并行编译与优化
Parallel Compiler and Optimization

计算机研究所编译系统室 方建滨

Lecture 18 Threads Implementation

第十八课 多线程实现

2024-06-11

学习内容

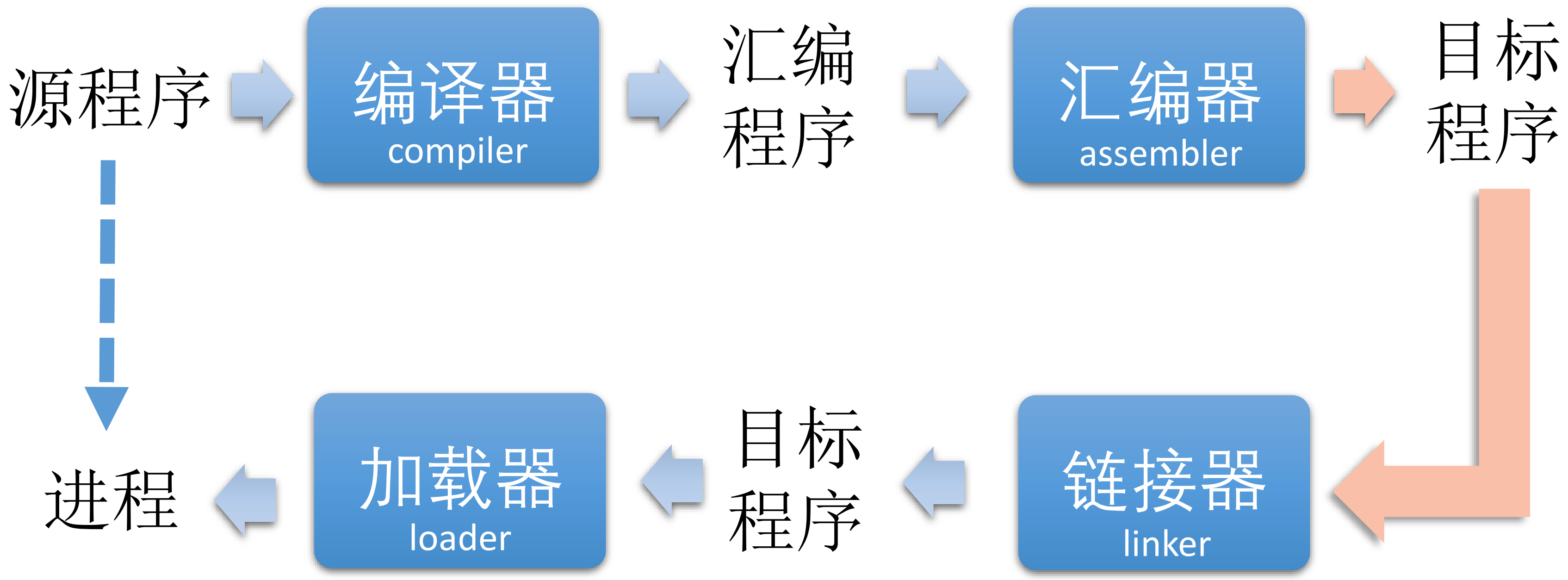
- 1. 多进程
  - 1.1 进程概念
  - 1.2 进程操作
  - 1.3 进程状态
  - 1.4 进程调度
  - 1.5 进程间通信
- 2. 多线程
  - 2.1 线程概念
  - 2.2 多核与多线程
  - 2.3 Pthreads编程
  - 2.4 自己动手实现线程

进程 vs. 程序

- 提问：什么是程序?
- 进程是一个执行程序的实例
- 两者关系
  - 用一个程序可以创建多个进程
  - 多个进程可以同时运行同一道程序

从程序到进程

源程序代码成为进程的过程



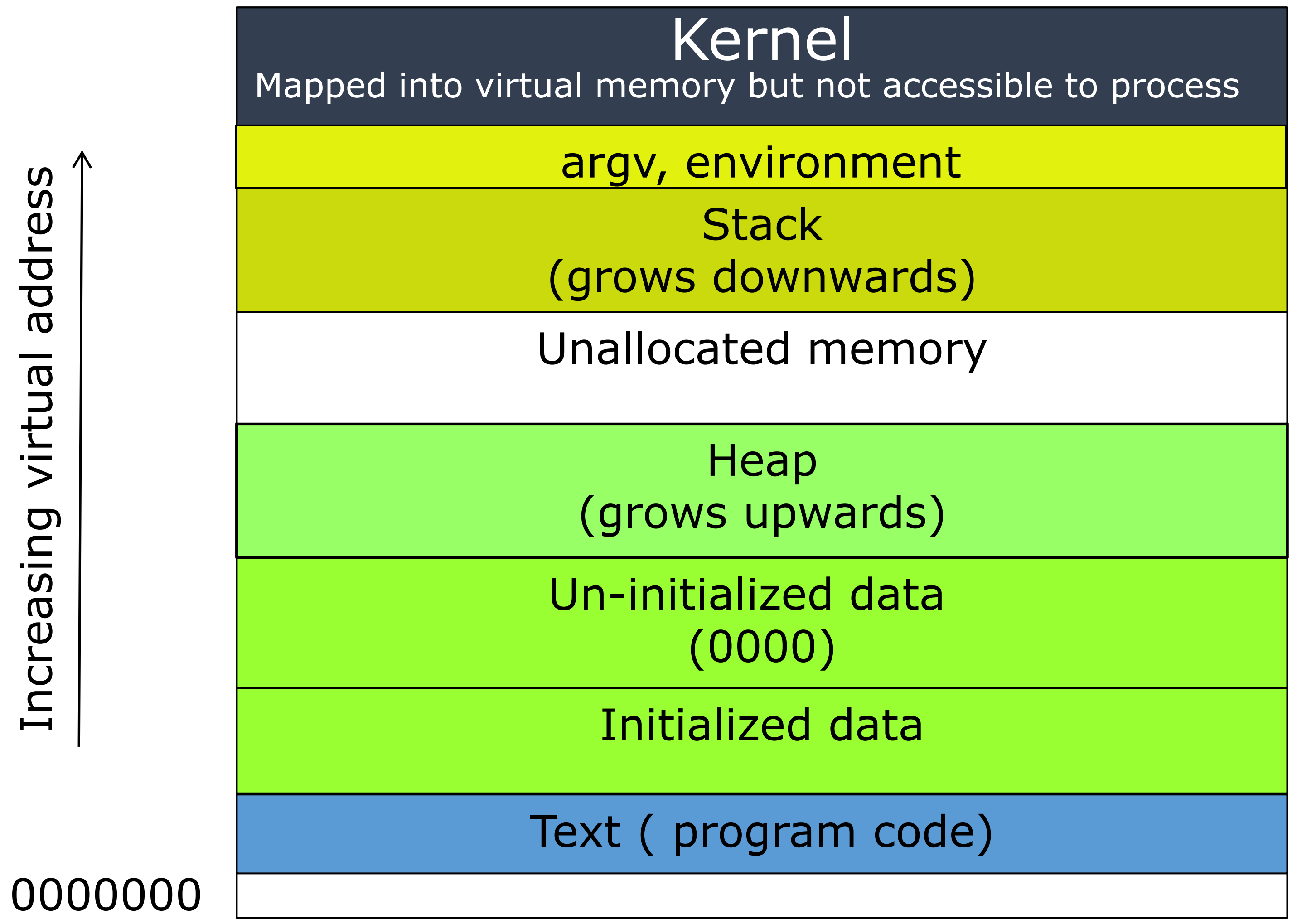
从操作系统视角看进程

- **在一个进程启动时，OS的工作包括：**
  - **将程序加载到内存（由加载器完成）**
  - **为程序数据分配内存**
  - **在OS内核中记录进程相关信息**

Process ID	Process State
Priority	Address of executing instruction
User IDs	Address of return instruction

内存中的进程

- 以段(segments)的形式为进程分配内存

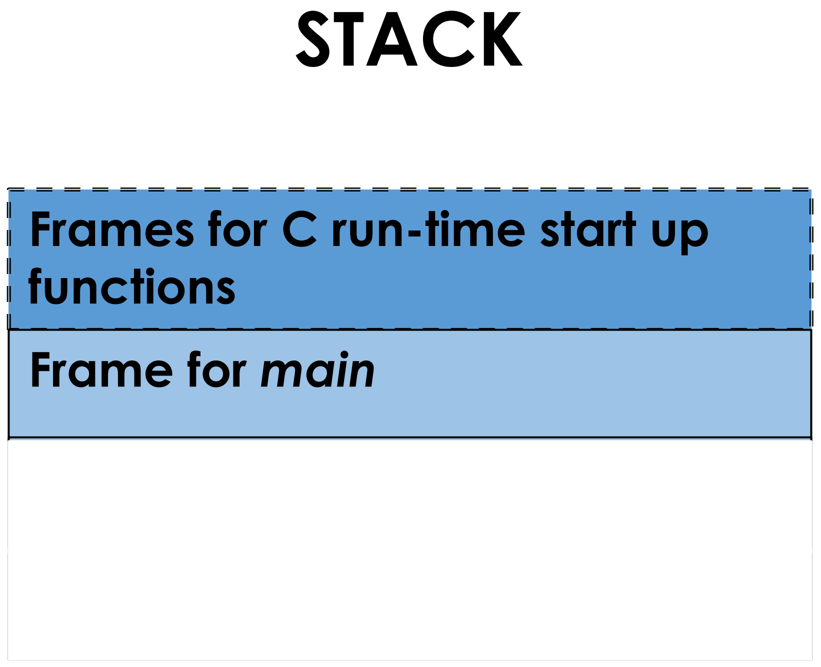


栈 (Stack)

- 栈包含栈帧(stack frames)并能够动态伸缩的段
- 栈帧(stack frame)
  - 为管理单个函数数据而分配：存放函数的局部变量、参数及返回值
  - 知道如何返回调用者函数（caller vs. callee)
  - 按照先进后出的方式进行管理
- 栈指针寄存器(stack pointer)
  - 用于追踪当前栈顶的特殊寄存器
- 发生函数调用时在栈中创建一个新栈帧；当函数返回时移除栈帧

栈与栈帧

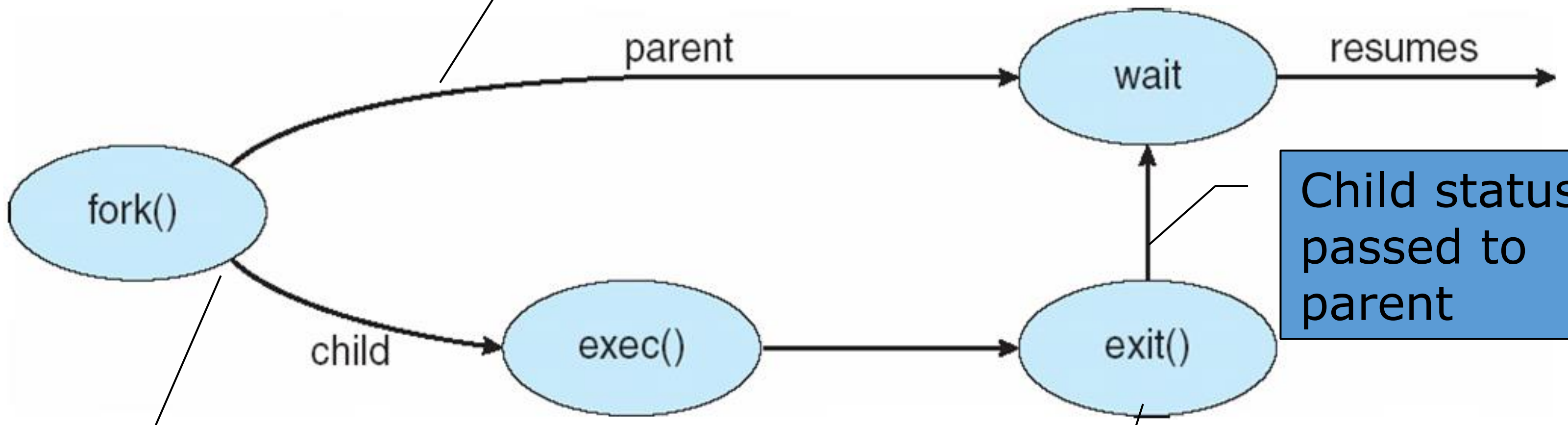
```
square (int x){
  return x*x;
}
doCalc(int val){
  printf ("square is %d\n",
          square(val));
}
main (int x, int y){
  key = 9999;
  doCalc (key);
}
```



进程创建

- 父进程创建子进程，子进程接着创建其它进程，从而形成一棵进程树
- 以Unix进程为例
  - 系统调用fork可创建与父进程几乎一样的子进程
  - 系统调用exec在fork后使用，并使用新程序替换进程的内存空间

进程创建



Parent my perform other actions here

Child statuses passed to parent

Memory of parent copied to child

Kernel restarts parent

C Program Forking Separate Process

```
int main(){
     int  pid;
   pid = fork();  /* fork another process */
   if (pid < 0) {  /* error occurred */
          fprintf(stderr, "Fork Failed");
          exit(-1); }
   else if (pid == 0) {  /* child process */
          execlp("/bin/ls", "ls", NULL);       }
   else {  /* parent process */
          /* parent will wait for the child to complete */
          wait (NULL);
          printf ("Child Complete!\n");
          exit(0);          }
}
```

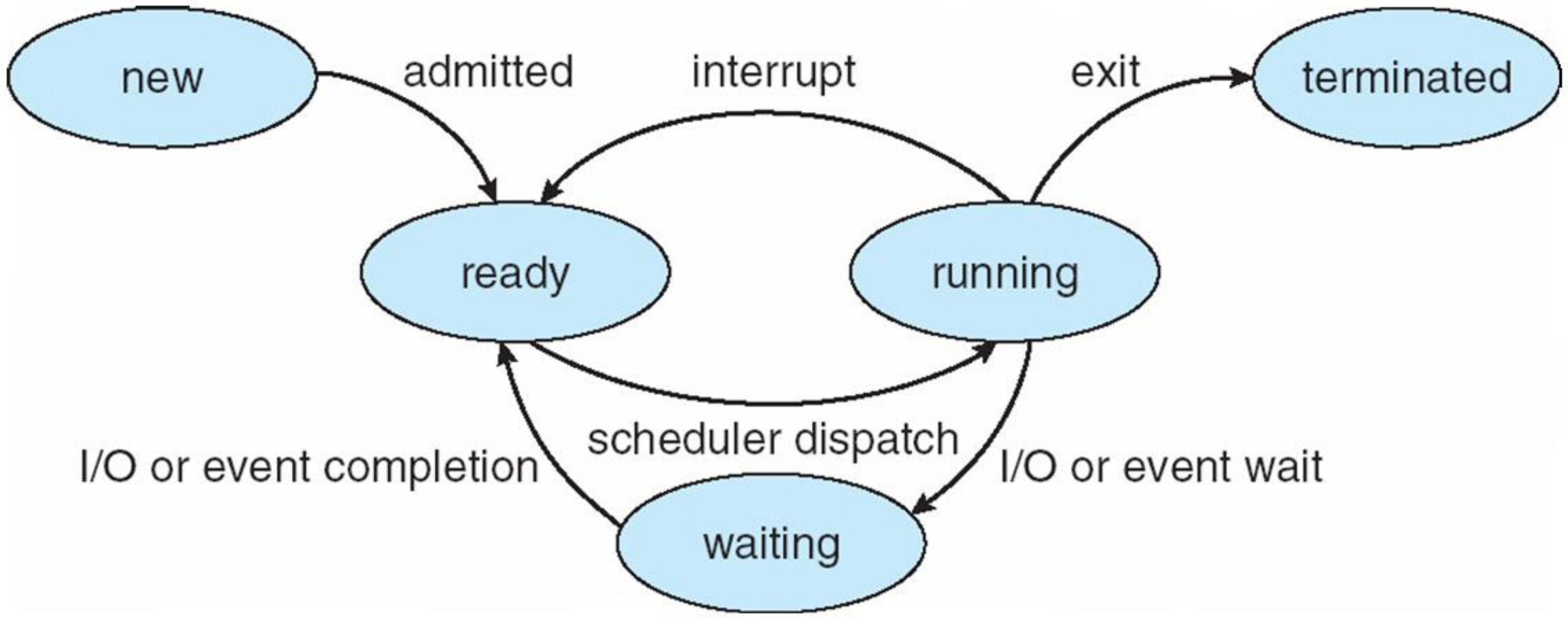
进程终止

- 进程执行最后一条语句后，通过exit将自身终止
  - 将数据输出给父进程 (通过wait)
  - 进程资源被OS释放
- 父进程可通过abort终止子进程的执行
  - 当子进程使用超过了其所分配的资源时
  - 当分配给子进程的任务不再需要时
  - 级联终止 (cascade termination): 在一些OS上，当发生父进程退出时，子进程继续执行是不被允许的

进程状态

- 当一个进程执行时，它在以下状态间转换
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: Waiting for some event to occur
  - ready: Waiting to be assigned to a processor
  - terminated: The process has finished execution

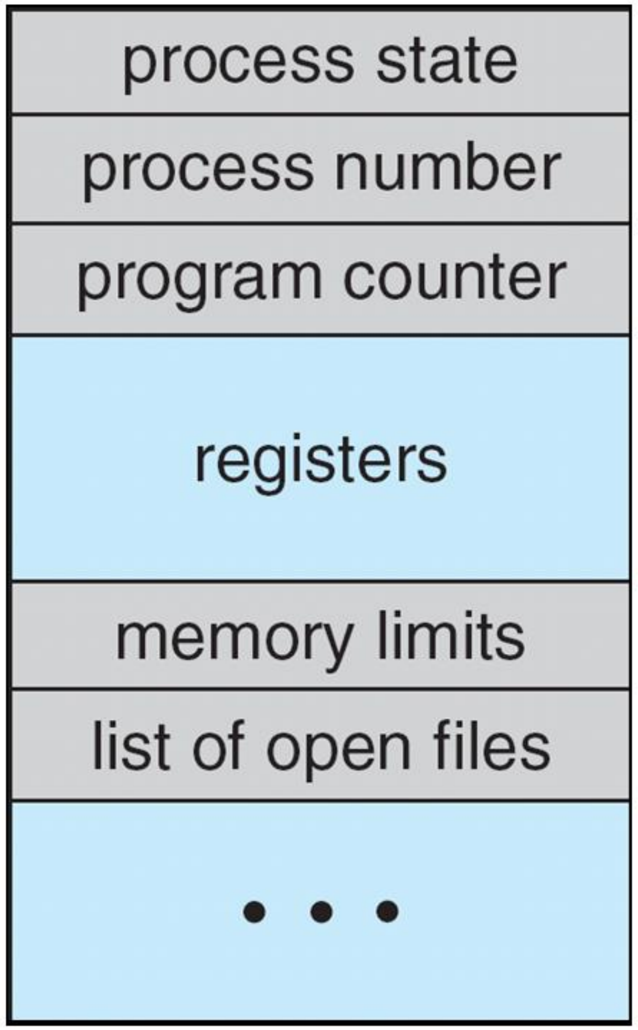
进程状态转换图


new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

进程控制块

- 进程控制块 (Process Control Block)
  - Process ID
  - Program Counter
  - CPU Registers
  - CPU scheduling information
  - Priority
  - Process state
  - Memory management information
  - Accounting information
  - List of I/O devices allocated to process

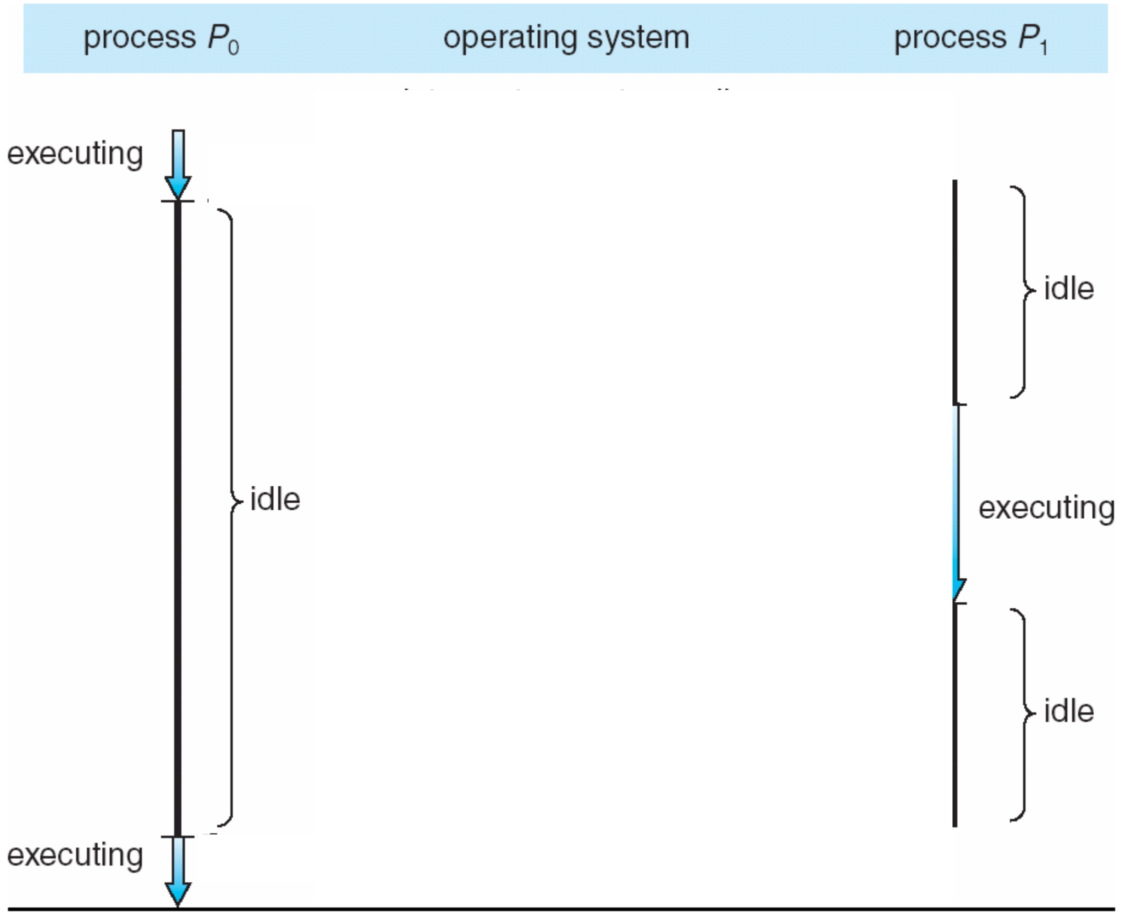
进程控制块


process state
process number
program counter
registers
memory limits
list of open files
• • •

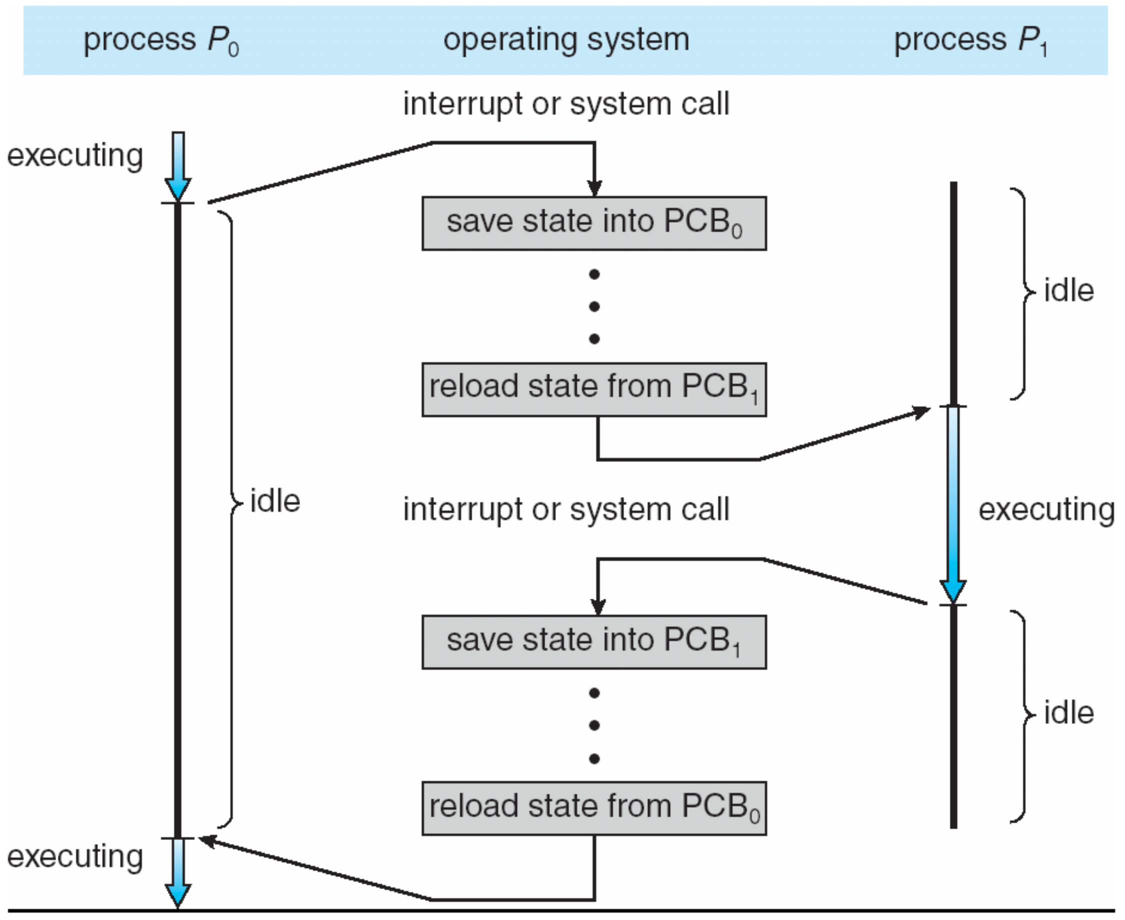
上下文切换

- 当CPU切换到另外一个进程时，系统通过上下文切换(context switch)保存老进程的状态并加载新进程的状态
- 一个进程的上下文表示为PCB
- 上下文切换时系统没有做有用的工作，故是一个开销
- 具体的切换开销取决于硬件支持

上下文切换


process P_0
operating system
process P_1
executing
idle
executing
idle
executing
idle

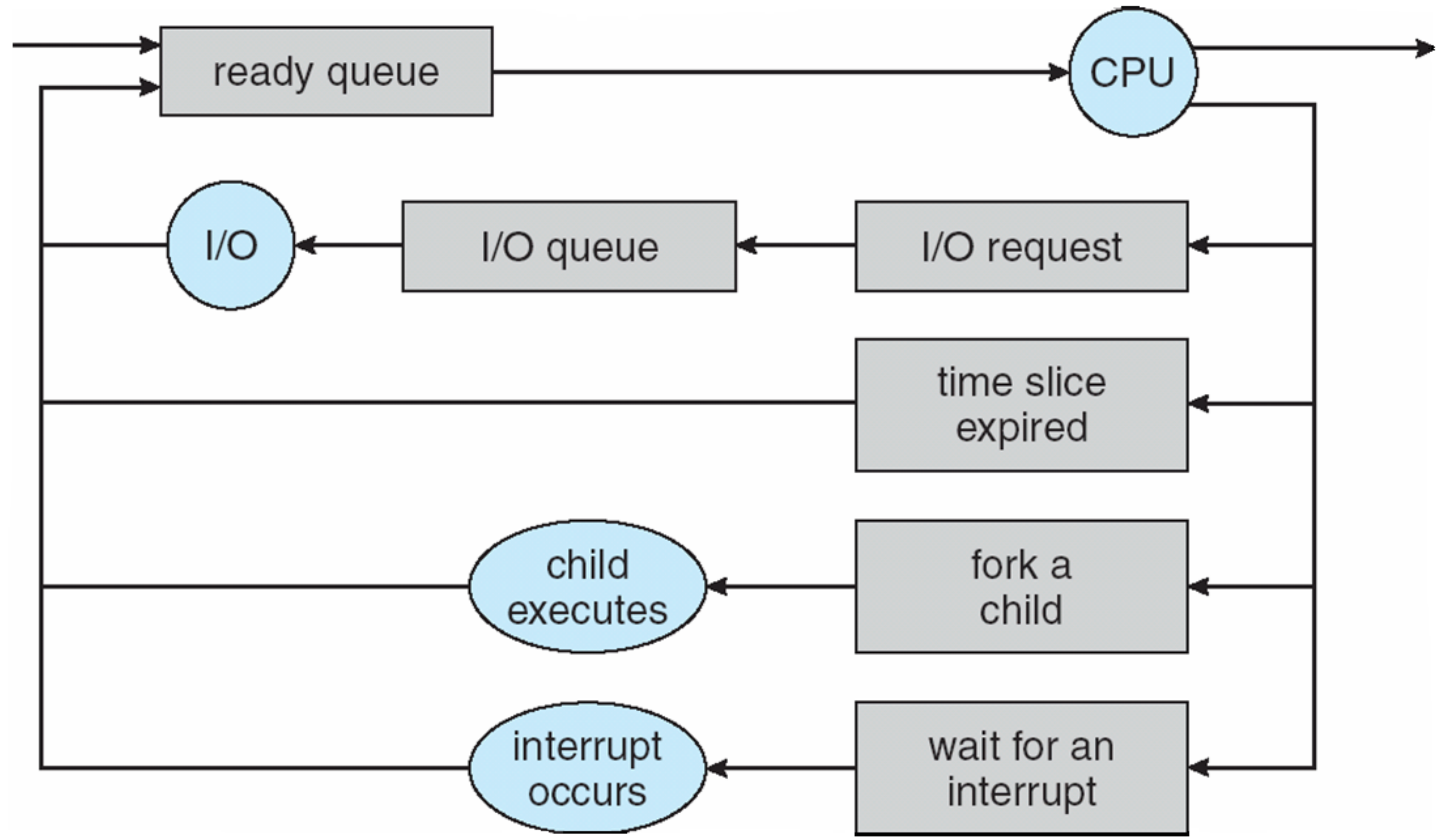
上下文切换


process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
idle
reload state from PCB_1
idle
interrupt or system call
executing
save state into PCB_1
idle
reload state from PCB_0
executing

进程调度

- 为使CPU处于忙碌状态，OS支持多道程序，提高资源利用率
- 进程调度的动机
- 进程调度器
  - 从所有进程中选择一个准备好的进程进入程序执行
  - 抢占式：迫使一个进程进入空闲状态，使另外一个进程得以执行

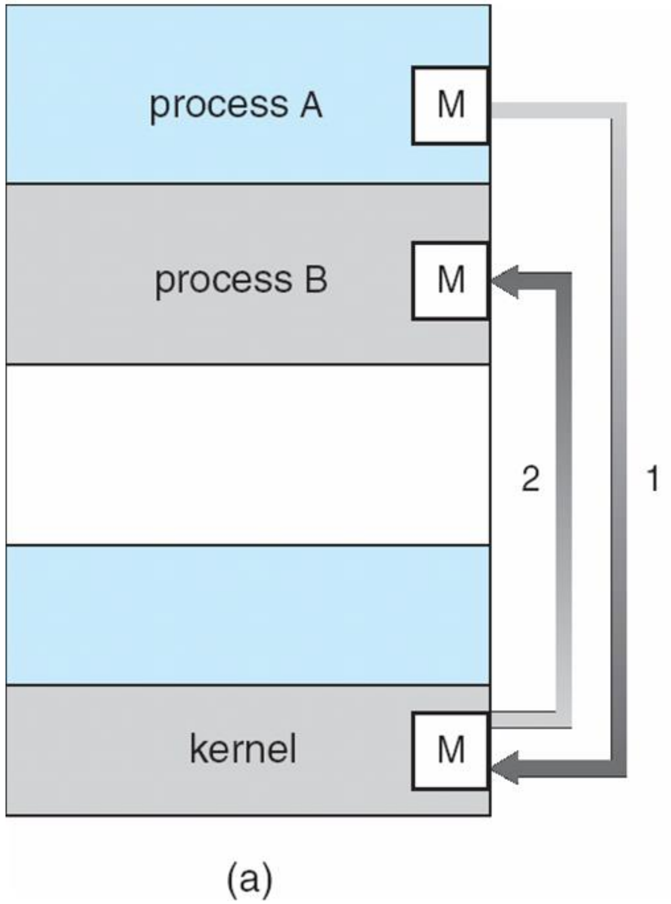
进程调度过程


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

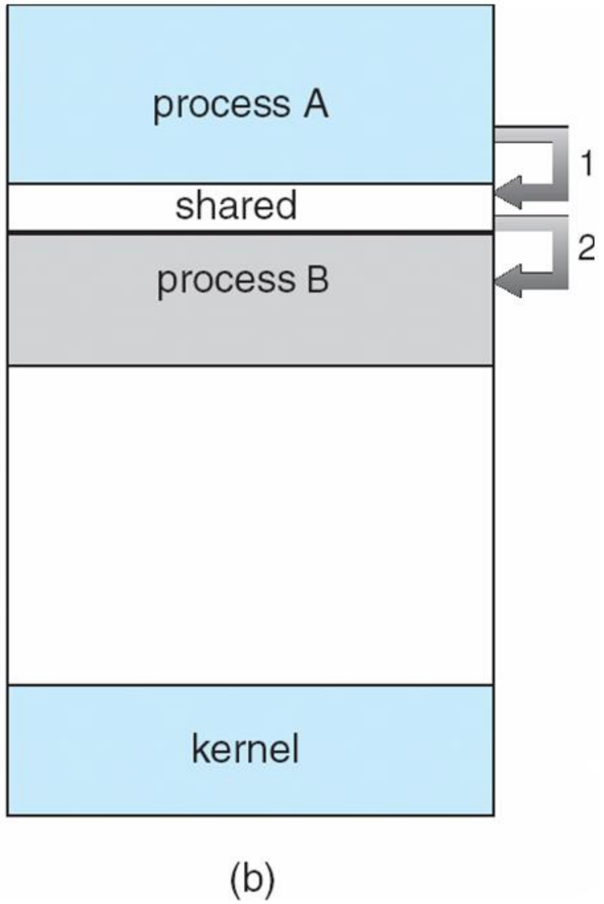
进程间通信

- 多个进程相互合作完成一个任务
  - 加速计算
  - 信息共享
- 合作进程需要进程间通信(IPC)
- IPC的两种模式：消息传递和共享内存

通信模式



Message Passing



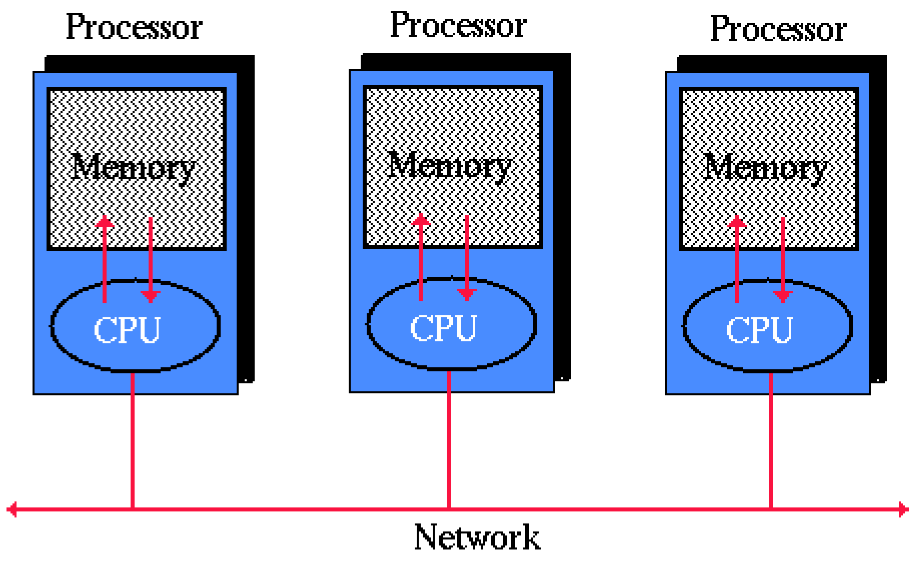
Shared Memory

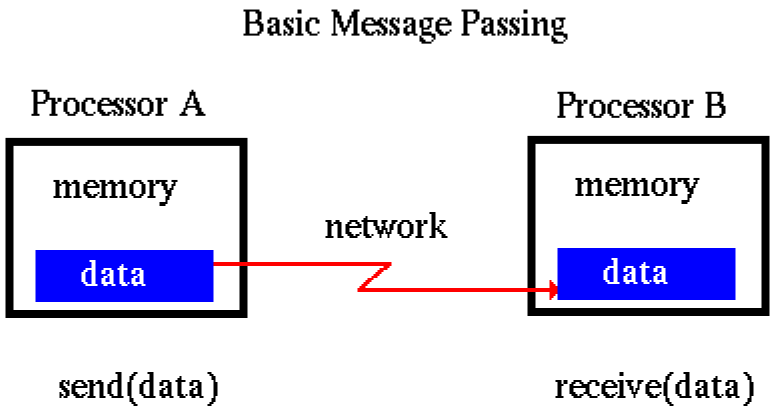
消息传递

- 用于进程间通信、同步的机制
- 消息系统：不借助于共享变量实现进程间通信
- 两个基本操作
  - send(message)
  - receive(message)
- 如果进程P、Q要通信，那么需要
  - 在它们之间建立一条通信链路
  - 通过send/receive交换消息

消息传递

- 消息传递发生在不同的处理器之间
- MPI (Message Passing Interface)
  - 超算系统上结点间并行编程的事实标准
  - 在每个计算结点创建一个或多个进程
  - 实现为库(library)的形式





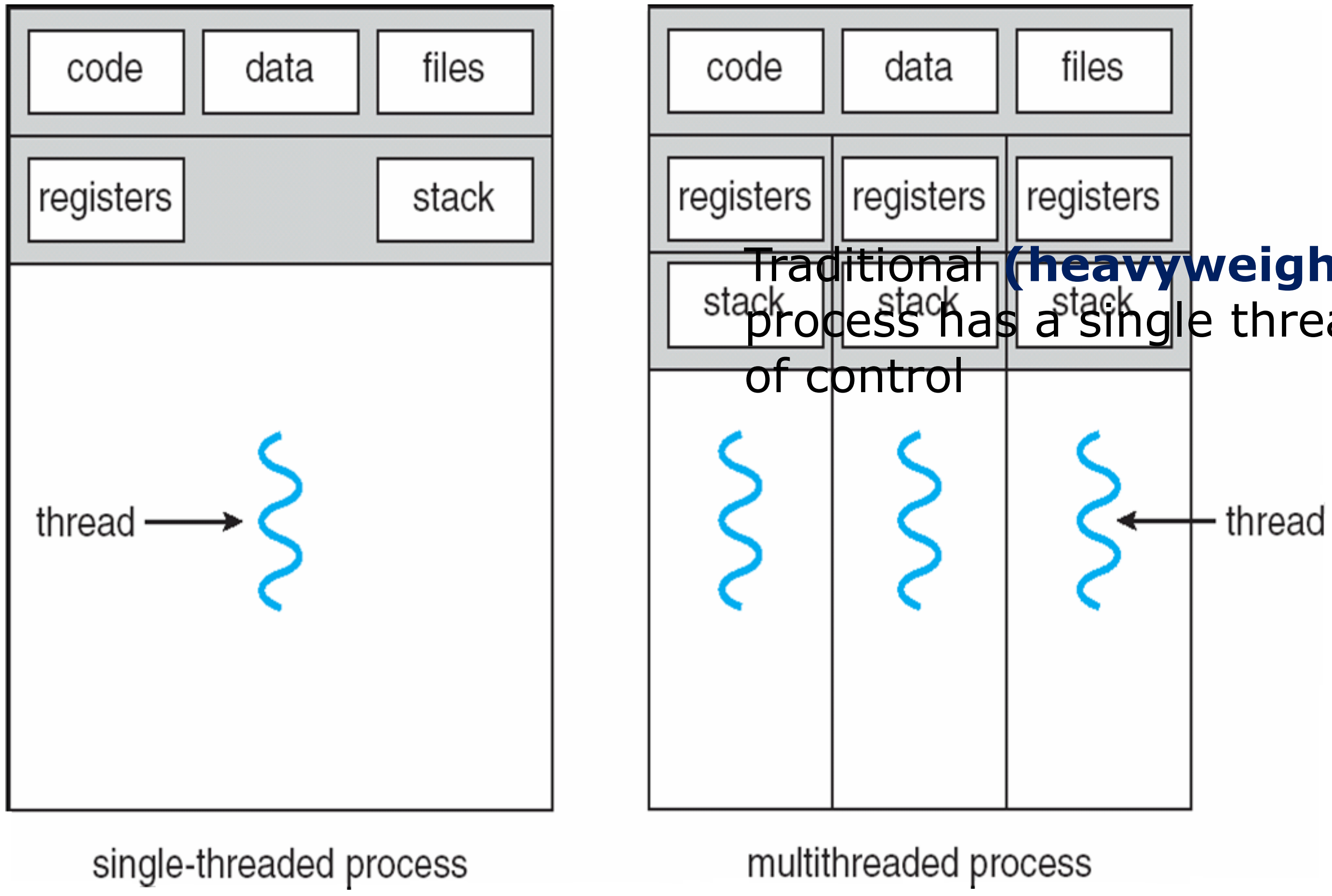
学习内容

- 1. 多进程
  - 1.1 进程概念
  - 1.2 进程操作
  - 1.3 进程状态
  - 1.4 进程调度
  - 1.5 进程间通信
- 2. 多线程
  - 2.1 线程概念
  - 2.2 多核与多线程
  - 2.3 Pthreads编程
  - 2.4 自己动手实现线程

线程概述

- 线程是一种允许应用并发执行多个任务的机制
- 线程是使用CPU的基本单位
  - Thread ID
  - Program counter
  - Register set
  - Stack
- 一个进程可以包含多个线程
  - 同一个进程的多个线程共享信息，如代码、数据、打开文件等

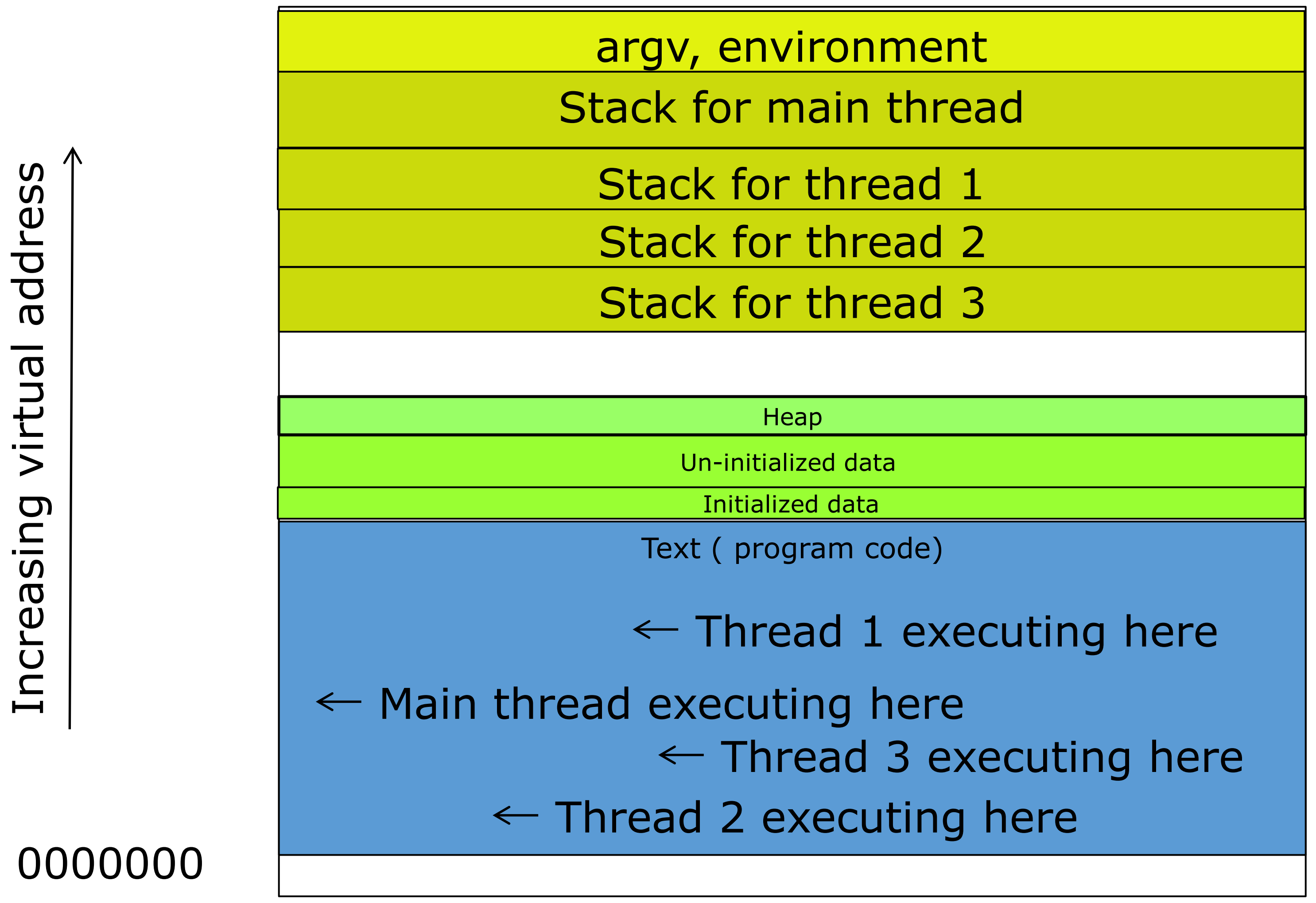
单线程进程与多线程进程



Traditional **(heavyweight)** process has a single thread of control

内存中的线程

- 以段(segments)的形式为进程开辟内存



共享属性与私有属性

线程共享

- Global memory
- Process ID and parent process ID
- Controlling terminal
- Process credentials (user)
- Open file information
- Timers …

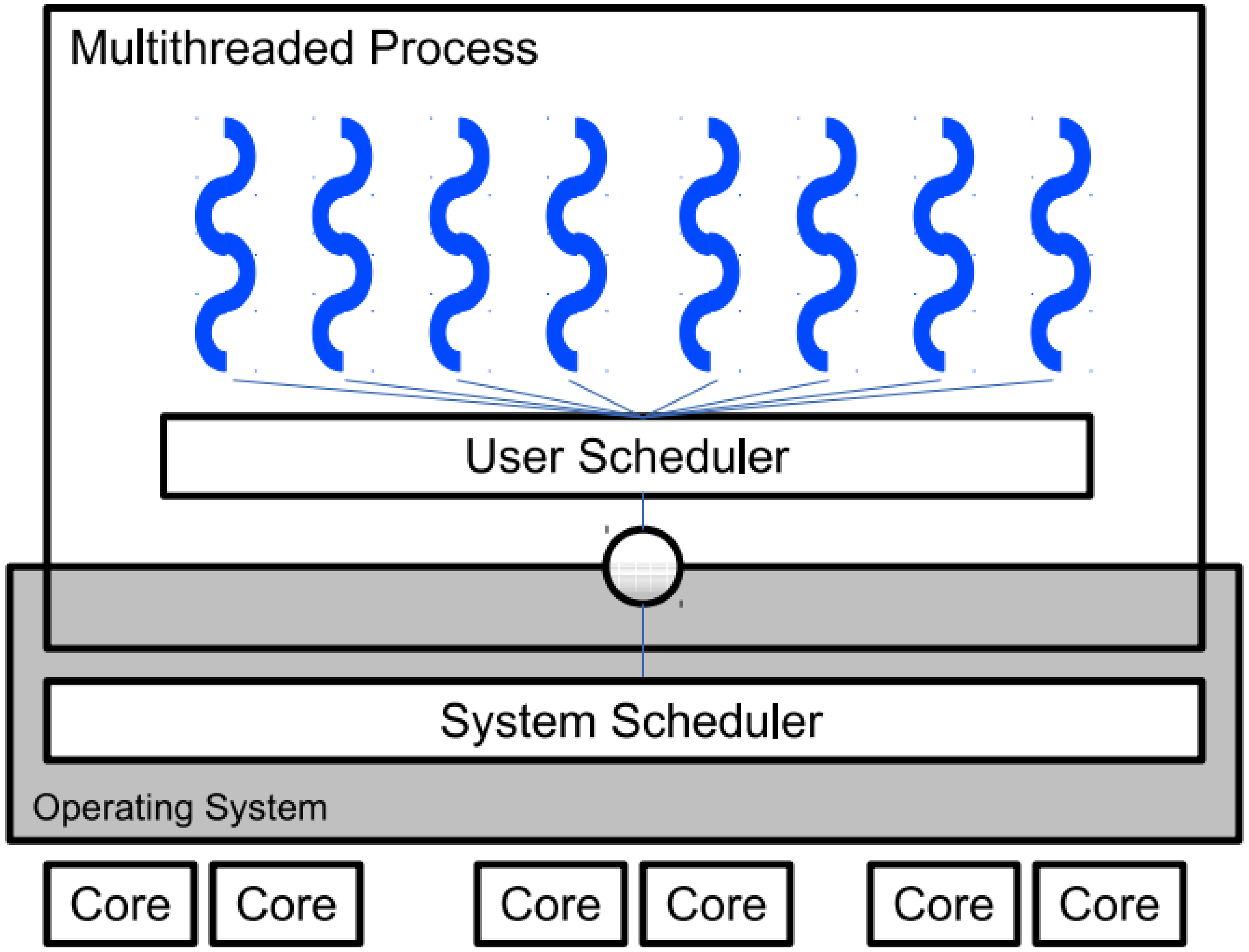
线程私有

- Thread ID
- Thread specific data
- CPU affinity
- Stack (local variables and function call linkage information) …

多核与多线程

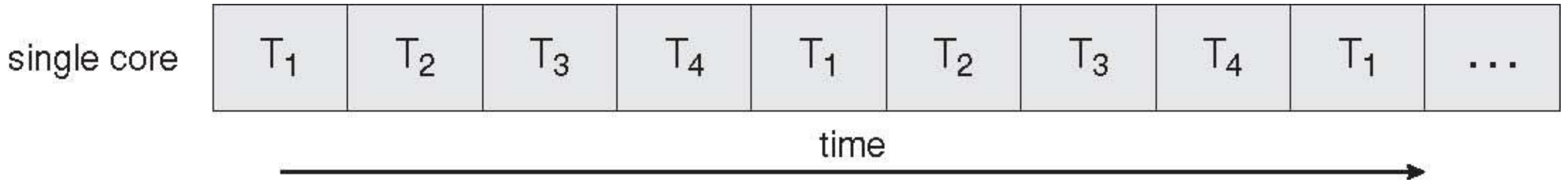
- 硬件线程通常被认为是物理CPU或核
  - 超线程也经常被看作硬件线程
- 软件线程是OS对物理处理器的抽象
  - 一个硬件线程可以运行多个软件线程
  - 由OS进行调度并切换线程

多核与多线程

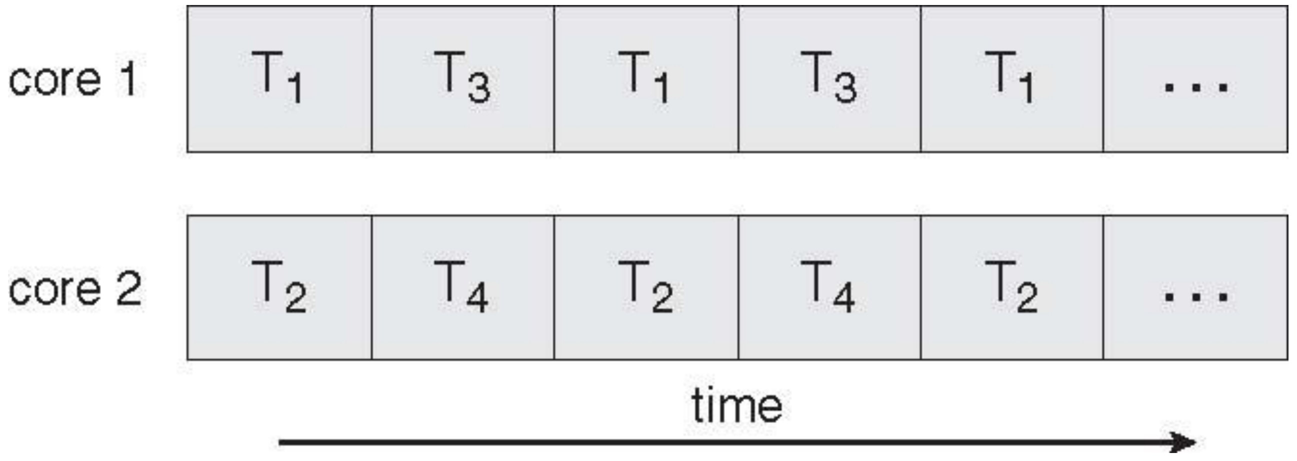

Multithreaded Process
User Scheduler
System Scheduler
Operating System
Core
Core
Core
Core
Core
Core

多核与多线程

Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



多线程编程支持

- **Threads as a programming abstraction**
  - **Dynamically create/terminate threads**
  - **Communicate among threads**
  - **Synchronize activities of threads**
- **Typical thread libraries**
  - **POSIX Pthreads**
  - **Win32 threads**
  - **hthreads (heterogeneous threads) for Matrix-3000**

Jianbin Fang, Peng Zhang, Chun Huang, Tao Tang, Kai Lu, Ruibo Wang, Zheng Wang: Programming bare-metal accelerators with heterogeneous threading models: a case study of Matrix-3000. Frontiers Inf. Technol. Electron. Eng. 24(4): 509-520 (2023)

hthreads主机端运行时

- **hthread_host.h**
  - **功能上分为四类**
    - ◆设备管理
    - ◆线程组管理
    - ◆数据管理
    - ◆设备共享资源管理

```
int hthread_dev_open(int cluster_id);
int hthread_dev_close(int cluster_id);
int hthread_dat_load(int cluster_id, char *file_path);
int hthread_dat_unload (int cluster_id);

int hthread_group_create(int cluster_id, int num,
        char *func_name, int scalar_args,
        int ptr_args, uint64_t *arg_array);
int hthread_group_create_masked(int cluster_id,
        unsigned int pmask, char *func_name,
        int scalar_args, int ptr_args, uint64_t *arg_array);
int hthread_group_exec(int gid, char *func_name,
        int scalar_args,  int ptr_args, uint64_t *arg_array);
int hthread_group_get_status(int gid);
int hthread_group_wait(int thread_id);
int hthread_group_destroy(int thread_id);
int hthread_get_avail_threads(int cluster_id);

void *hthread_malloc(int cluster_id, int bytes, int mode);
void hthread_free(void *ptr);

int hthread_barrier_malloc(int cluster_id);
void hthread_barrier_free(int b_id);
int hthread_rwlock_malloc(int cluster_id);
void hthread_rwlock_free(int lock_id);
```

hthreads设备端运行时

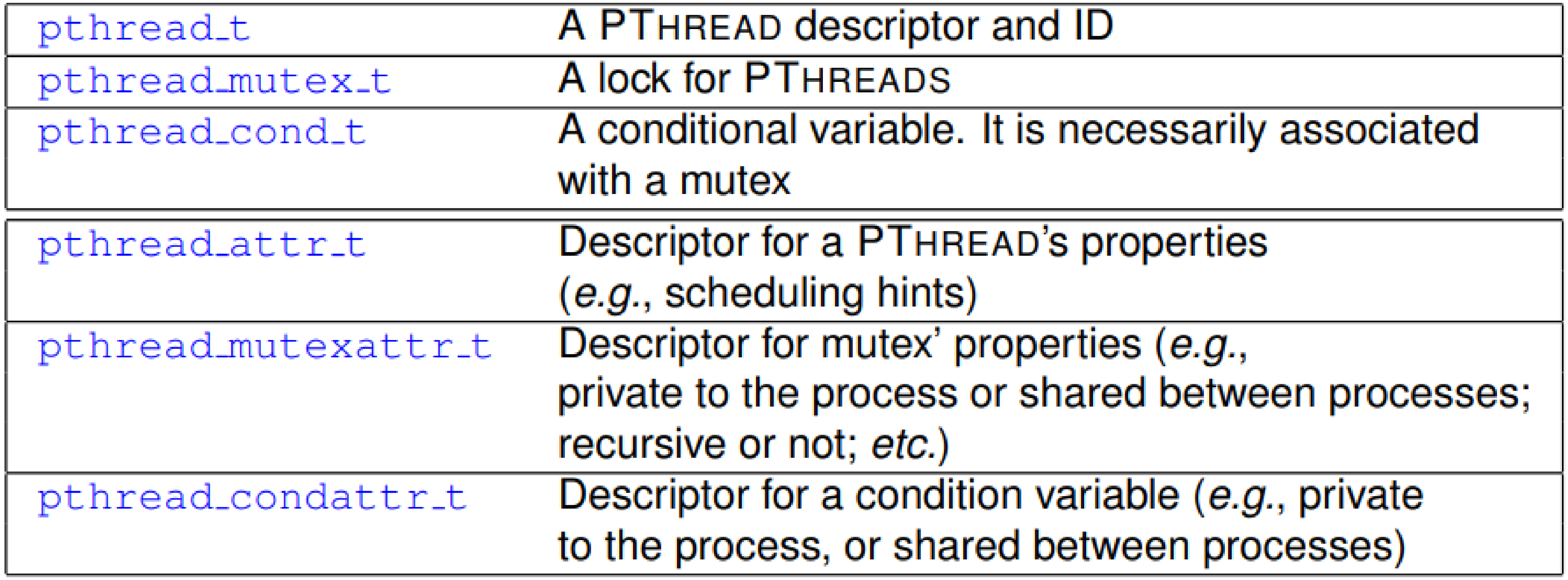
- **hthread_device.h**
  - **功能上分为五类**
    - ◆并行管理
    - ◆同步管理
    - ◆私有存储管理
    - ◆DMA传输
    - ◆系统调用

```
int get_group_size();
int get_thread_id();

void group_barrier(unsigned int b_id);
void rwlock_rdlock(unsigned int lock_id);
void rwlock_wrlock(unsigned int lock_id);
void rwlock_unlock(unsigned int lock_id);

void * vector_malloc(unsigned int bytes );
int vector_free(void *ptr);
int vector_load(void *mem, void *buf, unsigned int bytes);
int vector_store(void *buf, void *mem, int bytes);
int vector_load_async(void *mem, void *buf, int bytes);
int vector_store_async(void *buf, void *mem, int bytes);

void dsp_abort(int err_no);
void dsp_halt();
void hthread_printf(const char *fmt, …);

unsigned int dma_p2p(void *src, unsigned long src_row_num, unsigned int
src_row_size, …);
unsigned int dma_broadcast(void *src, unsigned long src_row_num, unsigned int
src_row_size, …);
unsigned int dma_segment(void *src, unsigned long src_row_num, unsigned int
src_row_size, …);
unsigned int dma_sg(void *src_base, void *src_index, unsigned long
src_row_num, …);
void dma_wait(int channelNo);
```

Pthread

- IEEE POSIX 1003.1c-1995线程标准--Pthread
  - Sun Solaris 2.5, Silicon Graphics IRIX 6, IBM AIX, Linux
- 线程管理
  - 使用**线程库**来管理线程
- 线程同步
  - 互斥(mutex)变量
  - 条件(cond)变量

基本数据类型



pthread_t	A PTHREAD descriptor and ID
pthread_mutex_t	A lock for PTHREADS
pthread_cond_t	A conditional variable. It is necessarily associated with a mutex
pthread_attr_t	Descriptor for a PTHREAD's properties (*e.g.*, scheduling hints)
pthread_mutexattr_t	Descriptor for mutex' properties (*e.g.*, private to the process or shared between processes; recursive or not; *etc.*)
pthread_condattr_t	Descriptor for a condition variable (*e.g.*, private to the process, or shared between processes)

线程管理

- 基本线程管理原语
  - 创建线程 pthread_create
  - 中止线程 pthread_exit
  - 等待其它线程中止 pthread_join
  - 获取当前线程id pthread_self

线程管理

创建线程

Asynchronously invoke `thread_function` in a new thread

```
#include <pthread.h>
int pthread_create(
   pthread_t *thread_handle,  /* returns handle here */
   const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);  /* single argument; perhaps a structure */
```

`attribute` created by `pthread_attr_init`

contains details about
- whether scheduling policy is inherited
- scheduling parameters
- stack size, stack guard size

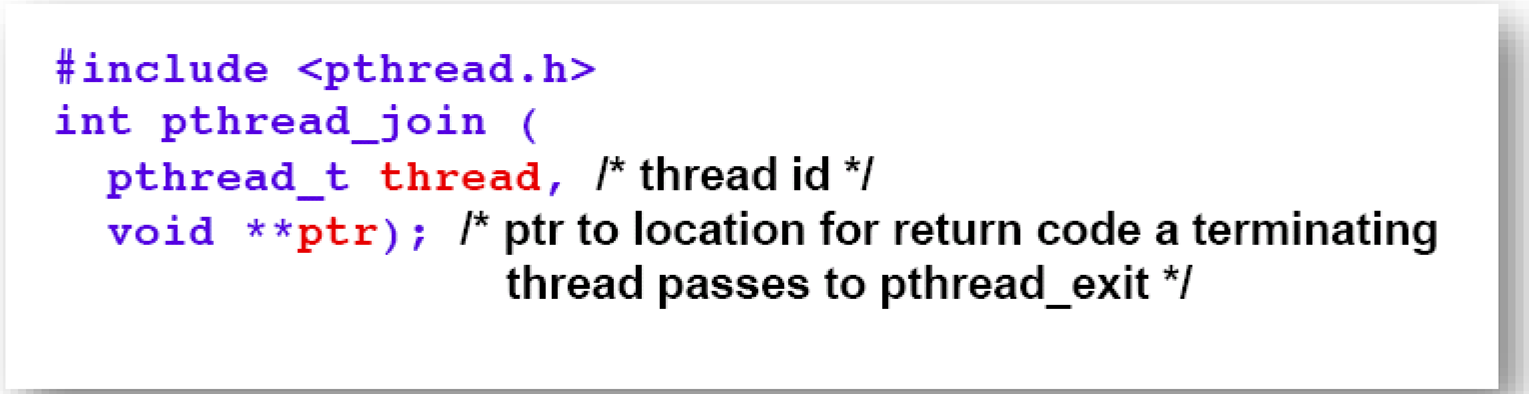
线程管理

中止线程

```
#include <pthread.h>
void pthread_exit (void *retval)
```

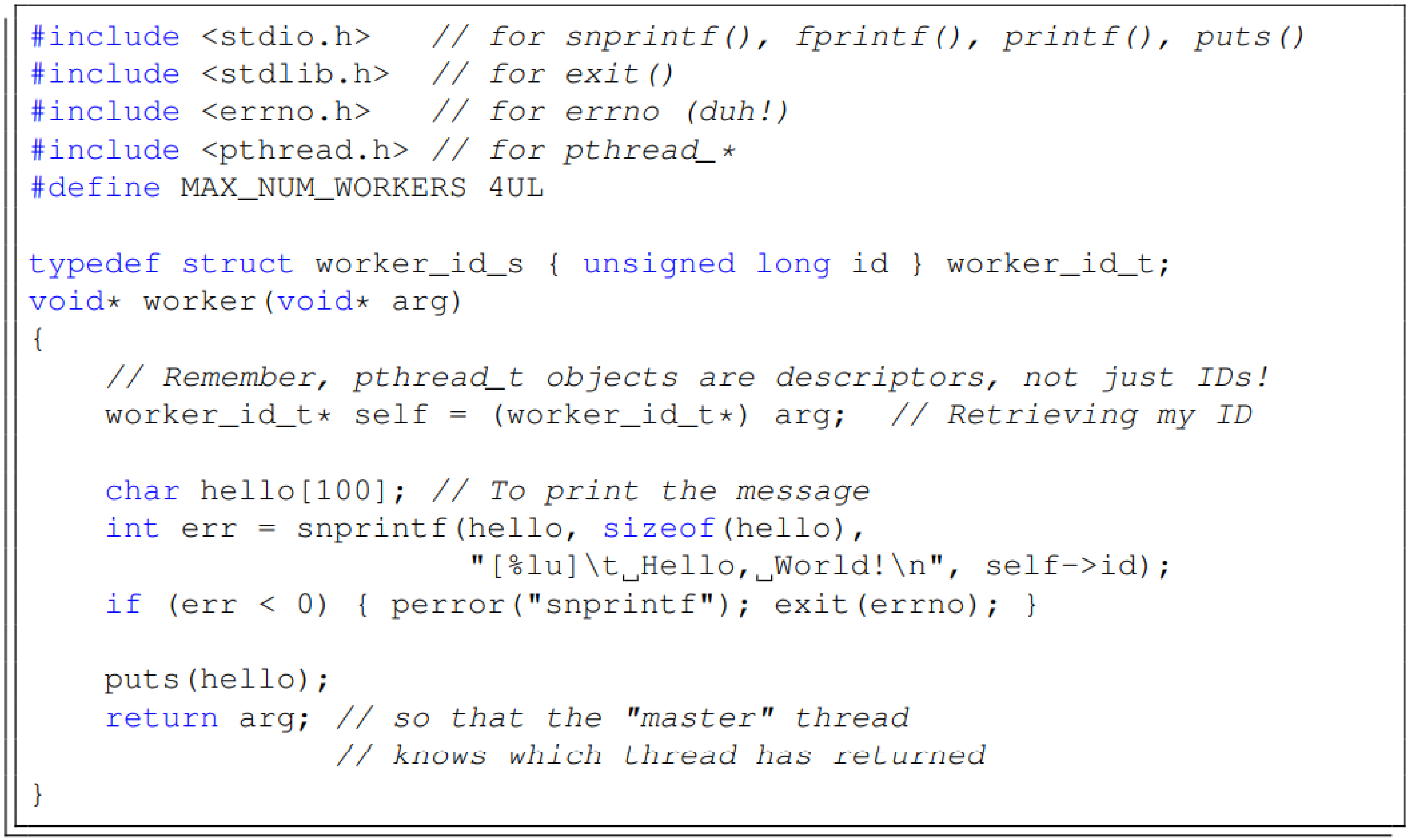
线程管理

- 调用pthread_join的线程挂起，直到指定线程中止



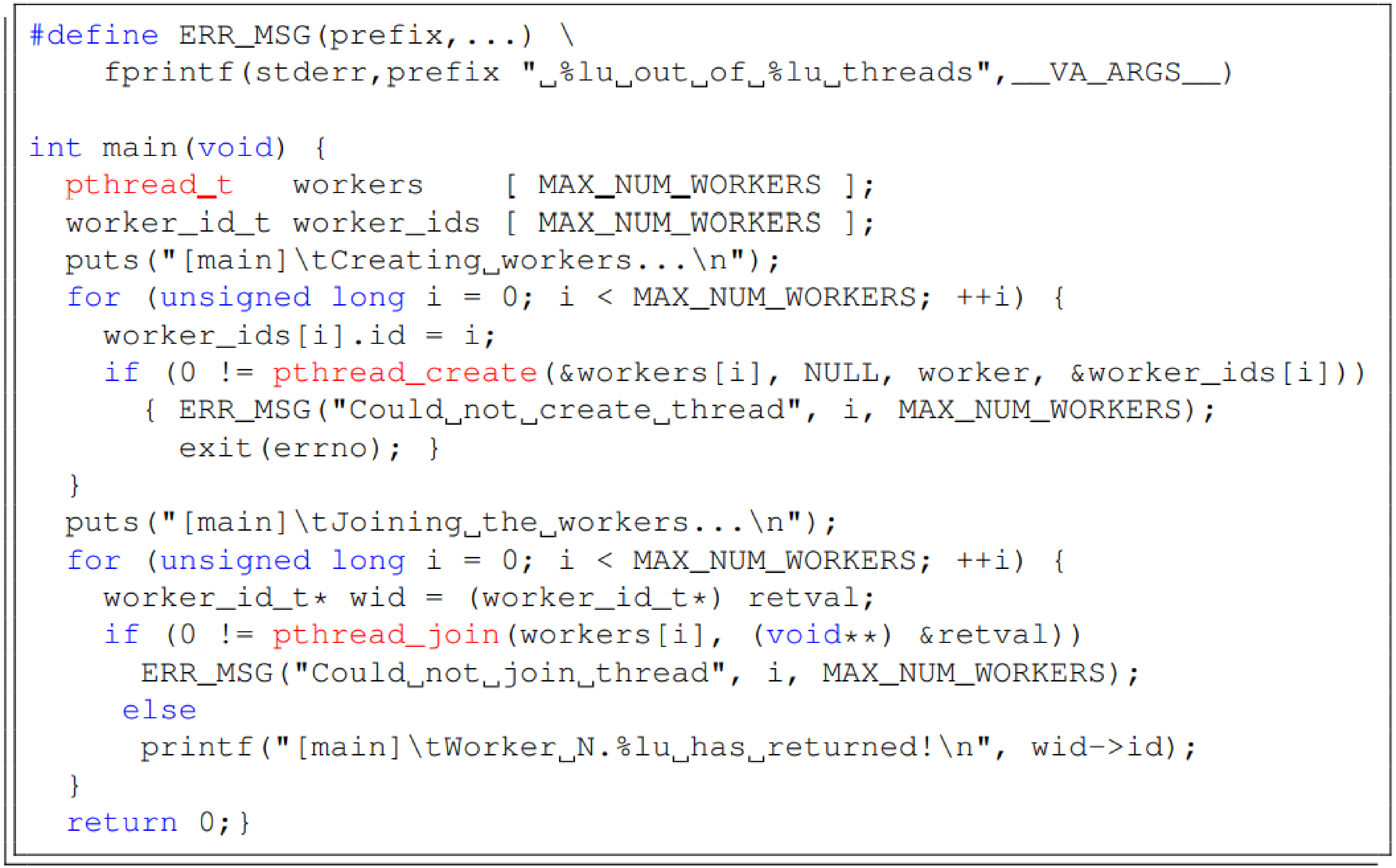
```
#include <pthread.h>
int pthread_join (
  pthread_t thread,  /* thread id */
  void **ptr);  /* ptr to location for return code a terminating
                   thread passes to pthread_exit */
```

A First Pthreads Example: worker



```c
#include <stdio.h>   // for snprintf(), fprintf(), printf(), puts()
#include <stdlib.h>  // for exit()
#include <errno.h>   // for errno (duh!)
#include <pthread.h> // for pthread_*
#define MAX_NUM_WORKERS 4UL

typedef struct worker_id_s { unsigned long id } worker_id_t;
void* worker(void* arg)
{
    // Remember, pthread_t objects are descriptors, not just IDs!
    worker_id_t* self = (worker_id_t*) arg;  // Retrieving my ID

    char hello[100]; // To print the message
    int err = snprintf(hello, sizeof(hello),
                       "[%lu]\t␣Hello,␣World!\n", self->id);
    if (err < 0) { perror("snprintf"); exit(errno); }

    puts(hello);
    return arg; // so that the "master" thread
                // knows which thread has returned
}
```

A First Pthreads Example: main



```
#define ERR_MSG(prefix,...) \
    fprintf(stderr,prefix "␣%lu␣out␣of␣%lu␣threads",__VA_ARGS__)

int main(void) {
  pthread_t   workers     [ MAX_NUM_WORKERS ];
  worker_id_t worker_ids [ MAX_NUM_WORKERS ];
  puts("[main]\tCreating␣workers...\n");
  for (unsigned long i = 0; i < MAX_NUM_WORKERS; ++i) {
    worker_ids[i].id = i;
    if (0 != pthread_create(&workers[i], NULL, worker, &worker_ids[i]))
      { ERR_MSG("Could␣not␣create␣thread", i, MAX_NUM_WORKERS);
        exit(errno); }
  }
  puts("[main]\tJoining␣the␣workers...\n");
  for (unsigned long i = 0; i < MAX_NUM_WORKERS; ++i) {
    worker_id_t* wid = (worker_id_t*) retval;
    if (0 != pthread_join(workers[i], (void**) &retval))
      ERR_MSG("Could␣not␣join␣thread", i, MAX_NUM_WORKERS);
     else
      printf("[main]\tWorker␣N.%lu␣has␣returned!\n", wid->id);
  }
  return 0;}
```

A First Pthreads Example: output

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c hello.c
gcc -o hello hello.o -lpthread
```

. . . Don't forget to link with the PTHREAD library!

. . . And the output:

Output of `./hello`

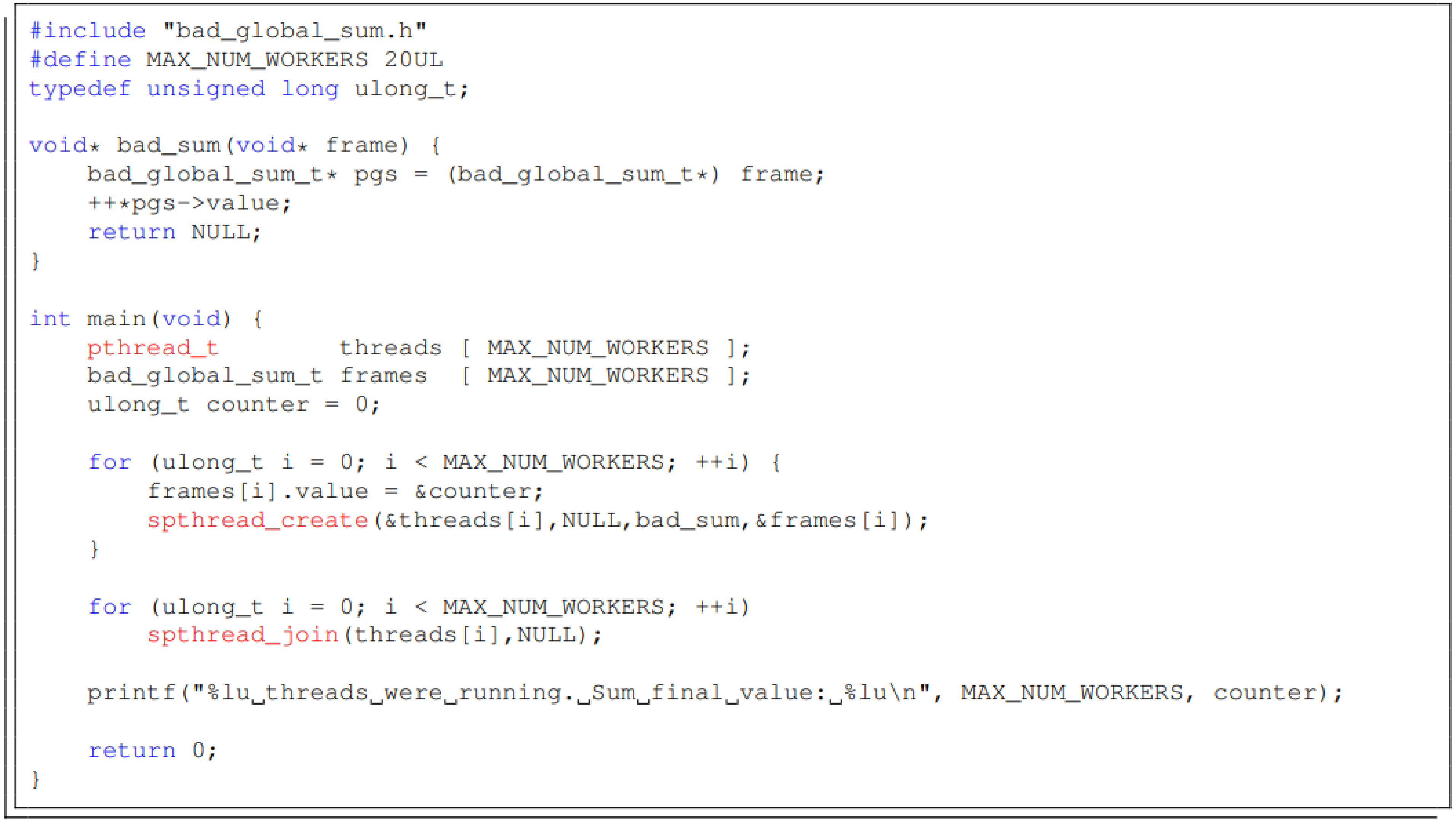
```
[main]  Creating workers...
[0]  Hello, World!
[main]  Joining the workers...
[2]  Hello, World!
[main]  Worker N.0 has returned!
[1]  Hello, World!
[3]  Hello, World!
[main]  Worker N.1 has returned!
[main]  Worker N.2 has returned!
[main]  Worker N.3 has returned!
```

Another Pthreads Example: parallel counter

```
#ifndef BAD_GLOBAL_SUM_H
#define BAD_GLOBAL_SUM_H

#include <stdio.h>
#include <stdlib.h>
#include "utils.h"

typedef struct bad_global_sum_s {
    unsigned long *value;
} bad_global_sum_t;

#endif // BAD_GLOBAL_SUM_H
```

Figure : bad_global_sum.h

Another Pthreads Example: parallel counter



```
#include "bad_global_sum.h"
#define MAX_NUM_WORKERS 20UL
typedef unsigned long ulong_t;

void* bad_sum(void* frame) {
    bad_global_sum_t* pgs = (bad_global_sum_t*) frame;
    ++*pgs->value;
    return NULL;
}

int main(void) {
    pthread_t        threads [ MAX_NUM_WORKERS ];
    bad_global_sum_t frames  [ MAX_NUM_WORKERS ];
    ulong_t counter = 0;

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i) {
        frames[i].value = &counter;
        spthread_create(&threads[i],NULL,bad_sum,&frames[i]);
    }

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)
        spthread_join(threads[i],NULL);

    printf("%lu␣threads␣were␣running.␣Sum␣final␣value:␣%lu\n", MAX_NUM_WORKERS, counter);

    return 0;
}
```

Figure : bad_sum_pthreads.c

Another Pthreads Example: parallel counter

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c bad_sum_pthreads.c
gcc -o badsum bad_sum_pthreads.o -lpthread
```

. . . Don't forget to link with the PTHREAD library!

Output of `./badsum`

```
szuckerm@evans201g:bad$ ./badsum
20 threads were running. Sum final value: 20
```

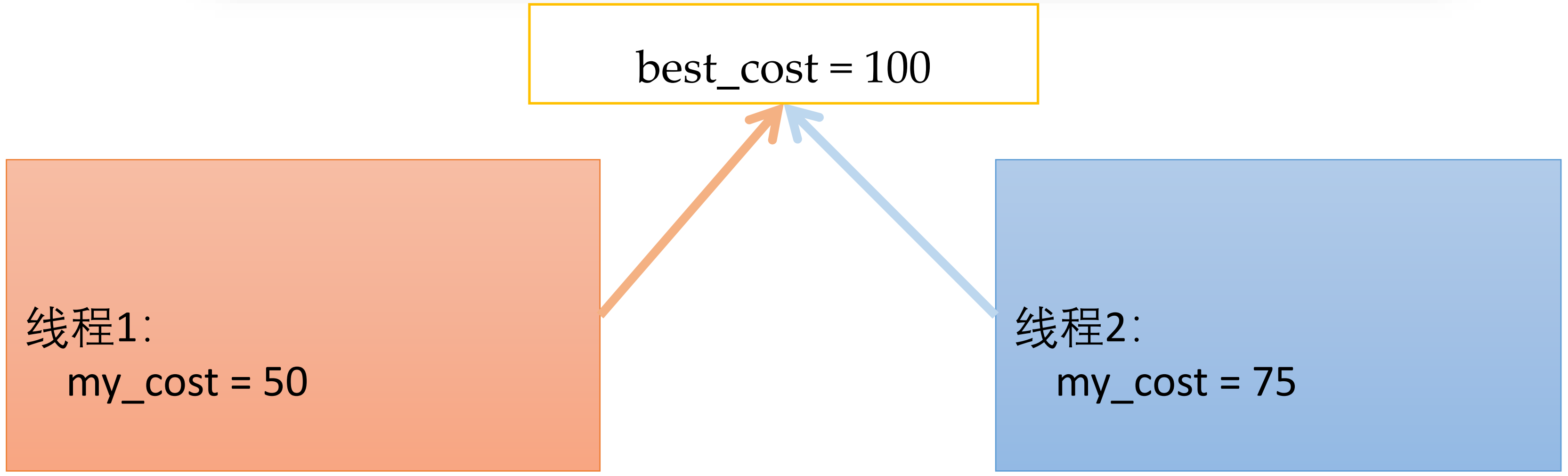
Hey, it's working!

Multiple executions of `./badsum`

```
szuckerm@evans201g:bad$ (for i in `seq 100`;do ./badsum ;done)|uniq
20 threads were running. Sum final value: 20
20 threads were running. Sum final value: 19
20 threads were running. Sum final value: 20
20 threads were running. Sum final value: 19
20 threads were running. Sum final value: 20
```

多线程程序设计—数据竞争

```
/* threads compete to update global variable best_cost */
if (my_cost < best_cost)
   best_cost = my_cost;
```



？ best_cost = 50 ? 75

临界区和锁

- 临界区
  - 同一时刻只有一个线程执行的代码段
  - 互斥锁实现临界区
- 互斥锁

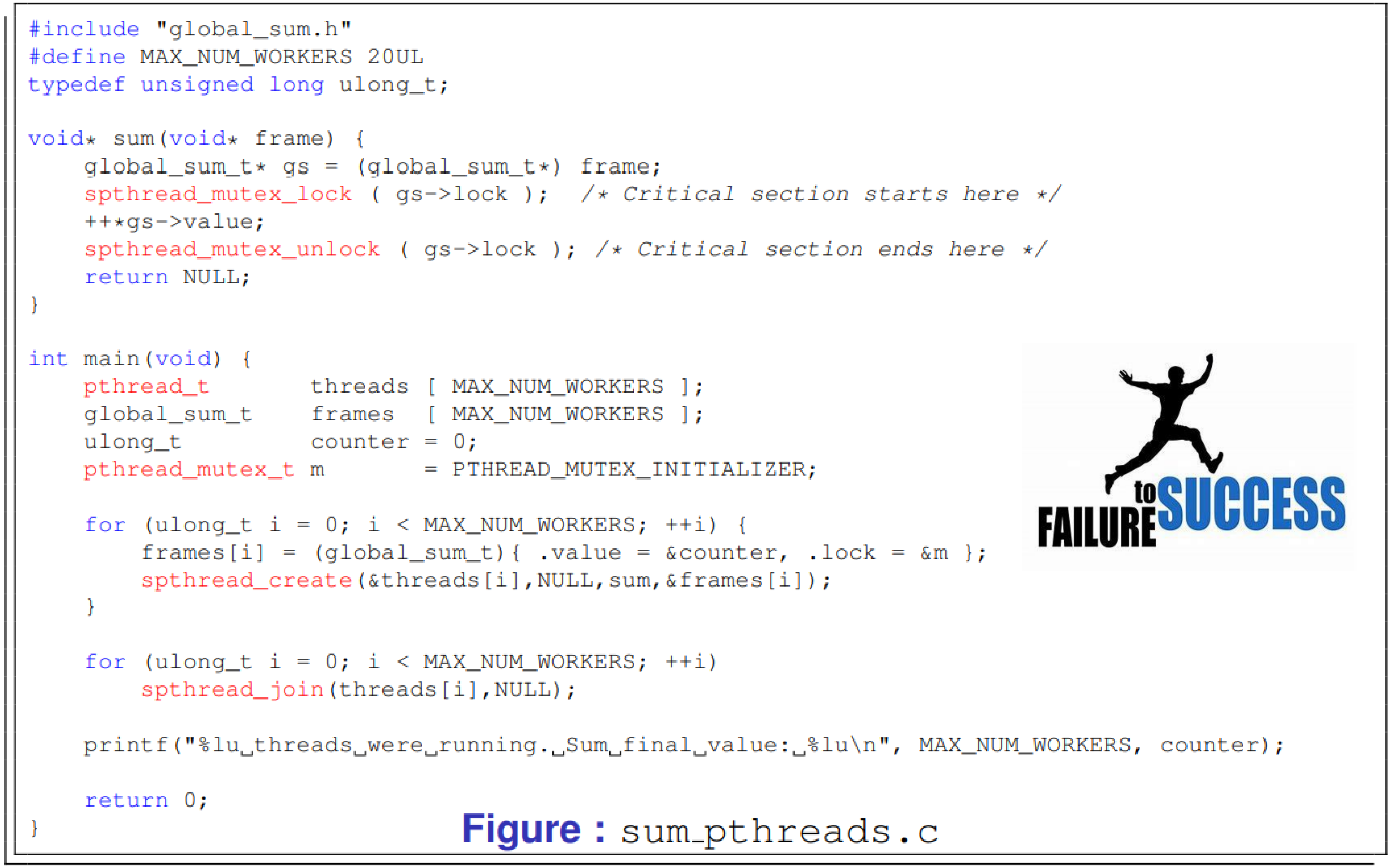
```
pthread_mutex_init (mutex_lock, attr)
pthread_mutex_lock (mutex_lock)
if (my_cost < best_cost)
   best_cost = my_cost
pthread_mutex_unlock (mutex_lock)
```

Another Pthreads Example: parallel counter (fixed)

```
#ifndef GLOBAL_SUM_H
#define GLOBAL_SUM_H

#include <stdio.h>
#include <stdlib.h>
#include "utils.h"

typedef struct global_sum_s {
    unsigned long   *value;
    pthread_mutex_t *lock;
} global_sum_t;

#endif // GLOBAL_SUM_H
```

Figure : global_sum.h

Another Pthreads Example: parallel counter (fixed)



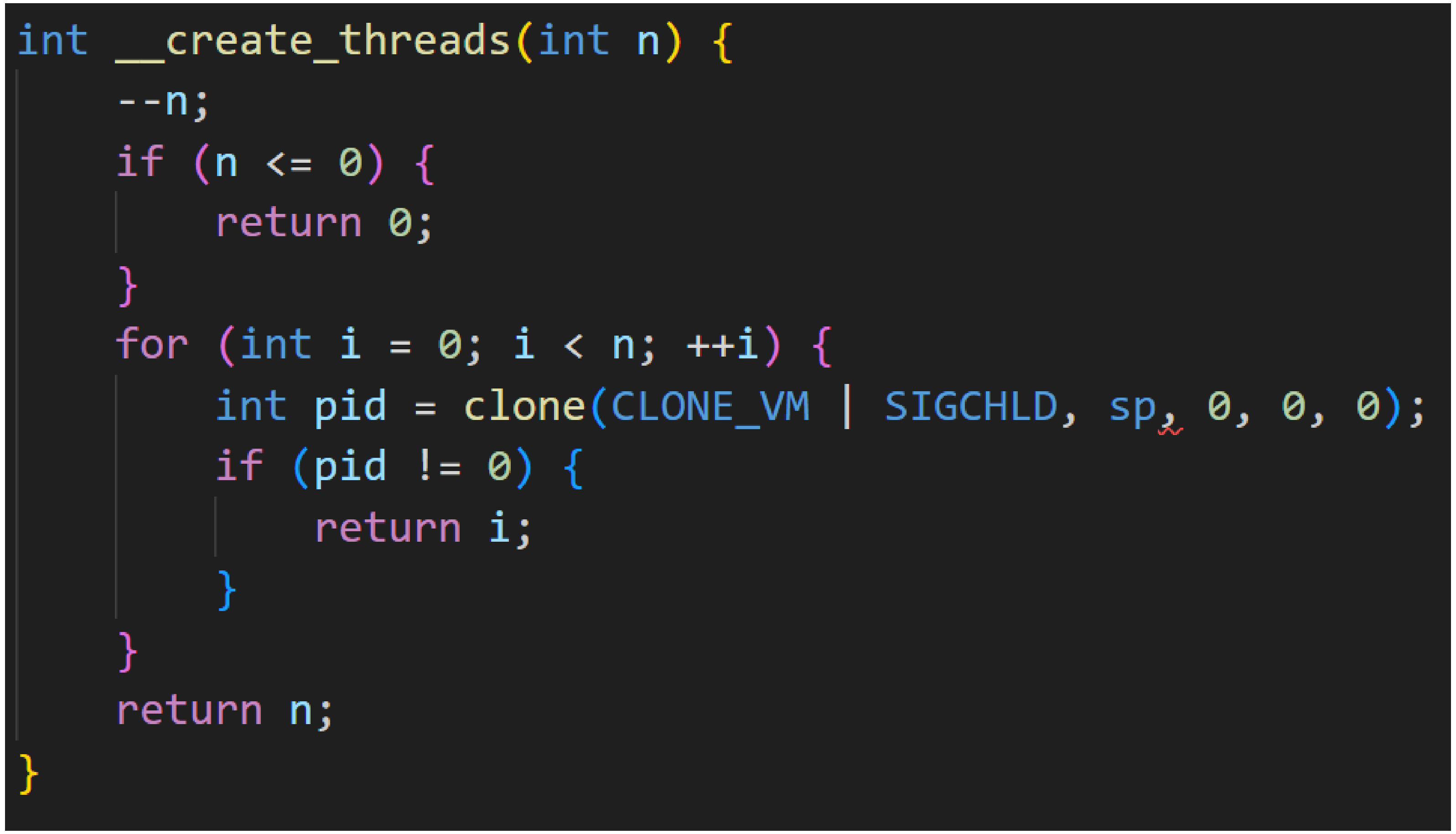
```c
#include "global_sum.h"
#define MAX_NUM_WORKERS 20UL
typedef unsigned long ulong_t;

void* sum(void* frame) {
    global_sum_t* gs = (global_sum_t*) frame;
    spthread_mutex_lock ( gs->lock );  /* Critical section starts here */
    ++*gs->value;
    spthread_mutex_unlock ( gs->lock ); /* Critical section ends here */
    return NULL;
}

int main(void) {
    pthread_t        threads [ MAX_NUM_WORKERS ];
    global_sum_t     frames  [ MAX_NUM_WORKERS ];
    ulong_t          counter = 0;
    pthread_mutex_t  m       = PTHREAD_MUTEX_INITIALIZER;

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i) {
        frames[i] = (global_sum_t){ .value = &counter, .lock = &m };
        spthread_create(&threads[i],NULL,sum,&frames[i]);
    }

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)
        spthread_join(threads[i],NULL);

    printf("%lu␣threads␣were␣running.␣Sum␣final␣value:␣%lu\n", MAX_NUM_WORKERS, counter);

    return 0;
}
```

Figure : sum_pthreads.c

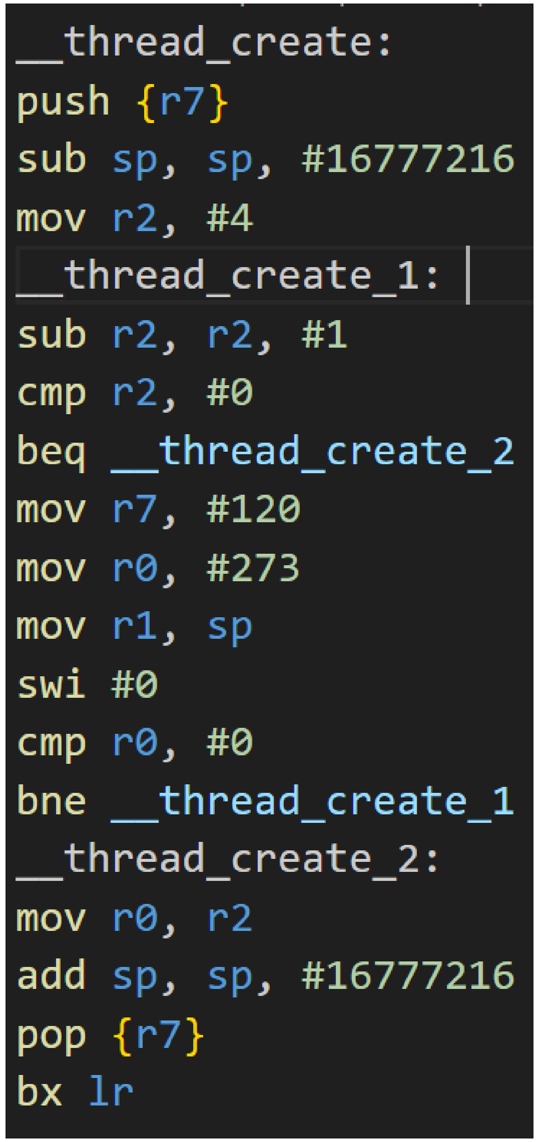
__thread_create函数

- 使用clone创建轻量级的线程

```
int clone(int (*fn)(void *fnarg), void *child_stack, int flags, void *arg);
```



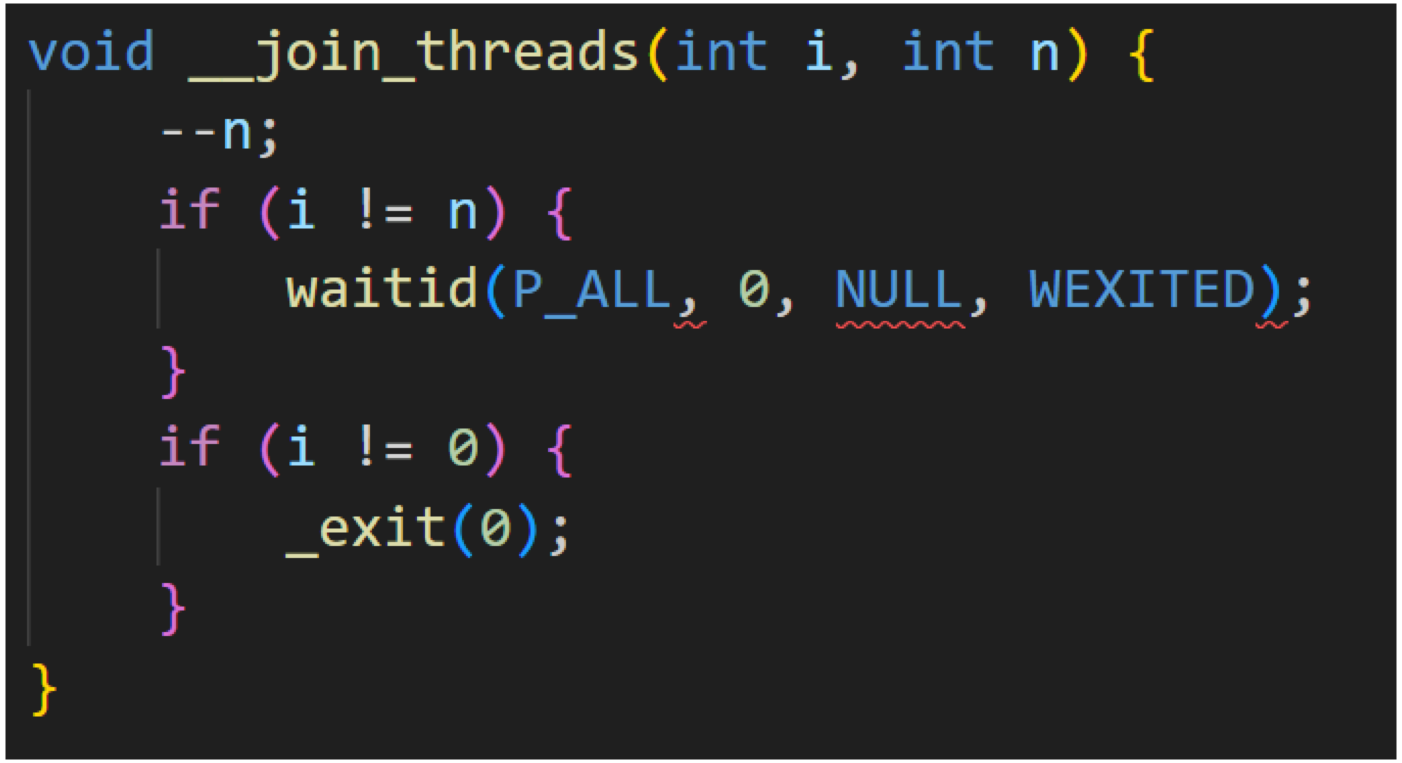
```
int __create_threads(int n) {
    --n;
    if (n <= 0) {
        return 0;
    }
    for (int i = 0; i < n; ++i) {
        int pid = clone(CLONE_VM | SIGCHLD, sp, 0, 0, 0);
        if (pid != 0) {
            return i;
        }
    }
    return n;
}
```



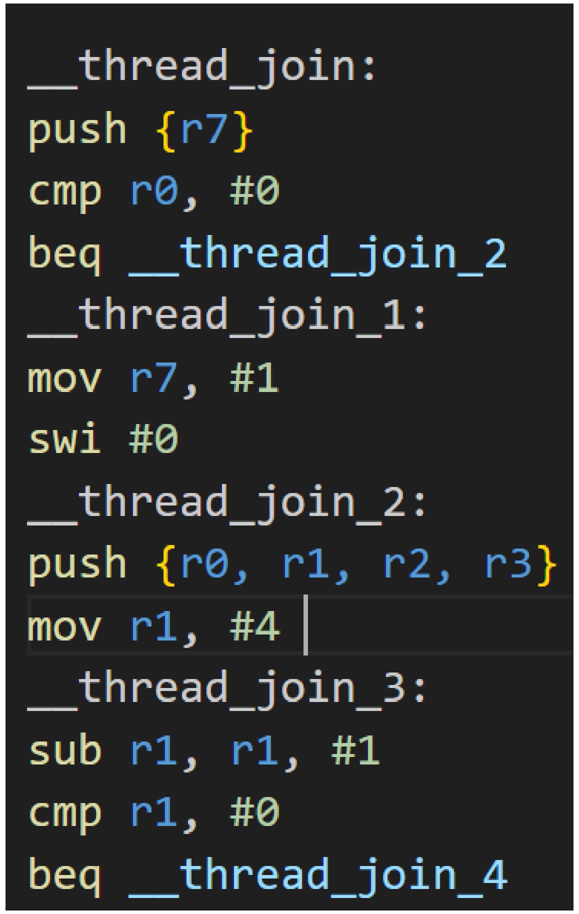
```
__thread_create:
push {r7}
sub sp, sp, #16777216
mov r2, #4
__thread_create_1:
sub r2, r2, #1
cmp r2, #0
beq __thread_create_2
mov r7, #120
mov r0, #273
mov r1, sp
swi #0
cmp r0, #0
bne __thread_create_1
__thread_create_2:
mov r0, r2
add sp, sp, #16777216
pop {r7}
bx lr
```

__thread_join函数

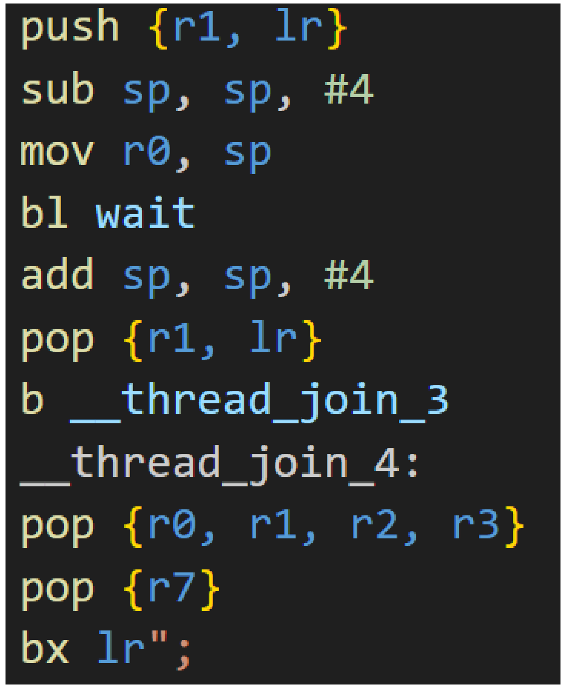
- 使用waitid等待进程改变状态
- 使用_exit终止进程



```c
void __join_threads(int i, int n) {
    --n;
    if (i != n) {
        waitid(P_ALL, 0, NULL, WEXITED);
    }
    if (i != 0) {
        _exit(0);
    }
}
```



```
__thread_join:
push {r7}
cmp r0, #0
beq __thread_join_2
__thread_join_1:
mov r7, #1
swi #0
__thread_join_2:
push {r0, r1, r2, r3}
mov r1, #4
__thread_join_3:
sub r1, r1, #1
cmp r1, #0
beq __thread_join_4
```



```
push {r1, lr}
sub sp, sp, #4
mov r0, sp
bl wait
add sp, sp, #4
pop {r1, lr}
b __thread_join_3
__thread_join_4:
pop {r0, r1, r2, r3}
pop {r7}
bx lr";
```

Loop Parallel

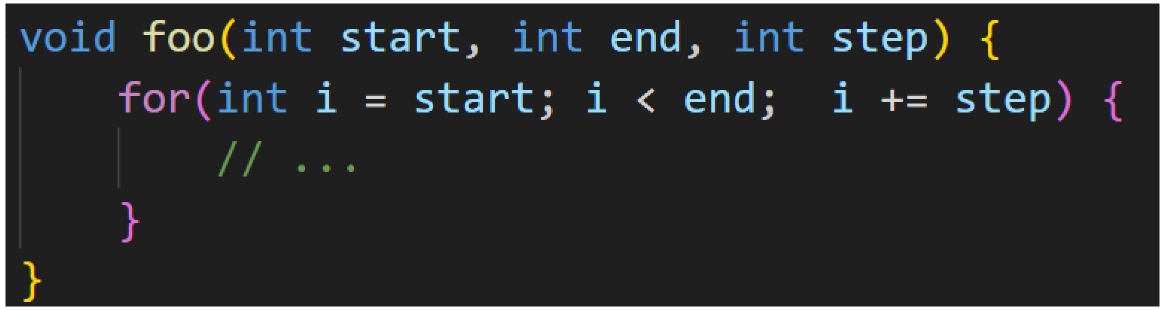
当前循环并行化判断条件

- As for the Loop, it must be the simple for-loop, the MAXN Count of the Loop is invarilant.
- Not in the recursive function.
- Do not has any non-prue function call.
- need to be loopInvarilant
- In the Loop, there is no alias.
- For enery store, there must be a single gep.
- There is no gep for the scalar GV.
- ***For the Red condition, these condition can be weaker, or as long as there is no loop-carry dependency where the dependency distance is less than the loop limit, which need Dependency Analysis, you must build the dependency tree!***

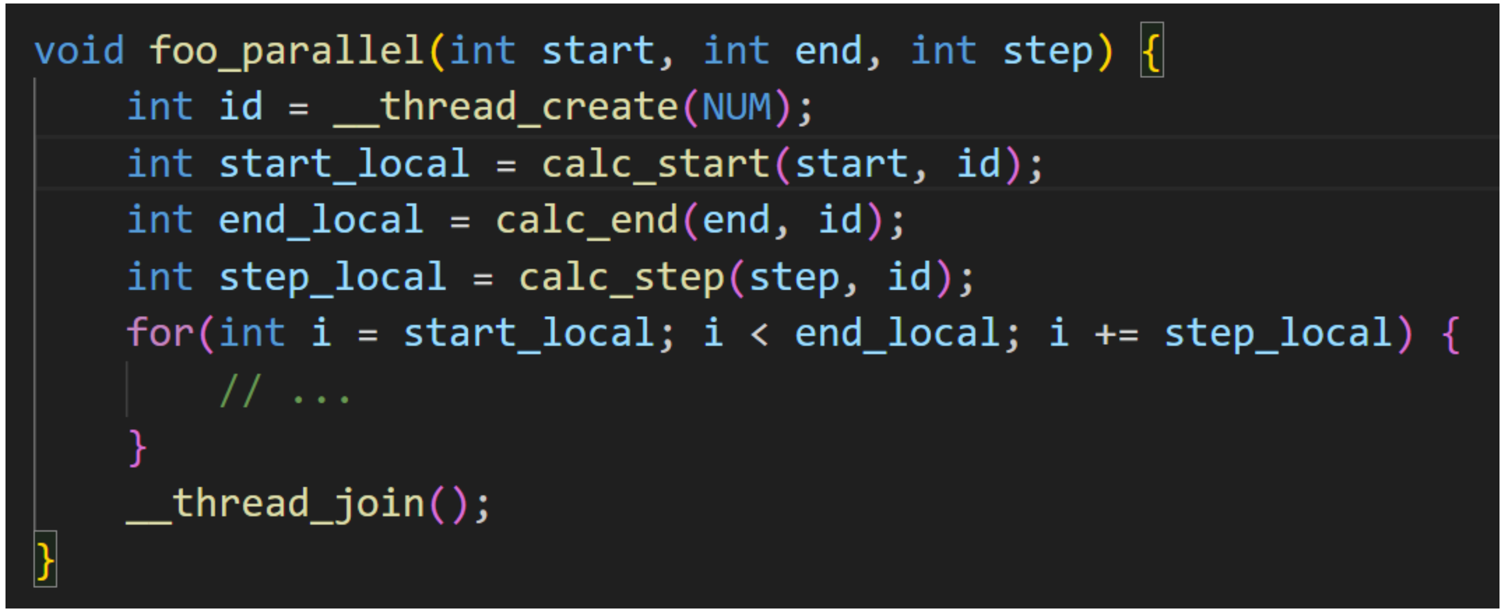
https://github.com/RaVincentHuang/Diana/blob/master/lib/Transform/Loop/LoopParallel.cpp

并行化具体实现

- **就地修改原循环，插入建立新线程的函数**



```
void foo(int start, int end, int step) {
    for(int i = start; i < end;  i += step) {
        // ...
    }
}
```



```
void foo_parallel(int start, int end, int step) {
    int id = __thread_create(NUM);
    int start_local = calc_start(start, id);
    int end_local = calc_end(end, id);
    int step_local = calc_step(step, id);
    for(int i = start_local; i < end_local; i += step_local) {
        // ...
    }
    __thread_join();
}
```

并行化具体实现

- 假设某个循环满足并行化的要求
- 首先需要计算新的循环的结构 *for i in range(start, end, step)*
  - 假设并行成N个线程, 原始循环为(start, end, step)
  - start_local = start + id * (end -start) / N
  - end_local = start_local + (end -start) / N
  - step_local = step
- 修改循环，包括当前的循环限、与循环相关的phi函数
- 在循环前后插入__thread_create函数和__thread_join函数

课堂总结

- 1. 多进程
  - 1.1 进程概念
  - 1.2 进程操作
  - 1.3 进程状态
  - 1.4 进程调度
  - 1.5 进程间通信
- 2. 多线程
  - 2.1 线程概念
  - 2.2 多核与多线程
  - 2.3 Pthread编程
  - 2.4 自己动手实现线程