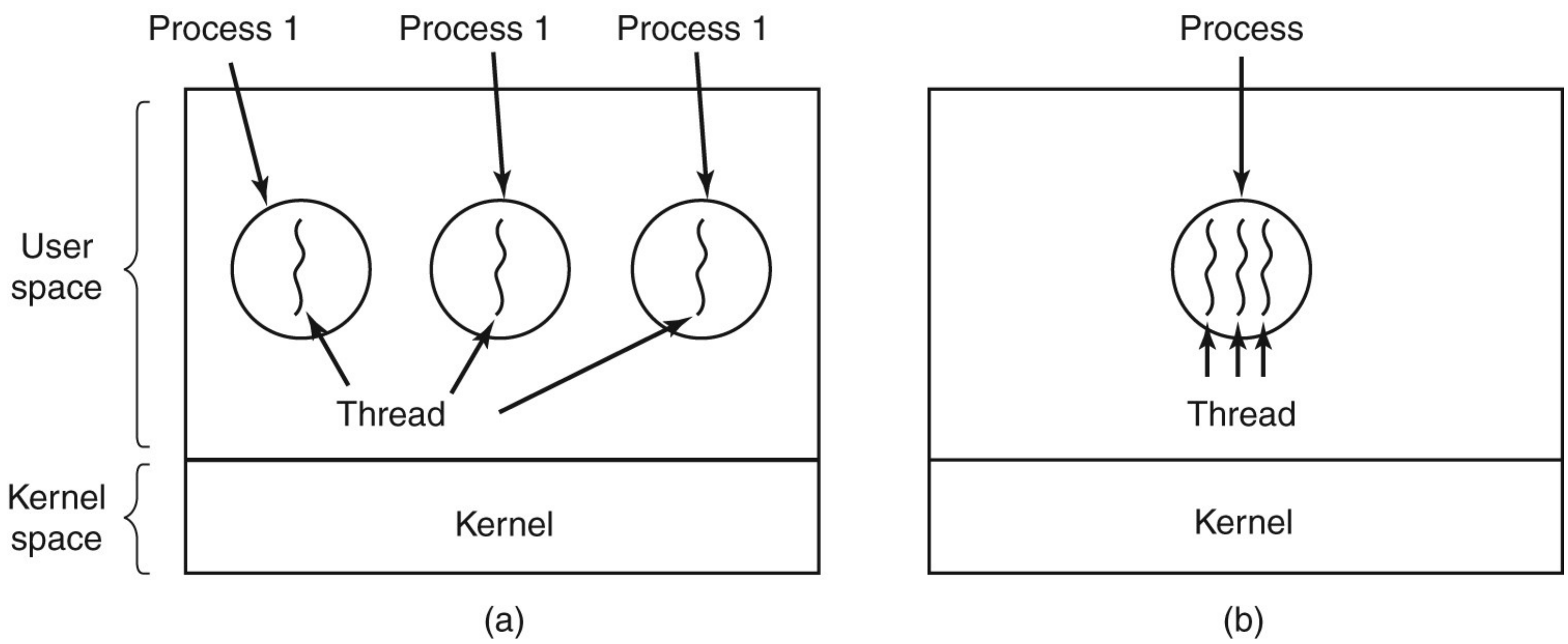
One thread reads from keyboard, another thread checks spelling, and the third writes to disk.

Example 2: Multithreaded Web Server

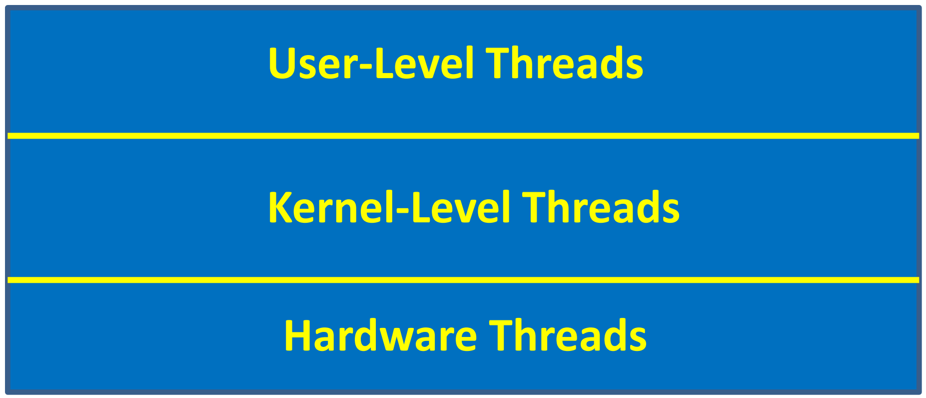


One thread gets request from client through the network, another threads analyses the request, and a third thread does the requested work (such as retrieving data).

Different Implementation of Concurrency

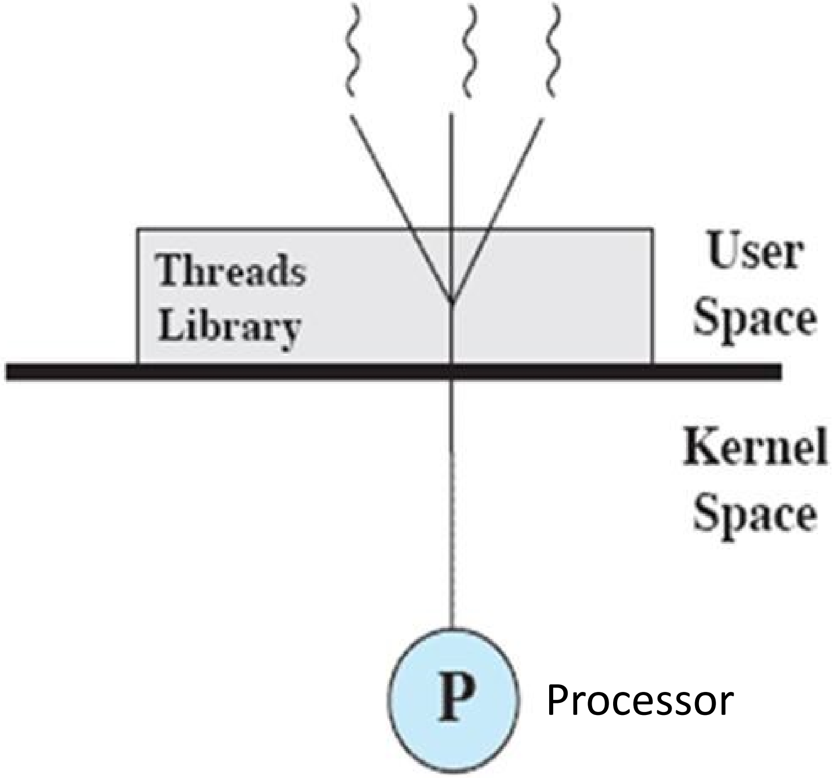


Not all threads are created the same



User-Lever Threads (ULT)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



Implementing Threads in User Space

- Threads are implemented by a library.
- Kernel knows nothing about threads.
- Each process needs its own private thread table.
- Thread table is managed by the runtime system.

User-Level Threads (ULTs)

Advantages

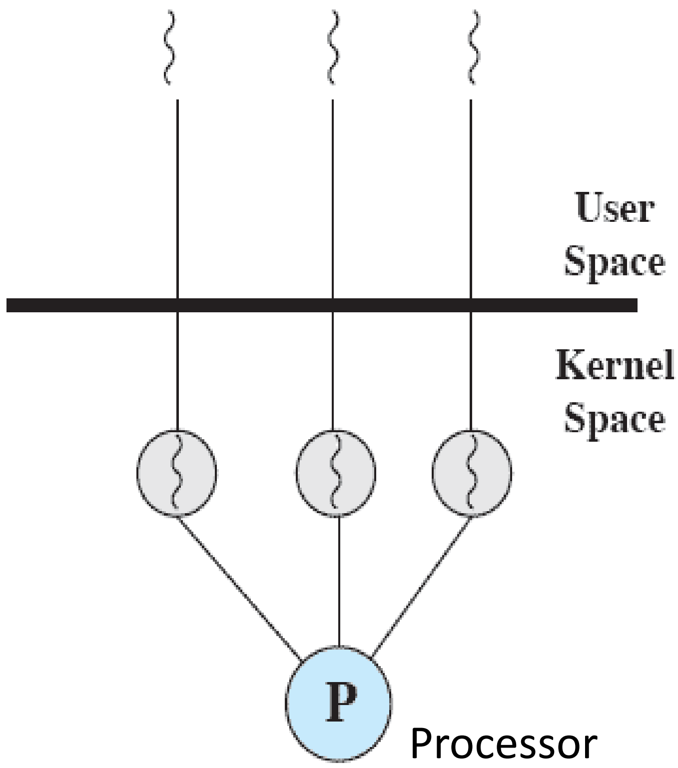
- Thread switch does not require kernel-mode.
- Scheduling (of threads) can be application specific.
- Can run on any OS.

Disadvantages

- A system-call by one thread can block all threads of that process.
- In pure ULT, multithreading cannot take advantage of multiprocessing/multicore

Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- No thread management is done by the application
- Windows OS and Linux threads are examples of this approach.



Implementing Threads in Kernel Space

- Kernel knows about and manages the threads
- Creating/destroying/(other thread related operations) a thread involves a system call

Kernel-Level Threads (KLTs)

Advantages

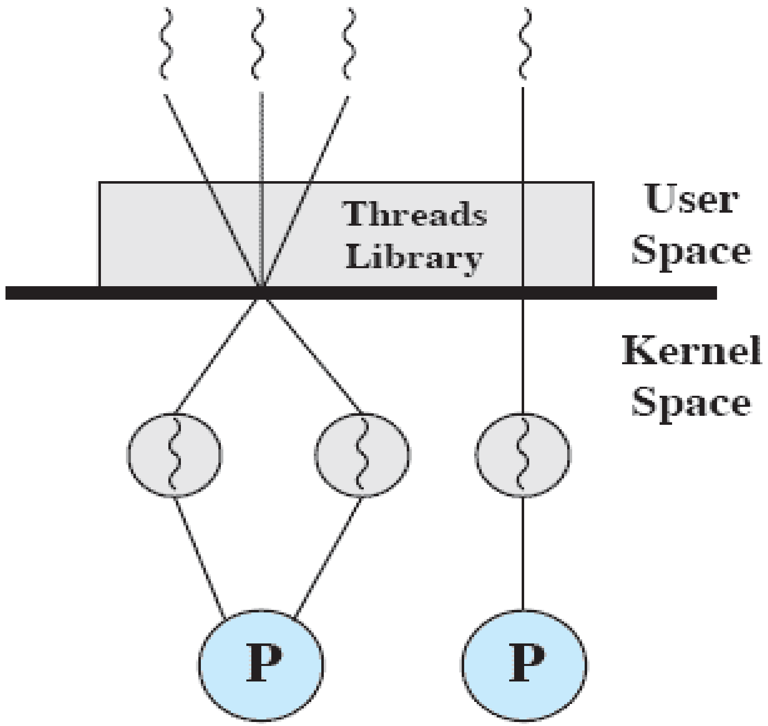
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

Disadvantages

- The transfer of control from one thread to another within the same process requires a switch to the kernel

Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example



Relationship Between ULTs & KLTs

- 1:1
  - user-level thread maps to kernel-level thread
- N:1 (user-level threads)
  - Kernel is not aware of the existence of threads
  - e.g. Early version of Java
- M:N

Example 1

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

This how we initialize a thread pointer.

create a thread that executes the function “mythread”

Wait till thread pointed to by p1 finishes.

Example 1

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Possible Execution 2

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Example 1

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Possible Execution 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Example 1

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Possible Execution 3

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

What Does Example 1 Tell Us?

Order of execution of threads is, in general, non-deterministic.

Example 2

```
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
            counter);
    return 0;
}
```

Possible Executions of Example 2

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Correct result!

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Wrong result!

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Wrong result!

Why did this happen?

counter = counter + 1;

Is translated to machine code that does the following:

1. Read counter from memory to register
2. Increment the content of the register by 1
3. Write the register value back to memory

Why did this happen?

Assume current value of *counter* = 50

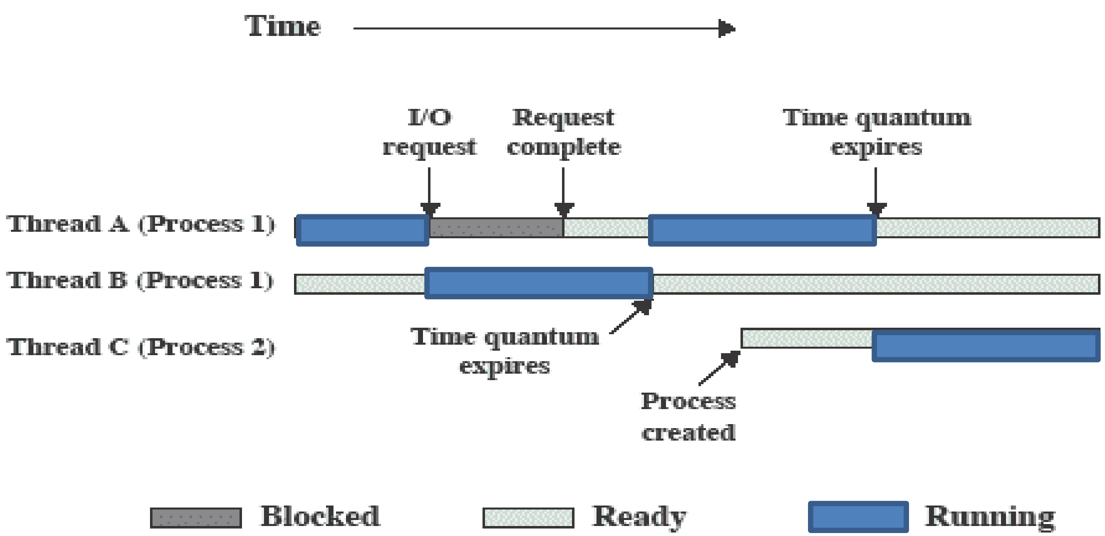
- Thread 1: Reads *counter* into register Rx
- Thread 1: Rx = Rx + 1 (so Rx now is 51)
- Time slice is up and Thread 2 is scheduled
- Thread 2: Reads *counter* into register Ry
- Thread 2: Ry = Ry + 1 (so Rx now is 51)
- Time slice is up and Thread 1 is scheduled
- Thread 1: Write Rx to counter (i.e. 51)
- Time slice is up and Thread 2 is scheduled
- Thread 2: Write Ry to counter (i.e. 51)
- Final value of counter after is 51 (correct value is 52)

Time

What Does Example 2 Tell Us?

Non-determinism in threads execution may lead to wrong results if more than one thread try to update the same piece of data.

Multithreading on Uniprocessor System



The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

—*THE SCIENCES OF THE ARTIFICIAL*, Herbert Simon

Conclusions

- Processes → threads → processors
- Threads ensure concurrency and reduce blocking due to I/O.
- There is non-determinism in threads execution, which can cause problems.