Concurrency – Multithreading

Jinyang Li

based on slides by Tiger Wang
Example

```c
long bigloop(int *arr) {
    long r = 0;
    for(int i = 0; i < 8; i++)
        r += arr[i];
    return r;
}

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

How to improve the performance with multicore?
Parallelization

bigloop: 0→7

CPU0
CPU1
CPU2
CPU3
Parallelization

Performance can be improved by 4X
What's 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's concurrency?
- things happening "simultaneously"
  - 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.
In this lecture

What's concurrency?
- things happening "simultaneously"
  - 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
In this lecture

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

Use multiple threads
Multiple threads (Multithreading)

Process

long bigloop(int *arr) {
    long r = 0;
    for(int i = 0; i < 8; i++)
        r += arr[i];
    return r;
}

int main() {
    int *arr = malloc(8 * sizeof(int));
    ...
    long r = bigloop(arr, 1);
    ...
}
Multiple threads (Multithreading)

Process

thread 0
bigloop: 0→1

thread 1
bigloop: 2→3

thread 2
bigloop: 4→5

thread 3
bigloop: 6→7

CPU0

CPU1

CPU2

CPU3
Multiple threads (Multithreading)

Single process, multiple threads
- Share the same memory space
- 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><img src="image" alt="bigloop: 0→1" /></td>
<td><img src="image" alt="bigloop: 2→3" /></td>
<td><img src="image" alt="bigloop: 4→5" /></td>
<td><img src="image" alt="bigloop: 6→7" /></td>
</tr>
<tr>
<td><img src="image" alt="0" /> <img src="image" alt="1" /></td>
<td><img src="image" alt="2" /> <img src="image" alt="3" /></td>
<td><img src="image" alt="4" /> <img src="image" alt="5" /></td>
<td><img src="image" alt="6" /> <img src="image" alt="7" /></td>
</tr>
</tbody>
</table>

CPU0
CPU1
CPU2
CPU3
Share the memory space

Different processes have different page tables
Different processes have different page tables

Different threads of the same process share the same page table
Thread local stack

Process 1

thread 0  thread 1  thread 2  thread 3

Kernel virtual memory
- User stack
- Shared libraries
- Runtime heap
- Read/write segment
- Read-only segment
- Unused

Memory invisible to user code

%rsp (stack pointer)

brk

Loaded from the executable file
Thread local stack

Each thread has its own stack segment
- Each thread has its own stack pointer
- Store the stack pointer into the %rsp before running

Process 1

thread 0 thread 1 thread 2 thread 3

CPU 0
RSP: sp0

CPU 1
RSP: sp1

CPU 2
RSP: sp2

CPU 3
RSP: sp3

Memory
invisible to
user code

Kernel virtual memory

User stack 0

sp0

User stack 1

sp1

User stack 2

sp2

User stack 3

sp3

Shared libraries

Runtime heap

Read/write segment

Read-only segment

brk

Loaded from
the
executable
file

Unused

0x400000

0
Own control flow

Each thread loads PC register of local CPU with different instructions

Process 1

thread 0 thread 1 thread 2 thread 3

CPU 0
PC: addr1
IR: movq ...
RSP: sp0

CPU 1
PC: addr2
IR: addq ...
RSP: sp1

CPU 2
PC: addr3
IR: mulq ...
RSP: sp2

CPU 3
PC: addr4
IR: subq ...
RSP: sp3

Kernel virtual memory
User stack 0
User stack 1
User stack 2
User stack 3
Shared libraries
Runtime heap
Read/write segment
Read-only segment
Unused

Memory invisible to user code
sp0
sp1
sp2
sp3
brk

Loaded from the executable file

0x400000
0
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 for the target thread terminated
**pthread_create**

```c
#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
Example 1 – Create

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;
}
```

Main thread returns before the created thread finishes.
- Automatically terminate and reclaim the created thread.

gcc create.c -lpthread
#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;
}
```

Question – what is expected result?
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;
}
```

p is on the stack of the created thread -- it is destroyed when the thread terminates
Example 3 – Parameter

```c
void* func(void* arg) {
    int p = *(int *)arg;
    p = p + 1;
    int *r = (void *)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 = (void*)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 – Interleave

```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 result?
Example 4 – Interleave

```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 result?

Answer: 012 or 021
Example 4 – Interleave

```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 result?

Answer: 012 or 021
Example 4 – Interleave

```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 result?

Answer: 012 or 021
Example 5 – Stack, Heap, Global

```c
int global = 0;

void* write(void* arg) {
    int local = 0;
    local = 100;
    global = 100;
    int *ptr = (int *)arg;
    (*ptr) = 100;
    return NULL;
}

int main(int argc, char* argv[]) {
    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;
}
```
Example 5 – Stack, Heap, Global

- Kernel virtual memory
  - User stack 0 local
  - User stack 1 local
  - Shared libraries
  - Runtime heap
  - Read/write segment
  - Read-only segment
  - Global

- Memory invisible to user code
  - brk
  - Loaded from the executable file
  - Unused

- Process 1
  - write
  - read
Example 5 – Stack, Heap, Global

```c
int global = 0;

void* write(void* arg) {
    int local = 0;
    local = 100;
    global = 100;
    int *ptr = (int *)arg;
    (*ptr) = 100;
}

int main(int argc, char* argv[]) {
    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 output?

local 0 global 100 heap 100
Example 5 – Stack, Heap, Global

```c
int global = 0;

void* write(void* arg) {
    int local = 0;
    local = 100;
    global = 100;
    int *ptr = (int *)arg;
    (*ptr) = 100;
}

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

int main(int argc, char* argv[]) {
    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;
}
```

What are the output?
- local 0 global 0 heap 0
- local 0 global 100 heap 0
- local 0 global 100 heap 100
void* func(void* arg) {
    int p = *(int *)arg;
    p = p + 1;
    int *r = (void *)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 – Review

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;
}
```c
#define LEN 1000000000

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

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

Parallelize bigloop into two threads
Example 6 – bigloop

```c
#define LEN 1000000000

void* loop_thr1(void *arg){
    long *r = malloc(sizeof(long));
    int *arr = (int *)arg;

    for(int i = 0; i < LEN/2; i++)
        (*r) += arr[i];
    return (void *)r;
}

void* loop_thr2(void *arg){
    long *r = malloc(sizeof(long));
    int *arr = (int *)arg;

    for(int i = LEN/2; i < LEN; i++)
        (*r) += arr[i];
    return (void *)r;
}

Can we merge loop_thr1 with loop_thr2?
```

```c
int main() {
    int *arr = malloc(LEN * sizeof(int));
    ...
    pthread_t tid1, tid2;
    pthread_create(&tid, NULL, &loop_thr1, (void *)arr);
    pthread_create(&tid, NULL, &loop_thr2, (void *)arr);
    long *res1, *res2;
    pthread_join(tid, &res1);
    pthread_join(tid, &res2);
    printf("result is %ld\n", (*res1) + (*res2));
    }
```
Example 6 – bigloop

#define LEN 1000000000

typedef struct {
  int *arr;
  int len;
} loop_info;

int main() {
  int *arr = malloc(LEN * sizeof(int));
  ...
  pthread_t tids[2];
  for (int i = 0; i < 2; i++) {
    loop_info *info = (loop_info *)malloc(sizeof(loop_info));
    info->arr = arr + i * LEN/2;
    info->len = LEN/2;
    pthread_create(&tids[i], NULL, &loop, (void *)info);
  }
  for (int i = 0; i < 2; i++) {
    long *res;
    pthread_join(tids[i], &res);
    result += (*res);
  }

  void* loop(void *arg) {
    loop_info *info = (loop_info *)arg;
    long *r = malloc(sizeof(long));
    for (int i = 0; i < info->len; i++)
      (*r) += info->arr[i];
    return (void *)&r;
  }
}