SYNC Basic

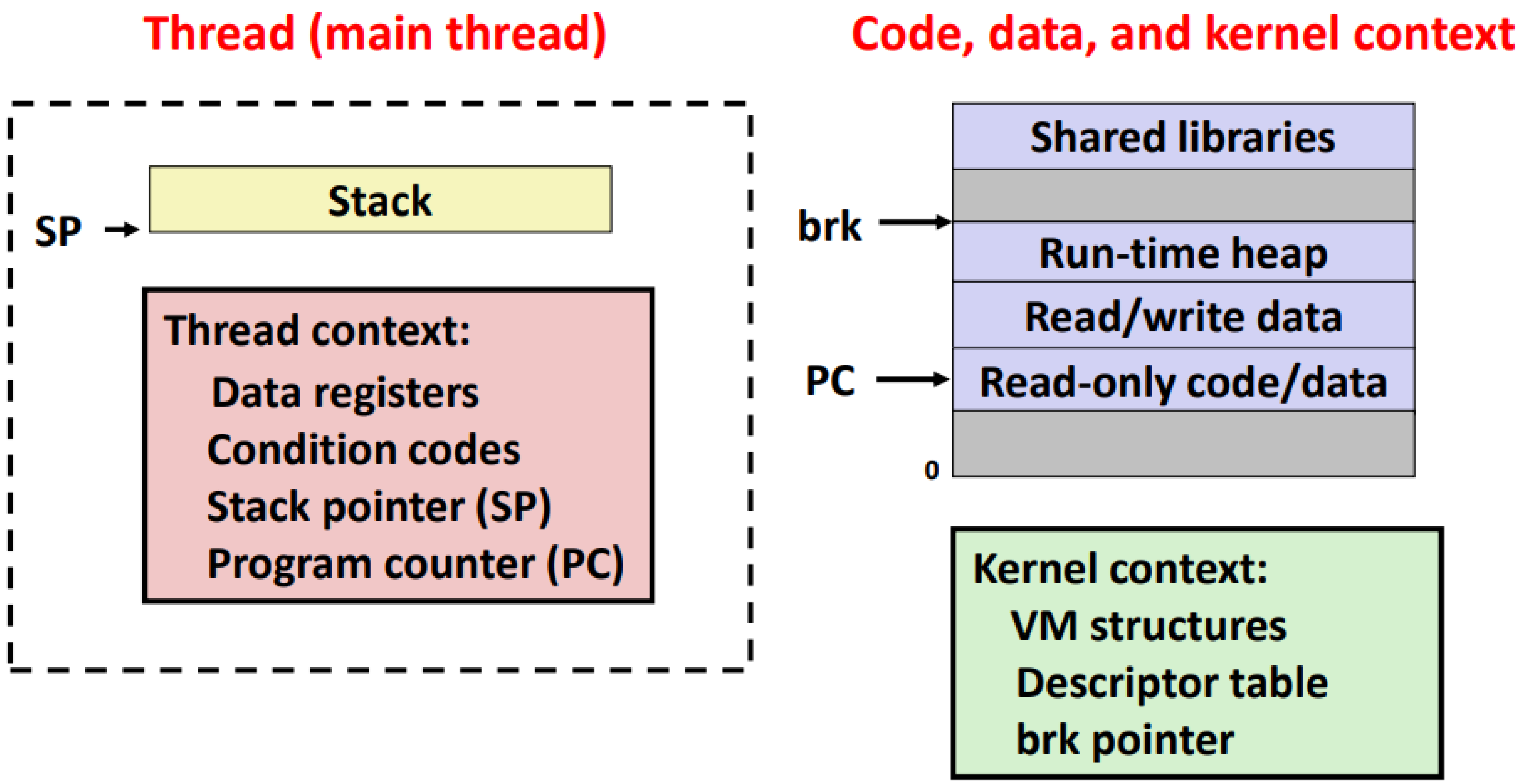
徐品原

Threads vs. Processes

- **Threads and processes: similarities**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is scheduled and context switched by the kernel

- **Threads and processes: differences**
  - Threads share code and data, processes (typically) do not
  - Threads are less expensive than processes
    - Process control (creating and reaping) is more expensive than thread control
    - Context switches for processes more expensive than for threads

Process = thread + code, data, and kernel context



- pthread_create()与fork()
- pthread头文件： pthread.h
- int pthread_create(pthread_t* restrict tidp,const pthread_attr_t* restrict_attr,void* (*start_rtn)(void*),void *restrict arg);
- tidp：事先创建好的pthread_t类型的参数。成功时tidp指向的内存单元被设置为新创建线程的线程ID。

attr：用于定制各种不同的线程属性。APUE的12.3节讨论了线程属性。通常直接设为NULL。

start_rtn：新创建线程从此函数开始运行。无参数是arg设为NULL即可。

arg：start_rtn函数的参数。无参数时设为NULL即可。有参数时输入参数的地址。当多于一个参数时应当使用结构体传入。

返回值：成功返回0，否则返回错误码

Pthread_join: int pthread_join(pthread_t thread, void **retval);相当于wait

线程竞争

下面的程序会引发竞争。一个可能的输出结果为2 1 2 2。解释输出这一结果的原因。

```
long foo = 0, bar = 0;

void *thread(void *vargp) {
    foo++;
    bar++;
    printf("%ld %ld ", foo, bar);
    fflush(stdout);
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
              *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

`cnt` should equal 20,000.

What went wrong?

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq  (%rdi), %rcx
    testq %rcx,%rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

H_i : Head

L_i : Load cnt

U_i : Update cnt

S_i : Store cnt

T_i : Tail

P(s)

- If *s* is nonzero, then decrement *s* by 1 and return immediately.
  - Test and decrement operations occur atomically (indivisibly)
- If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
- After restarting, the P operation decrements *s* and returns control to the caller.

V(s):

- Increment *s* by 1.
  - Increment operation occurs atomically
- If there are any threads blocked in a P operation waiting for *s* to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing *s*.

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

CS:APP wrapper functions:

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

信号量提供了一种很方便的方法来确保对共享变量的互斥访问。

基本的思想是

将每个共享变量（或一组相关的共享变量）与一个信号量s（初始为1）联系起来。
然后用P(s)和V(s)操作相应的临界区包围起来。
以这种方式保护共享变量的信号量叫做**二元信号量(binary semaphore)**，因为它的值总是0或者1。

以提供互斥为目的的二元信号量常常也称为**互斥锁(mutex)**。

在一个互斥锁上执行P操作叫做互斥锁加锁。
在一个互斥锁上执行V操作叫做互斥锁解锁。
对一个互斥锁加了锁还没有解锁的线程称为占用这个互斥锁。

`goodcnt.c`: Proper Synchronization

- Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;            /* Semaphore that protects cnt */

sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- Surround critical section with *P* and *V*:

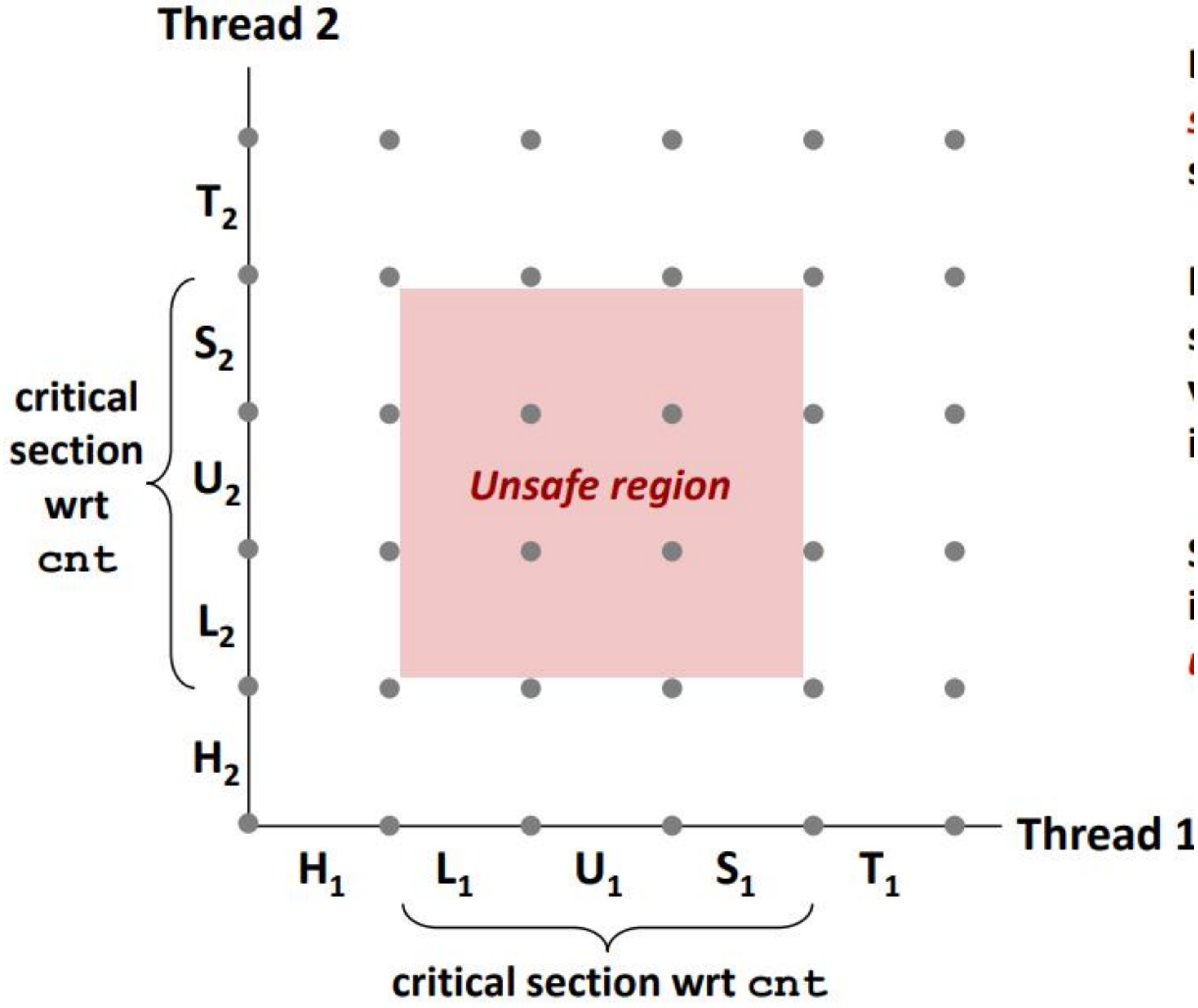
```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Function	badcnt	goodcnt
Time (ms) niters = 10^6	12	450
Slowdown	1.0	37.5

Warning: It's orders of magnitude slower than `badcnt.c`.



Thread 2
T_2
S_2
U_2
L_2
H_2
critical section wrt cnt
Unsafe region
Thread 1
H_1
L_1
U_1
S_1
T_1
critical section wrt cnt

信号量引入了一种潜在的令人厌恶的运行时错误，叫做**死锁 (deadlock)**。

指的是一组线程被阻塞，等待一个永远不为真的条件。
死锁的区域d是一个只能进，不能出的区域。

位置是合法的，并不是禁止区。
但是会发现无论向上，还是右，都只剩下禁止区了。
如果禁止区不重叠，一定不会发生死锁。

否则，可能发生死锁。
死锁是一个相当困难的问题，因为它不总是可预测的。
错误还不会重复，轨迹不同。

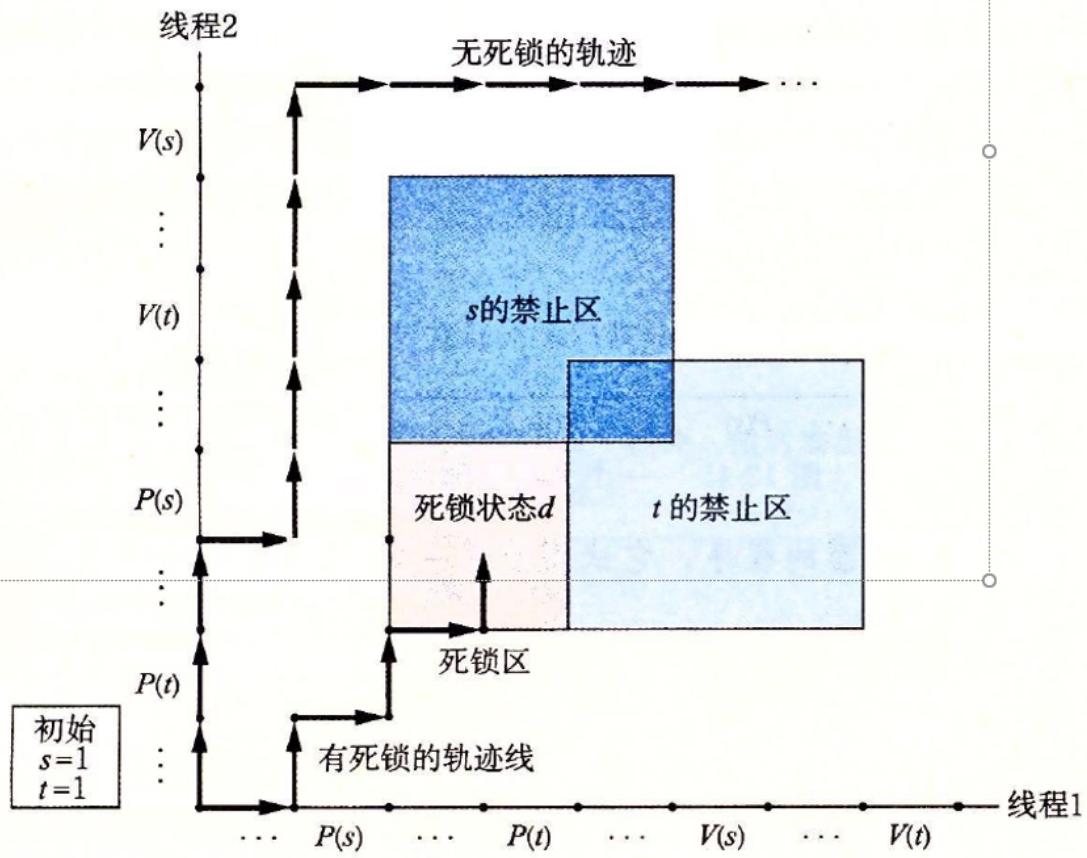


图 12-44 一个会死锁的程序的进度图
net/winter_wu_1998