CSCI-UA.0201

Computer Systems Organization

Concurrency - Multithreading

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Moore’s Law

40 Years of Microprocessor Trend Data

Transistors (thousands)

Single-Thread Performance (SpecINT x 10^3)

Frequency (MHz)

Typical Power (Watts)

Number of Logical Cores

Year


Moore’s Law

40 Years of Microprocessor Trend Data

Transistor count still rising

Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp
Moore’s Law

40 Years of Microprocessor Trend Data

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten

New plot and data collected for 2010-2015 by K. Rupp
Moore’s Law (in practice)
Traditional Scaling Process

Speedup

1.8x  3.6x  7x

User code

Traditional Uniprocessor

Time: Moore’s law
Ideal Scaling Process

Speedup

1.8x

3.6x

7x

User code

Multicore
Ideal Scaling Process

Unfortunately, not so simple...
Actual Scaling Process

Speedup:

- 1.8x
- 2x
- 2.9x

User code:

Multicore:

10
Actual Scaling Process

Parallelization and Synchronization require great care...
Multithreading Basics
Example

```c
long bigloop(int *arr, int len) {
    long r = 0;
    for(int i = 0; i < len; i++)
        r += arr[i];
    return r;
}

int main() {
    int *arr = malloc(8 * sizeof(int));
    ...
    long r = bigloop(arr, 8);
    ...
}
```

How to improve the performance with multithreading?
Parallelization

bigloop 0 → 7

0
1
2
3
4
5
6
7

CPU 0  CPU 1  CPU 2  CPU 3
Parallelization

Performance can be improved by 4X
Parallelization

What is concurrency?
• things happening "simultaneously"
  – multiple CPU cores concurrently executing instructions
  – CPU and I/O devices concurrently doing processing

Performance can be improved by 4X
Concurrency

What is concurrency?
- multiple CPU cores concurrently executing instructions
- CPU and I/O devices concurrently doing processing

• Why write concurrent programs?
  - speed up programs using multiple CPUs
  - speed up programs by interleaving CPU processing and I/O.
How to write concurrent programs?

• Use multiple processes
  – Each process uses a different CPU
  – Different processes runs different tasks
  – They have separate address spaces
  – It is difficult to communicate with each other

• Use multiple threads
How to write concurrent programs?

• Use multiple processes
  – Each process uses a different CPU
  – Different processes runs different tasks
  – They have separate address spaces
  – It is difficult to communicate with each other

• Use multiple threads
Multiple threads (Multithreading)

```c
long bigloop(int *arr, int len) {
    long r = 0;
    for(int i = 0; i < len; i++)
        r += arr[i];
    return r;
}

int main() {
    int *arr = malloc(8 * sizeof(int));
    ...
    long r = bigloop(arr, 8);
    ...
}
```
Multiple threads (Multithreading)
Multiple threads (Multithreading)

Single process, multiple threads
- Threads Share the same virtual memory space
- Each thread
  - has its own stack
  - has its own control flow

Process

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigloop 0 → 1</td>
<td>bigloop 2 → 3</td>
<td>bigloop 4 → 5</td>
<td>bigloop 6 → 7</td>
</tr>
</tbody>
</table>
| 0  
1 | 2  
3 | 4  
5 | 6  
7 |

CPU 0  
CPU 1  
CPU 2  
CPU 3
Thread Local Stack

Process

Thread 0  Thread 1  Thread 2  Thread 3

Memory invisible to user code

Kernel virtual memory

User stack

%rsp (stack pointer)

%rsp

brk

Loaded from the executable file

Shared libraries

Run-time heap

Read/write data segment

Read-only code segment

Unused

Thread 0
Thread 1
Thread 2
Thread 3

0x400000

0
Thread Local Stack

Each thread has its own stack segment
• Each thread has its own stack pointer
• Store the stack pointer into %rsp before running
POSIX Thread Interface

• POSIX: Portable Operating System Interface
  – POSIX defines the API for variants of Unix

• Thread interface defined by POSIX
  – pthread_create: create a new thread
  – pthread_join: wait until the target thread has terminated
#include <pthread.h>
int pthread_create(pthread_t *thread_id,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);

• Create a new thread
  – It executes start_routine with arg as its sole argument.
  – Its attribute is specified by attr
  – Upon successful completion, it will store the ID of the created thread in the location referenced by thread_id.

• Return value
  – zero: success
  – non-zero (error number): fail
void* func(void* arg) {
    printf("This is the created thread\n");
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, NULL);
    if(r != 0) {
        printf("create thread failed");
        return 1;
    }
    return 0;
}

$ gcc create.c -lpthread
Example 1 - Create

```c
void* func(void* arg) {
    printf("This is the created thread\n");
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, NULL);
    if(r != 0) {
        printf("create thread failed");
        return 1;
    }
    return 0;
}
```

$ gcc create.c -lpthread

Main thread returns before the created thread finishes.

- Automatically terminate and reclaim the created thread.
#include <pthread.h>

int pthread_join(pthread_t thread_id, void **ret_ptr);

- **Wait for the target thread to finish**
  - The target thread is specified by `thread_id`
  - Upon success, the return value of the created thread will be available in the location referenced by `ret_ptr`.

- **Return value**
  - zero: success
  - non-zero (error number): fail
void* func(void* arg) {
    printf("This is the created thread\n");
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, NULL);
    if(r != 0) {
        ...
        r = pthread_join(tid, NULL);
        if(r != 0)
            ...
    
    return 0;
}
Example 3 - Parameter

```c
void* func(void* arg) {
  int p = *(int *) arg;
  p = p + 1;
  return &p;
}

int main(int argc, char* argv[]) {
  int param = 100;
  pthread_t tid;
  int r = pthread_create(&tid, NULL, &func, (void *) &param);
  ...
  int *res = NULL;
  r = pthread_join(tid, &res);
  ...
  printf("result: addr %lx, val %d\n", res, *res);
  return 0;
}
```
Example 3 - Parameter

```c
void* func(void* arg) {
    int p = *(int *) arg;
    p = p + 1;
    return &p;
}

int main(int argc, char* argv[]) {
    int param = 100;
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, (void *) &param);
    ...
    int *res = NULL;
    r = pthread_join(tid, &res);
    ...
    printf("result: addr %lx, val %d\n", res, *res);
    return 0;
}
```

Question – what is the expected output?
Example 3 - Parameter

```c
void* func(void* arg) {
    int p = *(int *) arg;  // p is on the stack of the created thread
    p = p + 1;             // -- it is no longer valid when the thread terminates
    return &p;
}

int main(int argc, char* argv[]) {
    int param = 100;
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, (void *) &param);
    ...
    int *res = NULL;
    r = pthread_join(tid, &res);
    ...
    printf("result: addr %lx, val %d\n", res, *res);
    return 0;
}
```

Question – what is the expected output?
Example 3 - Parameter

```c
void* func(void* arg) {
    int p = *(int *) arg;
    p = p + 1;
    int *r = (int *) malloc(sizeof(int));
    *r = p
    return (void *) r;
}

int main(int argc, char* argv[]) {
    int param = 100;
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, (void *) &param);
    ... 
    int *res = NULL;
    r = pthread_join(tid, &res);
    ...
    printf("result: addr %lx, val %d\n", res, *res);
    return 0;
}
```
Example 3 - Parameter

```c
void* func(void* arg) {
    int p = *(int *) arg;
    p = p + 1;
    int *r = (int *) malloc(sizeof(int));
    *r = p
    return (void *) r;
}

int main(int argc, char* argv[]) {
    int param = 100;
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, (void *) &param);
    ...
    int *res = NULL;
    r = pthread_join(tid, &res);
    ...
    printf("result: addr %lx, val %d\n", res, *res);
    free(res);
    return 0;
}
```
Example 4 - Interleaving

```c
void* func(void* arg) {  
    printf("1");
}

int main(int argc, char* argv[]) {  
    printf("0");

    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, NULL);
    ...
    printf("2");
    ...
    return 0;
}
```

Question – what is the expected output?
Example 4 - Interleaving

```c
void* func(void* arg) {
    printf("1");
}

int main(int argc, char* argv[]) {
    printf("0");

    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, NULL);
    ...
    printf("2");
    ...
    return 0;
}
```

Question – what is the expected output?

Answer: 012 or 021
Example 4 - Interleaving

void* func(void* arg) {
    printf("1");
}

int main(int argc, char* argv[]) {
    printf("0");

    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, NULL);

    ...
    printf("2");
    ...
    return 0;
}
Example 4 - Interleaving

```c
void* func(void* arg) {
    printf("1");
}

int main(int argc, char* argv[]) {
    printf("0");

    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, NULL);

    ...
    printf("2");
    ...
    return 0;
}
```

Question – what is the expected output?

Answer: 012 or 021

**Diagram:**
- **Main thread**: 012
- **Func thread**: 021
void* func(void* arg) {
    int p = *(int *) arg;
    p = p + 1;
    int *r = (int *) malloc(sizeof(int));
    *r = p
    return (void *) r;
}

int main(int argc, char* argv[]) {
    int param = 100;
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, (void *) &param);
    ... 
    int *res = NULL;
    r = pthread_join(tid, &res);
    ...
    printf("result: addr %lx, val %d\n", res, *res);
    free(res);
    return 0;
}
Example 3 Revisited

```c
void* func(void* arg) {
    int *p = (int *) arg;
    *p = *p + 1;
    return NULL;
}

int main(int argc, char* argv[]) {
    int param = 100;
    pthread_t tid;
    int r = pthread_create(&tid, NULL, &func, (void *) &param);
    ...
    int *res = NULL;
    r = pthread_join(tid, &res);
    ...
    printf("result: %d\n", param);
    return 0;
}
```

Question – can we get rid of \texttt{r} in \texttt{func}?
Example 5 – Stack, Heap, Global

```c
int global = 0;

void* write(void* arg) {
    int local = 0;
    local = 10;
    global = 10;
    int *ptr = (int *)arg;
    (*ptr) = 10;
}

void* read(void* arg) {
    int local = 0;
    printf("local %d global %d heap %d\n", local, global, *(int *)arg);
    return NULL;
}

int main() {
    int *p = (int *) malloc(sizeof(int));
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, &write, (void *)p);
    ... // Wait for thread 1 to finish
    pthread_join(tid1, NULL);
    pthread_create(&tid2, NULL, &read, (void *)p);
    ... // Wait for thread 2 to finish
    return 0;
}
```
Example 5 – Stack, Heap, Global

Each thread has its own stack segment

- Each thread has its own stack pointer
- Store the stack pointer into %rsp before running
Example 5 – Stack, Heap, Global

```c
int global = 0;

void* write(void* arg) {
    int local = 0;
    local = 10;
    global = 10;
    int *ptr = (int*)arg;
    (*ptr) = 10;
}

int main() {
    int *p = (int*)malloc(sizeof(int));
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, &write, (void*)p);
    ... 
    pthread_join(tid1, NULL);
    pthread_create(&tid2, NULL, &read, (void*)p);
    ... 
    return 0;
}

void* read(void* arg) {
    int local = 0;
    printf("local %d global %d heap %d\n", local, global, *(int*)arg);
    return NULL;
}
```

What are the outputs?

```c
local 0 global 10 heap 10
```
Example 5 – Stack, Heap, Global

```c
int global = 0;

void* write(void* arg) {
    int local = 0;
    local = 10;
    global = 10;
    int *ptr = (int *)arg;
    (*ptr) = 10;
}

int main() {
    int *p = (int *) malloc(sizeof(int));
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, &write, (void *)p);
    pthread_create(&tid2, NULL, &read, (void *)p);
    pthread_join(tid1, NULL);
    pthread_create(&tid2, NULL, &read, (void *)p);
    return 0;
}
```

What are the outputs?

```
local 0 global 10 heap 10
```
Example 5 – Stack, Heap, Global

```c
int global = 0;

void* write(void* arg) {
    int local = 0;
    local = 10;
    global = 10;
    int *ptr = (int *) arg;
    (*ptr) = 10;
}

void* read(void* arg) {
    int local = 0;
    printf("local %d global %d heap %d\n", local, global, *(int *) arg);
    return NULL;
}

int main() {
    int *p = (int *) malloc(sizeof(int));
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, &write, (void *)p);
    pthread_create(&tid2, NULL, &read, (void *)p);
    pthread_join(tid1, NULL);
    pthread_create(&tid2, NULL, &read, (void *)p);
    return 0;
}
```

What are the outputs?

- local 0 global 10 heap 10
- local 0 global 10 heap 0
- local 0 global 0 heap 0