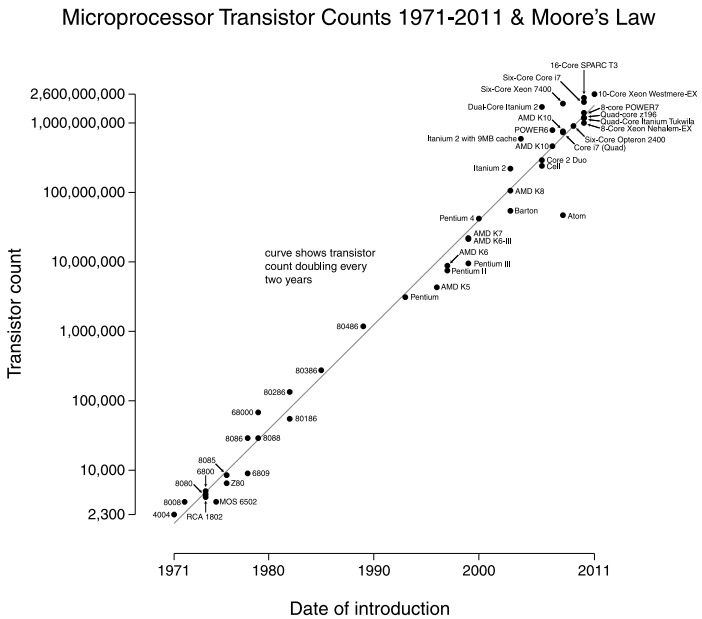
COMP20200 Unix Programming

Lecture 11

CS, University College Dublin, Ireland

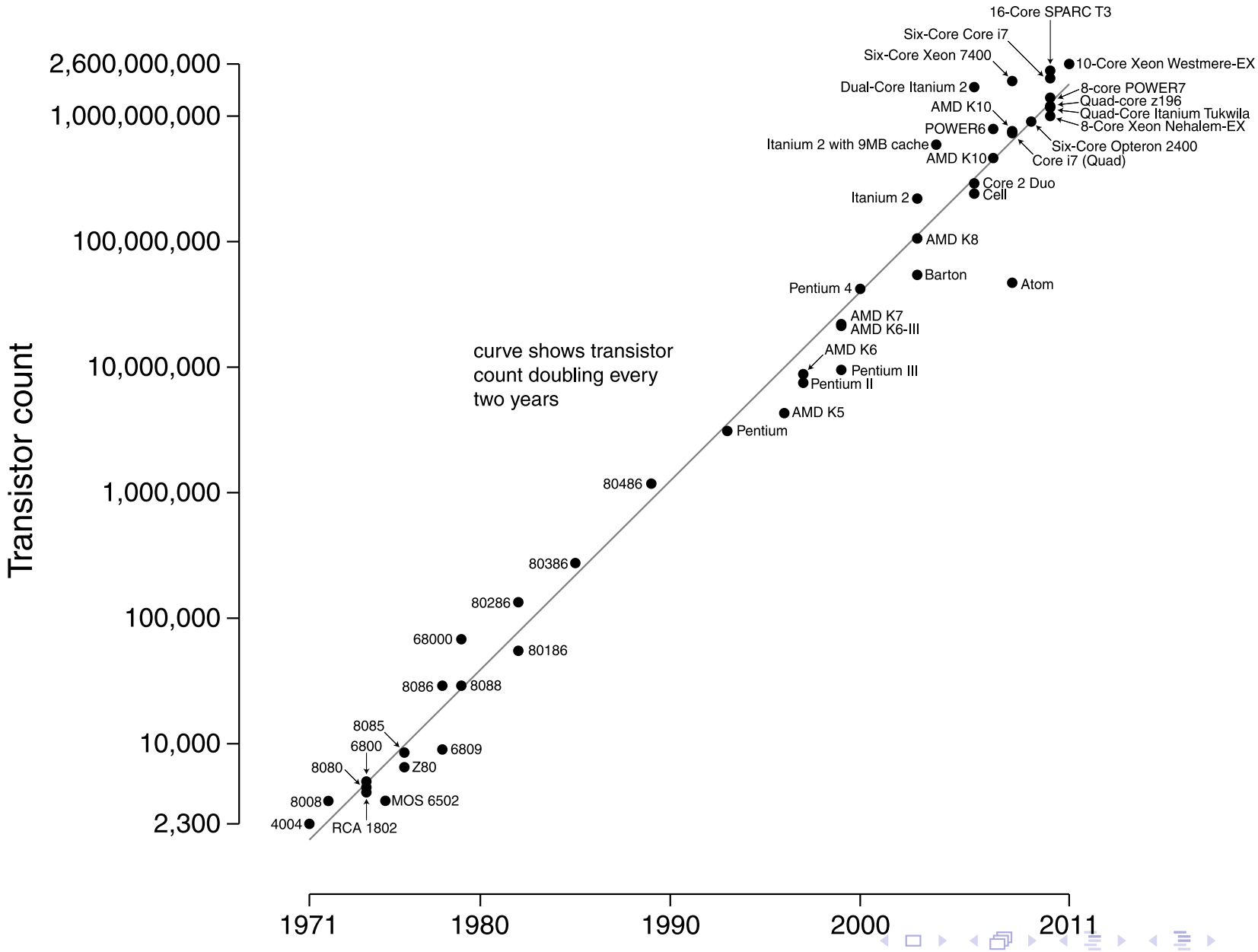
Moors law

- Gordon E. Moore in 1965 predicted "the number of transistors on integrated circuits doubles approximately every two years."
- Adjusted to doubling performance every 18 months.



- Need parallelism to use all these transistors: multi-core, many-core.

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Parallelism

- Blocking IO
  - writing to disk or network
  - Could wait, asynchronous IO or spawn a separate thread
- Server
  - receiving multiple clients requests (eg. web server)
- User Interface
  - Respond to user input while program engine busy
- Performance on machines
  - Desktop, mobile, supercomputer clusters.
  - SMPs (shared memory multiprocessor), GPU-accelerators, clusters
  - Program must be parallel for performance.
  - SIMD (single instruction, multiple data)

Processes and Threads

- `fork()` returns two completely independent copies of the original process.
  - Each process has own address space, own copies of variables independent of same variables in other process.
  - provides memory protection and therefore stability
  - problem when multiple processes working on same task/problem
- Light-weight process (LWP): POSIX threads or pthreads
  - set of C programming language types, functions and constants
  - Around 100 Pthreads procedures, all prefixed "`pthread_`"

POSIX threads

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

#define NUM_THREADS     5

void *TaskCode(void *argument) {
    int tid;

    tid = *((int *) argument);
    printf("Hello World! It's me, thread %d!\n", tid);

    /* optionally: insert more useful stuff here */

    return NULL;
}
```

```c
int main(void) {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int rc, i;

    /* create all threads */
    for (i=0; i<NUM_THREADS; ++i) {
        thread_args[i] = i;
        printf("In main: creating thread %d\n", i);
        rc = pthread_create(&threads[i], NULL, TaskCode,
                            (void *)&thread_args[i]);
        assert(0 == rc);
    }

    /* wait for all threads to complete */
    for (i=0; i<NUM_THREADS; ++i) {
        rc = pthread_join(threads[i], NULL);
        assert(0 == rc);
    }

    exit(EXIT_SUCCESS);
}
```

Pthreads

- When compiling with `gcc`, must add `-pthread` flag.

```
$ gcc -Wall -o pthread_example pthread_example.c -pthread
$
```

Output:

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

Race conditions

THREAD 1

```
a = data;
a++;
data = a;
```

THREAD 2

```
b = data;
b--;
data = b;
```

- If above executed serially, no problem, expected result.
- If in parallel it is completely non-deterministic. `data` may be +1, 0, -1.
- Solution: block other threads when writing to `data`
- Pthreads use a data type called *mutex to achieve this.*

mutex example

THREAD 1

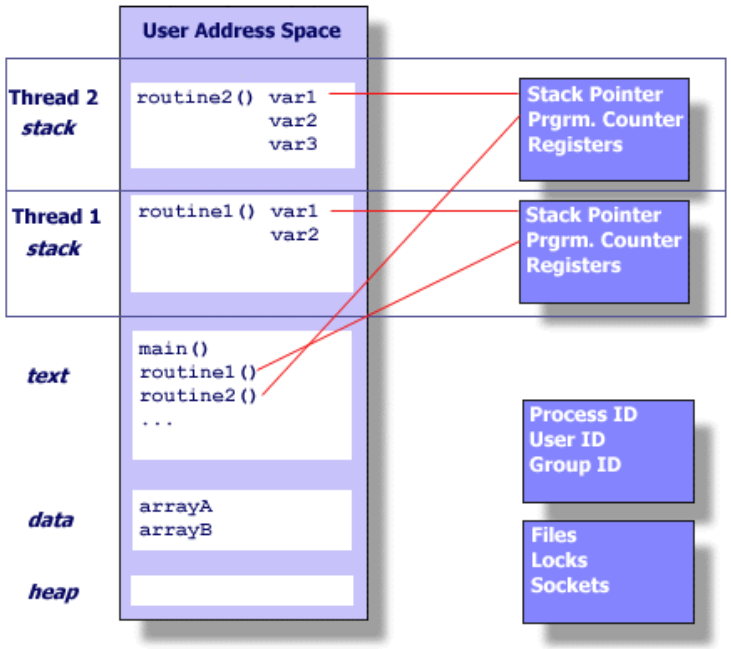
```
pthread_mutex_lock (&mut);

a = data;
a++;
data = a;
pthread_mutex_unlock (&mut);
```

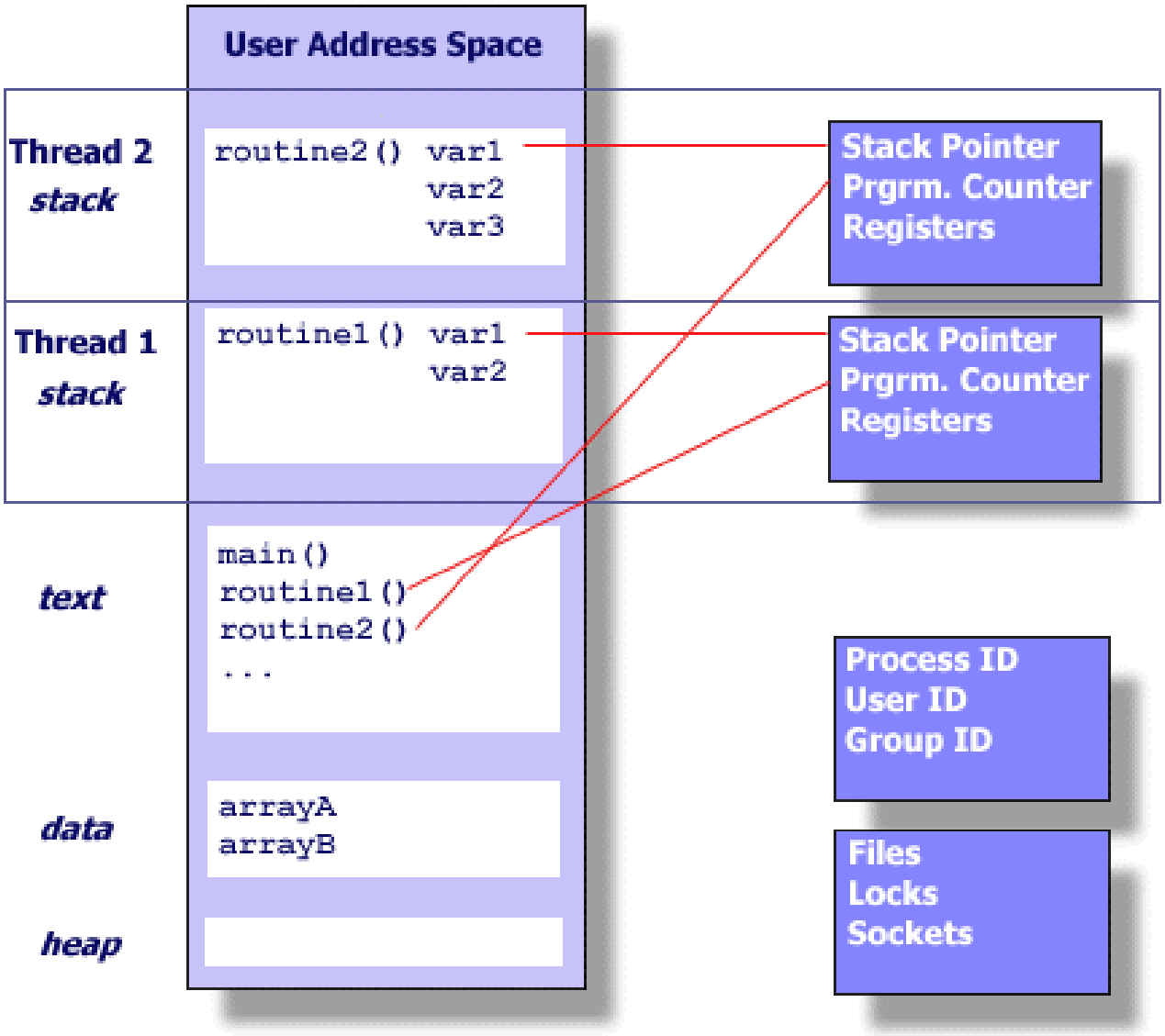
THREAD 2

```
pthread_mutex_lock (&mut);
/* blocked */
/* blocked */
/* blocked */
/* blocked */
b = data;
b--;
data = b;
pthread_mutex_unlock (&mut)
    ;
```

POSIX threads

- Independent stream of instructions
- High performance on SMP
- Duplicate only the bare essential resources
- Accomplished by the thread maintaining its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.




User Address Space
Thread 2
stack
routine2() var1
var2
var3
Thread 1
stack
routine1() var1
var2
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Stack Pointer
Prgrm. Counter
Registers
Stack Pointer
Prgrm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets

In UNIX environment, a thread:

- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists
- Duplicates only the essential resources it needs to be independently schedulable
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

Threads share the process resources

Because threads within the same process share resources:

- Changes made by one thread to shared resources will be seen by all threads.
  - eg. closing a file
- Two pointers having the same value point to the same data.
- Reading and writing to the same memory locations is possible
  - but requires explicit synchronization by the programmer.