15-213

"The course that gives CMU its Zip!"

Programming with Threads
Dec 5, 2002

Topics

- Shared variables
- The need for synchronization
- Synchronizing with semaphores
- Thread safety and reentrancy
- Races and deadlocks

class29.ppt

Shared Variables in Threaded C Programs

Question: Which variables in a threaded C program are shared variables?

- The answer is not as simple as "global variables are shared" and "stack variables are private".

Requires answers to the following questions:

- What is the memory model for threads?
- How are variables mapped to memory instances?
- How many threads reference each of these instances?

Threads Memory Model

Conceptual model:

- Each thread runs in the context of a process.
- Each thread has its own separate thread context.
  - Thread ID, stack, stack pointer, program counter, condition codes, and general purpose registers.
- All threads share the remaining process context.
  - Code, data, heap, and shared library segments of the process virtual address space.
  - Open files and installed handlers

Operationally, this model is not strictly enforced:

- While register values are truly separate and protected....
- Any thread can read and write the stack of any other thread.

Mismatch between the conceptual and operation model is a source of confusion and errors.

Example of Threads Accessing Another Thread's Stack

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int svar = 0;

    printf("[%d]: %s (svar=%d)\n",
         myid, ptr[myid], ++svar);
}
```

Peer threads access main thread's stack indirectly through global ptr variable

Mapping Variables to Mem. Instances

Global var: 1 instance (`ptr` [data])

Local automatic vars: 1 instance (`i.m`, `msgs.m`)

Local automatic var: 2 instances (
`myid.p0`[peer thread 0's stack],
`myid.p1`[peer thread 1's stack]
)

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int svar = 0;

    printf("[%d]: %s (svar=%d)\n",
         myid, ptr[myid], ++svar);
}
```

Local static var: 1 instance (`svar` [data])

Shared Variable Analysis

Which variables are shared?

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

Answer: A variable x is shared iff multiple threads reference at least one instance of x. Thus:

- `ptr`, `svar`, and `msgs` are shared.
- `i` and `myid` are NOT shared.

badcnt.c: An Improperly Synchronized Threaded Program

```
unsigned int cnt = 0; /* shared */

int main() {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
                   count, NULL);
    Pthread_create(&tid2, NULL,
                   count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n",
               cnt);
    else
        printf("OK cnt=%d\n",
               cnt);
}
```

```
/* thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

```
linux> ./badcnt
BOOM! cnt=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672
```

cnt should be equal to 200,000,000. What went wrong?!

Assembly Code for Counter Loop

C code for counter loop

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

Corresponding asm code
(gcc -O0 -fforce-mem)

Head (H_i)

```
.L9:
        movl -4(%ebp),%eax
        cmpl $99999999,%eax
        jle .L12
        jmp .L10
```

Load cnt (L_i)
Update cnt (U_i)
Store cnt (S_i)

```
.L12:
        movl cnt,%eax        # Load
        leal 1(%eax),%edx    # Update
        movl %edx,cnt        # Store
```

Tail (T_i)

```
.L11:
        movl -4(%ebp),%eax
        leal 1(%eax),%edx
        movl %edx,-4(%ebp)
        jmp .L9
.L10:
```

Concurrent Execution

Key idea: In general, any sequentially consistent interleaving is possible, but some are incorrect!

- I_i denotes that thread i executes instruction I
- $\%eax_i$ is the contents of %eax in thread i's context

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

Concurrent Execution (cont)

Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2.

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

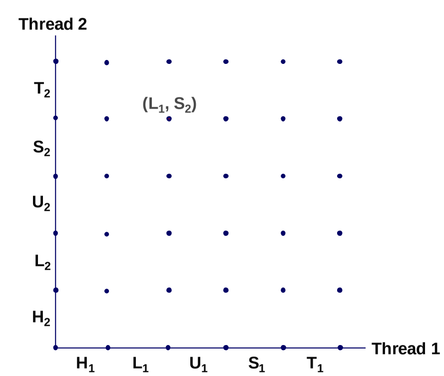
Concurrent Execution (cont)

How about this ordering?

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1			
1	L_1			
2	H_2			
2	L_2			
2	U_2			
2	S_2			
1	U_1			
1	S_1			
1	T_1			
2	T_2			

We can clarify our understanding of concurrent execution with the help of the *progress graph*

Progress Graphs



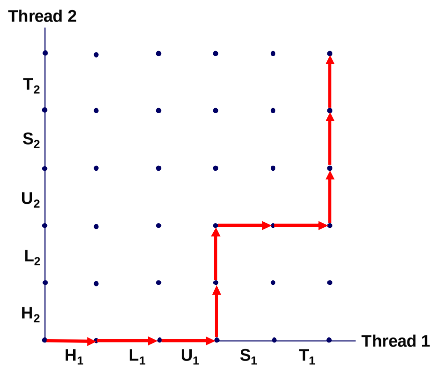
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible *execution state* $(Inst_1, Inst_2)$.

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Trajectories in Progress Graphs

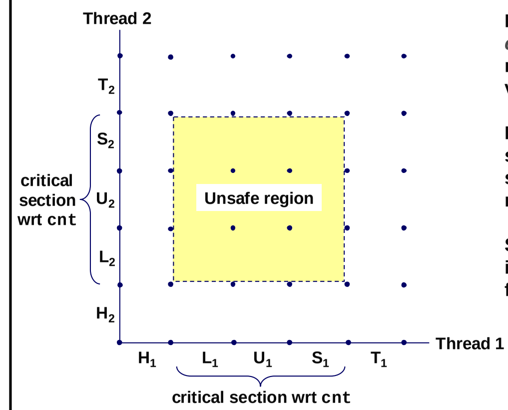


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

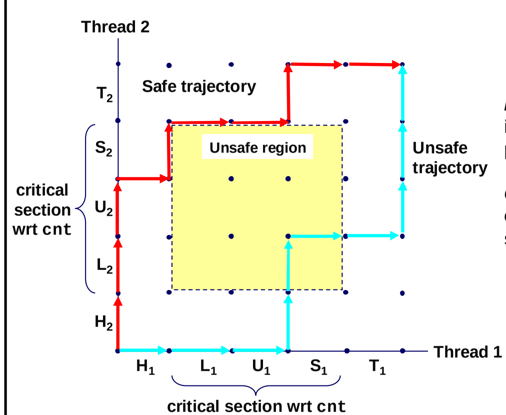


L, U, and S form a *critical section* with respect to the shared variable `cnt` .

Instructions in critical sections (wrt to some shared variable) should not be interleaved.

Sets of states where such interleaving occurs form *unsafe regions*.

Safe and Unsafe Trajectories



Def: A trajectory is *safe* iff it doesn't touch any part of an unsafe region.

Claim: A trajectory is correct (wrt `cnt`) iff it is safe.

Semaphores

Question: **How can we guarantee a safe trajectory?**

- We must *synchronize* the threads so that they never enter an unsafe state.

Classic solution**: Dijkstra's P and V operations on *semaphores.***

- *semaphore:* non-negative integer synchronization variable.
  - P(s): [`while (s == 0) wait(); s--;`]
    - » Dutch for "Proberen" (test)
  - V(s): [`s++;`]
    - » Dutch for "Verhogen" (increment)
- OS guarantees that operations between brackets [ ] are executed indivisibly.
  - Only one P or V operation at a time can modify s.
  - When `while` loop in P terminates, only that P can decrement s.

Semaphore invariant: *(s >= 0)*

Safe Sharing with Semaphores

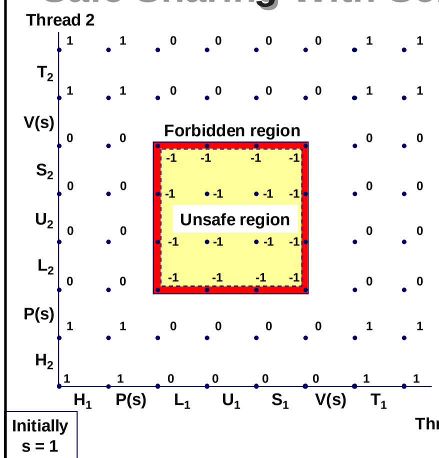
Here is how we would use P and V operations to synchronize the threads that update `cnt`.

```
/* Semaphore s is initially 1 */

/* Thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(s);
        cnt++;
        V(s);
    }
    return NULL;
}
```


Safe Sharing With Semaphores



Provide mutually exclusive access to shared variable by surrounding critical section with P and V operations on semaphore s (initially set to 1).

Semaphore invariant creates a *forbidden region* that encloses unsafe region and is never touched by any trajectory.

POSIX Semaphores

```
/* Initialize semaphore sem to value */
/* pshared=0 if thread, pshared=1 if process */
void Sem_init(sem_t *sem, int pshared, unsigned int value) {
  if (sem_init(sem, pshared, value) < 0)
    unix_error("Sem_init");
}

/* P operation on semaphore sem */
void P(sem_t *sem) {
  if (sem_wait(sem))
    unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
  if (sem_post(sem))
    unix_error("V");
}
```


Sharing With POSIX Semaphores

```
/* goodcnt.c - properly sync'd
counter program */
#include "csapp.h"
#define NITERS 10000000

unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;

    Sem_init(&sem, 0, 1); /* sem=1 */

    /* create 2 threads and wait */
    ...

   if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}
```

```
/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

Signaling With Semaphores

Common synchronization pattern:

- Producer waits for slot, inserts item in buffer, and "*signals*" consumer.
- Consumer waits for item, removes it from buffer, and "signals" producer.
  - "signals" in this context has nothing to do with Unix signals

Examples

- Multimedia processing:
  - Producer creates MPEG video frames, consumer renders the frames
- Event-driven graphical user interfaces
  - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer.
  - Consumer retrieves events from buffer and paints the display.

Producer-Consumer on a Buffer That Holds One Item

```
/* buf1.c - producer-consumer
on 1-element buffer */
#include "csapp.h"

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
  int buf; /* shared var */
  sem_t full; /* sems */
  sem_t empty;
} shared;
```

```
int main() {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* initialize the semaphores */
  Sem_init(&shared.empty, 0, 1);
  Sem_init(&shared.full,  0, 0);

  /* create threads and wait */
  Pthread_create(&tid_producer, NULL,
                 producer, NULL);
  Pthread_create(&tid_consumer, NULL,
                 consumer, NULL);
  Pthread_join(tid_producer, NULL);
  Pthread_join(tid_consumer, NULL);

  exit(0);
}
```


Producer-Consumer (cont)

Initially: empty = 1, full = 0.

```
/* producer thread */
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* produce item */
    item = i;
    printf("produced %d\n",
           item);

    /* write item to buf */
    P(&shared.empty);
    shared.buf = item;
    V(&shared.full);
  }
  return NULL;
}
```

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* consume item */
    printf("consumed %d\n",
           item);
  }
  return NULL;
}
```


Thread Safety

Functions called from a thread must be *thread-safe*.

We identify four (non-disjoint) classes of thread-unsafe functions:

- Class 1: Failing to protect shared variables.
- Class 2: Relying on persistent state across invocations.
- Class 3: Returning a pointer to a static variable.
- Class 4: Calling thread-unsafe functions.

Thread-Unsafe Functions

Class 1: Failing to protect shared variables.

- Fix: Use P and V semaphore operations.
- Issue: Synchronization operations will slow down code.
- Example: `goodcnt.c`

Thread-Unsafe Functions (cont)

Class 2: Relying on persistent state across multiple function invocations.

- Random number generator relies on static state
- Fix: Rewrite function so that caller passes in all necessary state.

```
/* rand - return pseudo-random integer on 0..32767 */
int rand(void)
{
    static unsigned int next = 1;
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```


Thread-Unsafe Functions (cont)

Class 3: Returning a ptr to a `static` variable.

```
struct hostent
*gethostbyname(char name)
{
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

Fixes:

- 1. Rewrite code so caller passes pointer to `struct`.
  - » Issue: Requires changes in caller and callee.

```
hostp = Malloc(...));
gethostbyname_r(name, hostp);
```

- 2. *Lock-and-copy*
  - » Issue: Requires only simple changes in caller (and none in callee)
  - » However, caller must free memory.

```
struct hostent
*gethostbyname_ts(char *p)
{
  struct hostent *q = Malloc(...);
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;   /* copy */
  V(&mutex);
  return q;
}
```


Thread-Unsafe Functions

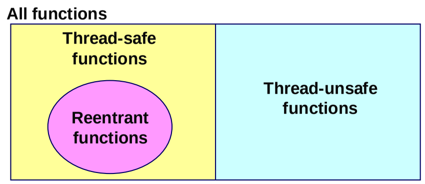
Class 4: Calling thread-unsafe functions.

- Calling one thread-unsafe function makes an entire function thread-unsafe.
- Fix: Modify the function so it calls only thread-safe functions

Reentrant Functions

A function is *reentrant* iff it accesses NO shared variables when called from multiple threads.

- Reentrant functions are a proper subset of the set of thread-safe functions.



- NOTE: The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant.

Thread-Safe Library Functions

All functions in the Standard C Library (at the back of your K&R text) are thread-safe.

- Examples: `malloc`, `free`, `printf`, `scanf`

Most Unix system calls are thread-safe, with a few exceptions:

Thread-unsafe function	Class	Reentrant version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	`(none)`
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

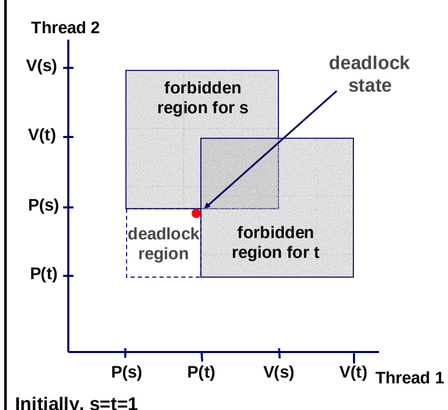
Races

A *race* occurs when the correctness of the program depends on one thread reaching point x before another thread reaches point y.

```
/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```


Deadlock



Locking introduces the potential for *deadlock:* waiting for a condition that will never be true.

Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state*, waiting for either s or t to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunate fact: deadlock is often non-deterministic.

Threads Summary

Threads provide another mechanism for writing concurrent programs.

Threads are growing in popularity

- **Somewhat cheaper than processes.**
- **Easy to share data between threads.**

However, the ease of sharing has a cost:

- **Easy to introduce subtle synchronization errors.**
- **Tread carefully with threads!**

For more info:

- **D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997.**