Synchronization
November 19, 2008

Topics
- Synchronizing with semaphores
- Races and deadlocks
- Thread safety and reentrancy

badcnt.c: An Improperly Synchronized Threaded Program

```c
/* shared */
volatile unsigned int cnt = 0;
define NITERS 100000000

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?!

Concurrent Execution

Key idea: In general, any sequentially consistent interleaving is possible, but some are incorrect!
- $i$, denotes that thread $i$ executes instruction $I$
- $\%eax$ is the contents of $\%eax$ in thread $i$’s context

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr$_i$</th>
<th>$%eax_1$</th>
<th>$%eax_2$</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H$_1$</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L$_1$</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S$_1$</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>H$_2$</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>L$_2$</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U$_2$</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S$_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>T$_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>T$_1$</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

OK
Concurrent Execution (cont)

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

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr</th>
<th>%eax1</th>
<th>%eax2</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U1</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H2</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L2</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!

Concurrent Execution (cont)

How about this ordering?

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr</th>
<th>%eax1</th>
<th>%eax2</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H2</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L2</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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 \((\text{Instr}_1, \text{Instr}_2)\). 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:

\[H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2\]
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 \geq 0)

Locking with Semaphores

Here is one way we could use P and V operations to synchronize the threads that update cnt:

- Semaphore used like this referred to as a “lock”

```c
/* 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 Locks**

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

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

**Wrappers on POSIX Semaphores**

```c
/* 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**

```c
/* properly sync'd counter program */
#include "csapp.h"
#define NITERS 10000000
volatile unsigned int cnt;
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;
    Sem_init(&sem, 0, 1); /* sem=1 */
    /* create 2 threads and wait */
    for (i=0; i<NITERS; i++) {
        P(&sem); cnt++; V(&sem);
    }
    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;
}
```

**One worry: races**

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

```c
/* 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;
}
```
Race Elimination

Make sure don’t have unintended sharing of state

```c
/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++) {
        int *valp = malloc(sizeof(int));
        *valp = i;
        Pthread_create(&tid[i], NULL, thread, valp);
    }
    for (i = 0; i < N; i++)
Pthread_join(tid[i], NULL);
exit(0);
}
/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

Another worry: Deadlock

- Processes wait for condition that will never be true

Typical Scenario
- Processes 1 and 2 needs two resources (A and B) to proceed
- Process 1 acquires A, waits for B
- Process 2 acquires B, waits for A
- Both will wait forever!

Deadlocking With POSIX Semaphores

```c
int main() {
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
Pthread_create(&tid[0], NULL, count, (void*) 1);
Pthread_create(&tid[1], NULL, count, (void*) 1);
Pthread_join(tid[0], NULL);
Pthread_join(tid[1], NULL);
printf("cnt=%d\n", cnt);
exit(0);
}
void *count(void *vargp) {
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    } 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 s0 or s1 to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunately fact: deadlock is often non-deterministic.
Avoiding Deadlock

Acquire shared resources in same order

```c
int main()
{
    pthread_t tid[2];
    sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    pthread_create(&tid[0], NULL, count, (void*) 0);
    pthread_create(&tid[1], NULL, count, (void*) 1);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}

void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]);
        P(&mutex[1]);
        cnt++;
        V(&mutex[id]);
        V(&mutex[1-id]);
    }
    return NULL;
}
```

Removed Deadlock

No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Crucial concept: Thread Safety

Functions called from a thread (without external synchronization) must be thread-safe

- i.e., it must be safe for multiple threads to be calling it concurrently

Some examples of thread-unsafe functions:

- Failing to protect shared variables
- Relying on persistent state across invocations
- Returning a pointer to a static variable
- Calling thread-unsafe functions

Thread-Unsafe Functions (1)

Failing to protect shared variables
- Fix: Use P and V semaphore operations
- Example: goodcnt.c
- Issue: Synchronization operations will slow down code
  - e.g., badcnt requires 0.5s, goodcnt requires 7.9s
Thread-Unsafe Functions (2)
Relying on persistent state across multiple function invocations
- Example: Random number generator that relies on static state

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

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

Making Thread-Safe RNG
Pass state as part of argument
- and, thereby, eliminate static state

```c
/* rand - return pseudo-random integer on 0..32767 */
int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- Consequence: programmer using rand must maintain seed

Thread-Unsafe Functions (cont)
Returning a ptr to a static variable
Fixes:
- 1. Rewrite code so caller passes pointer to struct
   - Issue: Requires changes in caller and callee
- 2. Lock-and-copy
   - Issue: Requires only simple changes in caller (and none in callee)
   - However, caller must free memory

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

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

Thread-Unsafe Functions
Calling thread-unsafe functions
- Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
- Fix: Modify the function so it calls only thread-safe functions ☺
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:

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Class</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>3</td>
<td>asctime_r</td>
</tr>
<tr>
<td>ctime</td>
<td>3</td>
<td>ctime_r</td>
</tr>
<tr>
<td>gethostbyaddr</td>
<td>3</td>
<td>gethostbyaddr_r</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>3</td>
<td>gethostbyname_r</td>
</tr>
<tr>
<td>inet_pton</td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td>localtime</td>
<td>3</td>
<td>localtime_r</td>
</tr>
<tr>
<td>rand</td>
<td>2</td>
<td>rand_r</td>
</tr>
</tbody>
</table>

Notifying With Semaphores

Common synchronization pattern:

- Producer waits for slot, inserts item in buffer, and notifies consumer.
- Consumer waits for item, removes it from buffer, and notifies producer.

Examples:

- Multimedia processing:
  - Producer creates MPEG video frames, consumer renders them.

- 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

```c
/* 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; /* sem */
  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 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);
    item = shared.buf;
    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.empty);
    item = shared.buf;
    printf("consumed %d\n", item);
    V(&shared.full);
  }
  return NULL;
}
```

Initially: empty = 1, full = 0.
Counting with Semaphores

Remember, it's a non-negative integer
- So, values greater than 1 are legal

Let's repeat thing_5() 5 times for every 3 of thing_3()

```c
/* thing_5() thread */
void *five_times(void *arg) {
    int i;
    while (1) {
        for (i=0; i<5; i++) {
            P(&five); // wait & thing_5()
            thing_5();
        } // end for
        V(&three); // release
        V(&three); // release
        V(&three); // release
    } // end while
    return NULL;
}
```

```c
/* thing_3() thread */
void *three_times(void *arg) {
    int i;
    while (1) {
        for (i=0; i<3; i++) {
            P(&three); // wait & thing_3()
            thing_3();
        } // end for
        V(&five); // release
        V(&five); // release
        V(&five); // release
        V(&five); // release
        V(&five); // release
    } // end while
    return NULL;
}
```

Initially: five = 5, three = 3

### 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:

### Beware of Optimizing Compilers!

```
#define NITERS 100000000
/* shared counter variable */
unsigned int cnt = 0;
/* thread routine */
void *count(void *arg) {
    int i;
    for (i = 0; i < NITERS; i++)
        cnt++;
    return NULL;
}
```

Generated Code:

```
movl cnt, %ecx
movl $99999999, %eax
.L6:
leal 1(%ecx), %edx
decl %eax
movl %edx, %ecx
jns .L6
movl %edx, cnt
```

- Compiler moved access to cnt out of loop
- Only shared accesses to cnt occur before loop (read) or after (write)
- What are possible program outcomes?