Concurrency – Multithreading

Jinyang Li

based on slides by Tiger Wang
Example

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

int main() {
    ...
    long r = bigloop(arr, 1000000);
    ...
}
Parallelization

bigloop: 0→7

CPU0
CPU1
CPU2
CPU3
Parallelization

Performance can be improved by 4X
Concurrency

What's concurrency?
- things happening "simultaneously"
  1. multiple CPU cores concurrently executing instructions
  2. CPU and I/O devices concurrently doing processing

Why write concurrent programs?
- speed up programs using multiple CPUs
- speed up programs by concurrently doing 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
  - Elaborate 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
    • Elaborate to communicate with each other

Use multiple threads
Multiple threads (Multithreading)

Process

bigloop: 0→7

0
1
2
3
4
5
6
7

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

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

Process

thread 0
bigloop: 0\rightarrow 1

thread 1
bigloop: 2\rightarrow 3

thread 2
bigloop: 4\rightarrow 5

thread 3
bigloop: 6\rightarrow 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>bigloop: 0→1</td>
<td>bigloop: 2→3</td>
<td>bigloop: 4→5</td>
<td>bigloop: 6→7</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

CPU0
CPU1
CPU2
CPU3
Share the memory space

Different processes have different page tables
Share the memory space

Different processes have different page tables

Different threads of the same process share the same page table
Single threaded process

- Single-threaded process
- Memory invisible to user code
- %rsp (stack pointer)
- brk
- Loaded from the executable file
- Unused
- Read-only segment
- Read/write segment
- Runtime heap
- Shared libraries
- User stack
- Kernel memory

0x400000
0
Multi-threaded process

Each thread has its own stack
- Each thread has its own stack pointer
- Store stack pointer into a CPU core’s %rsp before running
Each thread has its own CPU state (registers, RFLAGS). It loads its CPU state to a CPU core’s registers before running.

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
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
#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;
    pthread_create(&tid, NULL, &func, NULL);
    return 0;
}
```

```bash
gcc create.c -lpthread
```

Process finishes when its main thread exits.
- All created threads are terminated
**pthread_join**

```c
#include <pthread.h>
int pthread_join(pthread_t thread_id, void **ret_ptr);
```

**Wait for the target thread to finish**
- Upon success, the return value of the target thread is stored at the location pointed to by `ret_ptr`.

**Return value**
- zero: success
- non-zero (error number): fail
Example 2 – Join

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

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

    pthread_t tid;
    pthread_create(&tid, NULL, &func, NULL);
    pthread_join(tid, NULL);
    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;
  pthread_create(&tid, NULL, &func, (void*)&param);
  ...

  int *res = NULL;
  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;
    pthread_create(&tid, NULL, &func, (void*)&param);
    ...

    int *res = NULL;
    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 = malloc(sizeof(int));
  *r = p;
  return (void *)r;
}

int main(int argc, char* argv[]) {
  int param = 100;

  pthread_t tid;
  pthread_create(&tid, NULL, &func, (void *)&param);
  ...

  int *res = NULL;
  pthread_join(tid, &res);
  ...

  printf("result: addr %lx val %d\n", res, *res);
  return 0;
}
```
void* func(void* arg) {
    int p = *(int *)arg;
    p = p + 1;
    int *r = malloc(sizeof(int));
    *r = p;
    return (void *)r;
}

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

    int param = 100;

    pthread_t tid;
    pthread_create(&tid, NULL, &func, (void *)&param);
    ...

    int *res = NULL;
    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
int global = 0;

void* write(void* arg) {
    int local = 100;
    global = 100;
    *(int *)arg = 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;
}
Example 5 – Stack, Heap, Global

Process 1

- User stack 0 (local) with sp0
- User stack 1 (local) with sp1
- Shared libraries
- Runtime heap with *p
- Global variables
- Read/write segment
- Read-only segment
- Unused

Memory invisible to user code

Loaded from the executable file

brk

write
read
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;
}

What are the output?

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

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 0 heap 0
local 0 global 100 heap 0
local 0 global 100 heap 100
Example 6 – bigloop

```
#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;
}

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));
}

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?
#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;
}