Synchronization: Basics

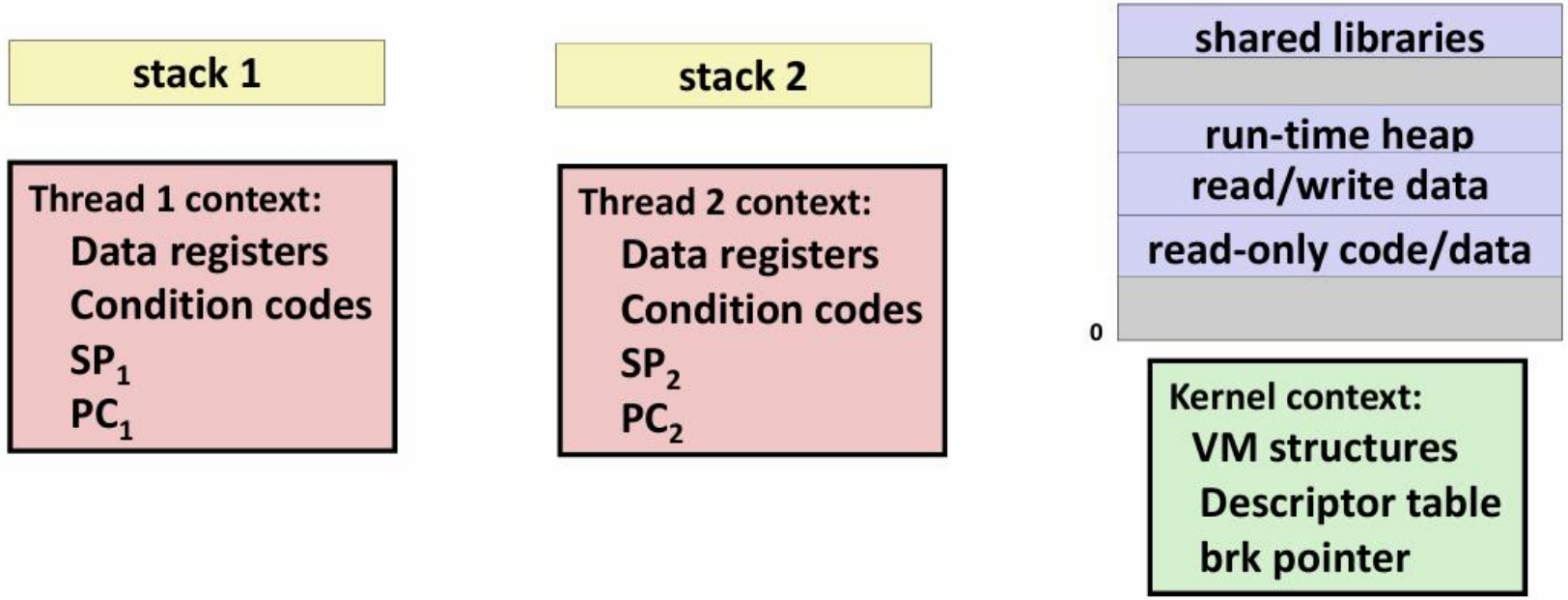
周百川 2024-12-25

- 线程间变量共享
- 线程同步与信号量

线程内存模型

进程：独立的进程上下文+独立的虚拟内存空间

线程：独立的线程上下文+共享的虚拟内存空间



线程1私有　　线程2私有　　所有线程共享

线程间共享变量

全局变量：虚拟内存中只有一个实例，所有线程共享 (ptr)

本地自动变量：在对应线程的栈中，线程间不共享 (myid)

本地静态变量：虚拟内存中只有一个实例，所有线程共享 (cnt)

```
#include "csapp.h"
#define N 2
void *thread(void *vargp);

char **ptr;  /* Global variable */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

线程间共享变量

1.Malloc and Free

```
int main(int argc, char *argv[]) {
    long i;
    pthread_t tids[N];
    for (i = 0; i < N; i++) {
        long* p= Malloc(sizeof(long));
        *p = i;
        Pthread_create(&tids[i], NULL, thread, (void *)p);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tids[i], NULL);
    check();
}

void *thread(void *vargp) {
    hist[*(long *)vargp] += 1;
    Free(vargp);
    return NULL;
}
```

OK

线程间共享变量

2.Cast of int

```
int main(int argc, char *argv[]) {
    long i;
    pthread_t tids[N];
    for (i = 0; i < N; i++) {
        Pthread_create(&tids[i], NULL, thread, (void *)i);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tids[i], NULL);
    check();
}

void *thread(void *vargp) {
    hist[(long)vargp] += 1;
    return NULL;
}
```

OK

线程间共享变量

3.Ptr to stack slot

WRONG!

Data race created

```
int main(int argc, char *argv[]) {
    long i;
    long *p;
    pthread_t tids[N];
    for (i = 0; i < N; i++) {
        p = &i;
        pthread_create(&tids[i], NULL, thread, (void *)p);
    }
    for (i = 0; i < N; i++)
        pthread_join(tids[i], NULL);
    check();
}

void *thread(void *vargp) {
    long *param = (long*)vargp;
    hist[*param] += 1;
    return NULL;
}
```

```
Failed at 1
```

```
Failed at 0
```

```
Failed at 3
```

- 共享(Shared)：一个变量是共享的，当且仅当多个线程引用这个变量的某个实例

Variable instance	Referenced by		
	main thread?	peer thread 0?	peer thread 1?
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

ptr, cnt , msgs are **shared**

```c
#include "csapp.h"
#define N 2
void *thread(void *vargp);

char **ptr;  /* Global variable */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

线程间的同步错误

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

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

    return NULL;
}
```

```
niters = 100000;
```

```
OK cnt=200000
```

```
niters = 1000000;
```

```
BOOM! cnt=1223033
```

```c
/* WARNING: This code is buggy! */
#include "csapp.h"

void *thread(void *vargp);  /* Thread routine prototype */

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

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

    /* Check input argument */
    if (argc != 2) {
        printf("usage: %s <niters>\n", argv[0]);
        exit(0);
    }
    niters = atoi(argv[1]);

    /* Create threads and wait for them to finish */
    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);
}
```

线程间的同步错误

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

→

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

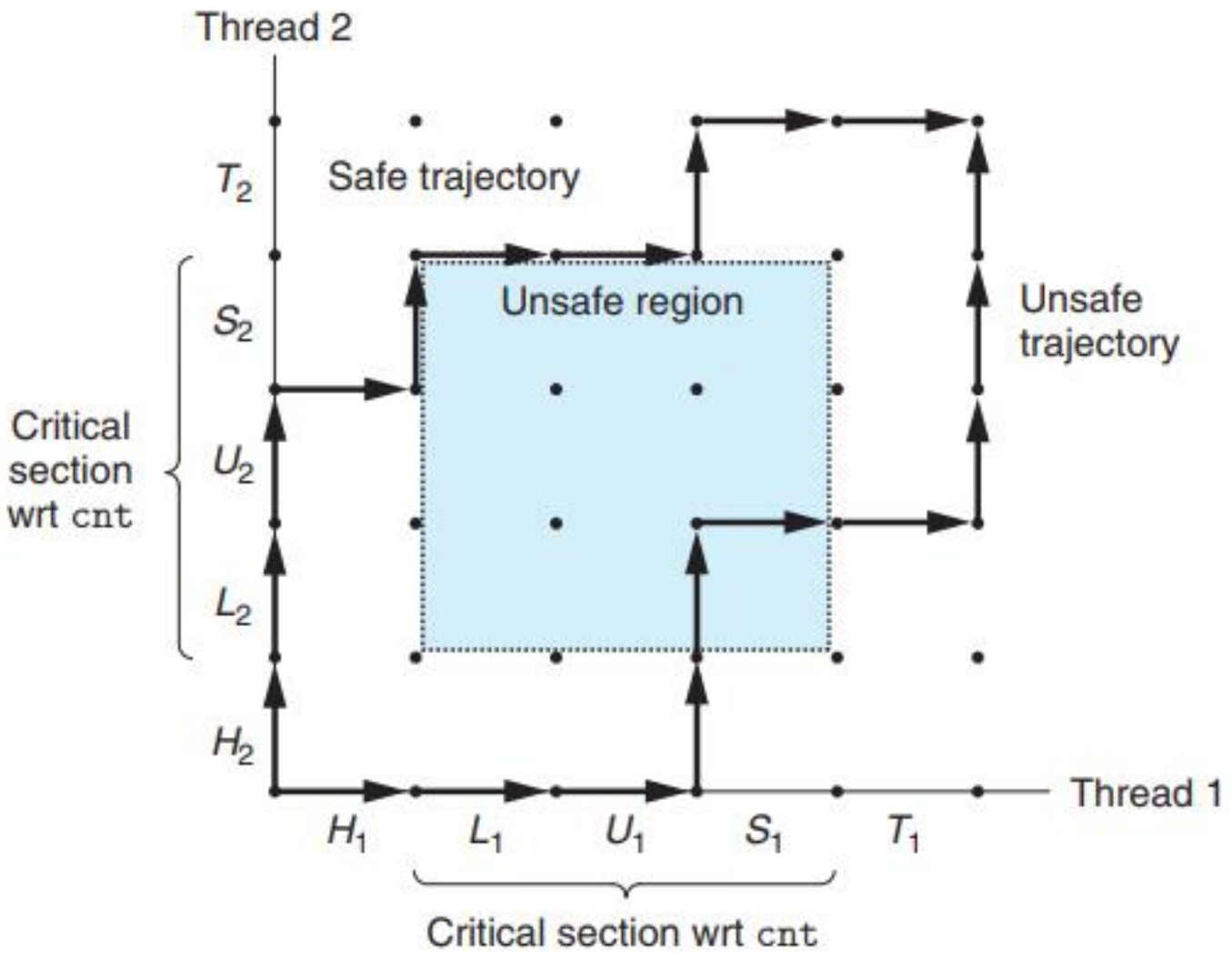
(a) Correct ordering

Step	Thread	Instr.	$\%rdx_1$	$\%rdx_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

(b) Incorrect ordering

Step	Thread	Instr.	$\%rdx_1$	$\%rdx_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

进度图与临界区


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

信号量(semaphore)：全局变量，非负整数

P(s): [**while (s == 0) wait(); s--;**]

V(s): [**s++;**]

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s);   /* P(s) */
int sem_post(sem_t *s);   /* V(s) */
```

Returns: 0 if OK, −1 on error

```
#include "csapp.h"

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

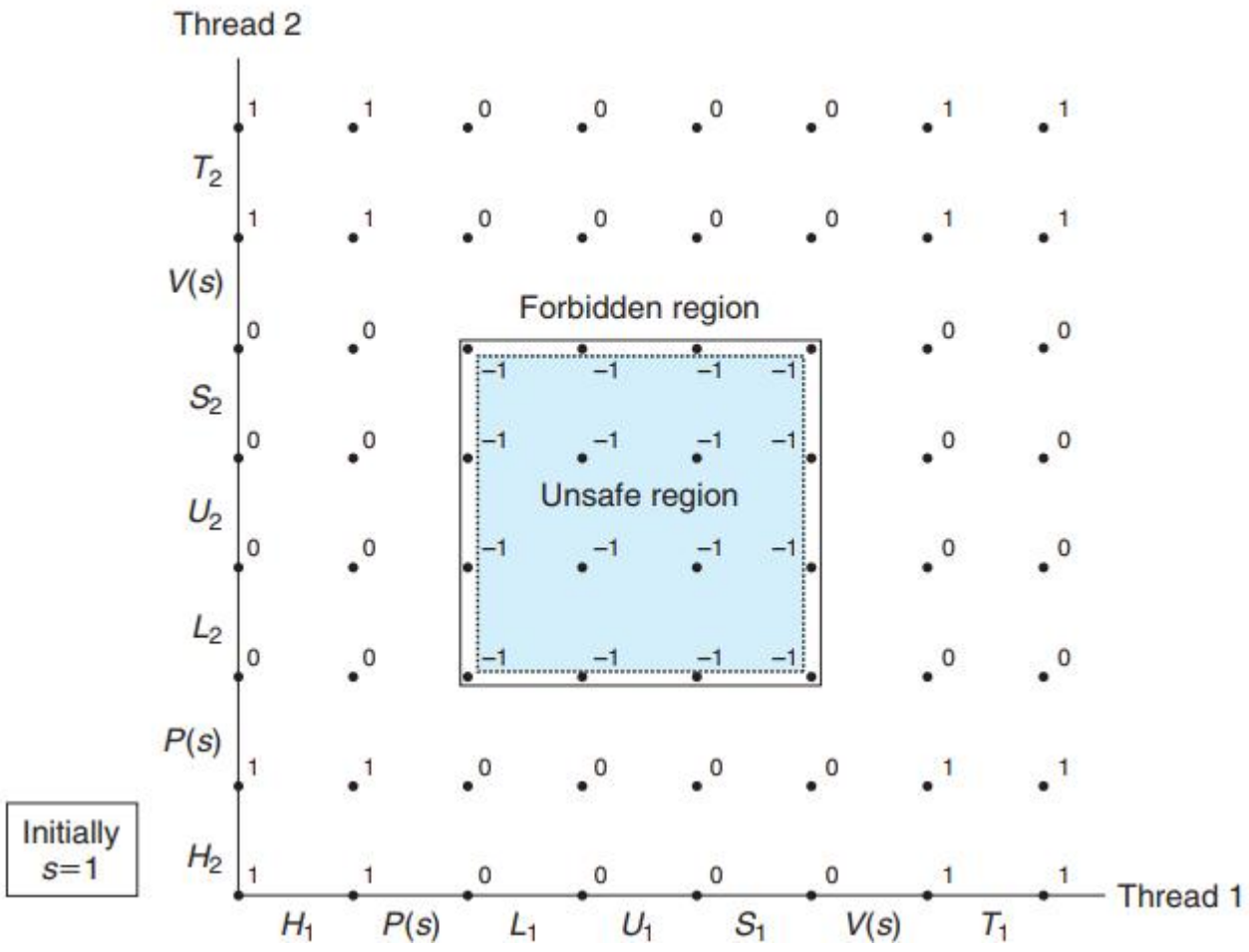
Returns: nothing

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

    for (i = 0; i < niters; i++)
        sem_wait(&mutex);
        cnt++;
        sem_post(&mutex);

    return NULL;
}
```

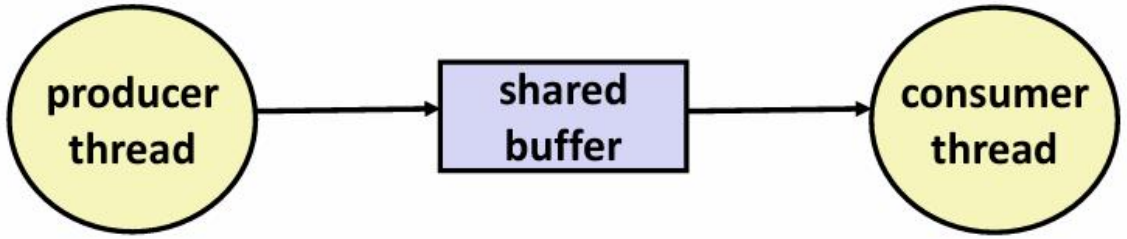
```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *((long *)vargp);

    for (i = 0; i < niters; i++)
        pthread_mutex_lock(&mutex);
        cnt++;
        pthread_mutex_unlock(&mutex);

    return NULL;
}
```

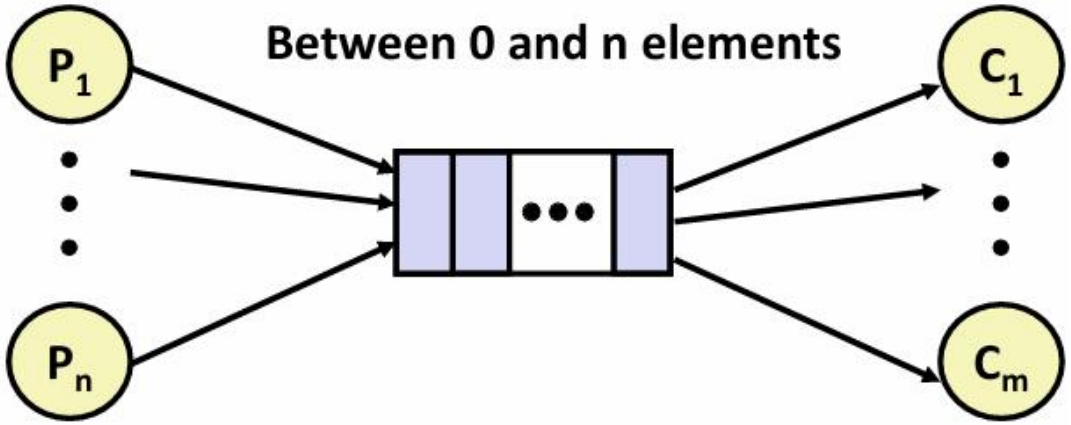
```
linux>  ./goodcnt 1000000
OK cnt=2000000

linux>  ./goodcnt 1000000
OK cnt=2000000
```

信号量与进度图


Thread 2
T_2
$V(s)$
S_2
U_2
L_2
$P(s)$
H_2
Forbidden region
Unsafe region
Initially
$s=1$
Thread 1
H_1
$P(s)$
L_1
U_1
S_1
$V(s)$
T_1

生产者-消费者问题





mutex: 互斥锁

slots:空缓冲区数量

items:可读项目数量

```
typedef struct {
    int *buf;          /* Buffer array */
    int n;             /* Maximum number of slots */
    int front;         /* buf[(front+1)%n] is first item */
    int rear;          /* buf[rear%n] is last item */
    sem_t mutex;       /* Protects accesses to buf */
    sem_t slots;       /* Counts available slots */
    sem_t items;       /* Counts available items */
} sbuf_t;
```

生产者-消费者问题

```
/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                        /* Buffer holds max of n items */
    sp->front = sp->rear = 0;         /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1);       /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n);       /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0);       /* Initially, buf has zero data items */
}

/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;   /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->items);                          /* Announce available item */
}
```

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                          /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)];  /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->slots);                          /* Announce available slot */
    return item;
}
```