Concurrent Programming with Threads: Part II

Topics

- Progress graphs
- Semaphores
- Mutex and condition variables
- Barrier synchronization
- Timeout waiting
A version of `badcnt.c` with a simple counter loop

```c
int ctr = 0; /* shared */

/* main routine creates*/
/* two count threads */

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

    for (i=0; i<NITERS; i++)
        ctr++;
    return NULL;
}
```

note: counter should be equal to 200,000,000

```
linux> badcnt
BOOM! ctr=198841183

linux> badcnt
BOOM! ctr=198261801

linux> badcnt
BOOM! ctr=198269672
```

What went wrong?
Assembly code for counter loop

C code for counter loop

for (i=0; i<NITERS; i++)
    ctr++;

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

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

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

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

.L10:
```

Head (H_i)

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

Tail (T_i)
Concurrent execution

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

- $l_i$ denotes that thread $i$ executes instruction $l$
- $%eax_i$ 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>ctr</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>$U_1$</td>
<td>1</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>-</td>
<td>1</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_i</th>
<th>%eax_1</th>
<th>%eax_2</th>
<th>ctr</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>U_1</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H_2</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L_2</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S_1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T_1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U_2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S_2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T_2</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>instr₁</th>
<th>%eax₁</th>
<th>%eax₂</th>
<th>ctr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>H₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T₂</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can clarify our understanding of concurrent execution with the help of the *progress graph*
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 (Inst₁, Inst₂).

E.g., (L₁, S₂) denotes state where thread 1 has completed L₁ and thread 2 has completed S₂.
Legal state transitions

Interleaved concurrent execution (one processor):

or

Parallel concurrent execution (multiple processors)

or

(or)

(parallel execution)

Key point: Always reason about concurrent threads as if each thread had its own CPU.
A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L2, 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 ctr.

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

Sets of states where such interleaving occurs form *unsafe regions*. 
Def: A safe trajectory is a sequence of legal transitions that does not touch any states in an unsafe region.

Claim: Any safe trajectory results in a correct value for the shared variable \( ctr \).
Unsafe trajectories

Thread 1

Touching a state of type x is always incorrect.

Touching a state of type y may or may not be OK:
- correct because store completes before load.
- incorrect because order of load and store are indeterminate.

Moral: be conservative and disallow all unsafe trajectories.

Thread 2

Unsafe region

(O,O) -> H1 -> L1 -> U1 -> S1 -> T1

Critical section

Critical section
Semaphore operations

**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)`
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.

Semaphore used in this way is often called a mutex (mutual exclusion).

Initially, s = 1
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");
}
/* goodcnt.c - properly synch’d */
/* version of badcnt.c */
#include <ics.h>
define NITERS 10000000

void *count(void *arg);

struct {
    int ctr;    /* shared ctr */
    sem_t mutex; /* semaphore */
} shared;

int main() {
    pthread_t tid1, tid2;

    /* init mutex semaphore to 1 */
    Sem_init(&shared.mutex, 0, 1);

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

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

    for (i=0; i<NITERS; i++) {
        P(&shared.mutex);
        shared.ctr++;
        V(&shared.mutex);
    }
    return NULL;
}
Progress graph for goodcnt.c

Initially, mutex = 1
Semaphores introduce 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.
A deterministic deadlock

Initially, s = 1, t = 0.

Sometimes though, we get "lucky" and the deadlock is deterministic.

Here is an example of a deterministic deadlock caused by improperly initializing semaphore t.

**Problem:** correct this program and draw the resulting forbidden regions.
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.

Examples
- Multimedia processing:
  - Producer creates MPEG video frames, consumer renders the frames
- 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 (1-buffer)

```c
/* buf1.c - producer-consumer on 1-element buffer */
#include <ics.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);
}
```

/* buf1.c - producer-consumer on 1-element buffer */
#include <ics.h>

#define NITERS 5

#include <ics.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);
}
Initially: empty = 1, full = 0.

```c
/* 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;
}
```
Producer-consumer progress graph

The forbidden regions prevent the producer from writing into a full buffer.

They also prevent the consumer from reading an empty buffer.

**Problem:** Write version for n-element buffer with multiple producers and consumers.

Initially, empty = 1, full = 0.
Limitations of semaphores

Semaphores are sound and fundamental, but they have limitations.

- **Difficult to broadcast a signal to a group of threads.**
  - e.g., *barrier synchronization*: no thread returns from the barrier function until every other thread has called the barrier function.

- **Impossible to do timeout waiting.**
  - e.g., wait for at most 1 second for a condition to become true.

For these we must use Pthreads mutex and condition variables.
Basic operations on mutex variables

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr)
```

Initializes a mutex variable (`mutex`) with some attributes (`attr`).
- attributes are usually NULL.
- like initializing a mutex semaphore to 1.

```c
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Indivisibly waits for `mutex` to be unlocked and then locks it.
- like P(`mutex`)

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Unlocks `mutex`.
- like V(`mutex`)
Basic operations on condition variables

```c
int pthread_cond_init(pthread_cond_t *cond,
                       pthread_condattr_t *attr)
```

Initializes a condition variable (cond) with some attributes (attr).

- attributes are usually NULL.

```c
int pthread_cond_signal(pthread_cond_t *cond)
```

Awakens one thread waiting on condition cond.

- if no threads waiting on condition, then it does nothing.
- key point: signals are not queued!

```c
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Indivisibly unlocks mutex and waits for signal on condition cond

- When awakened, indivisibly locks mutex.
Advanced operations on condition variables

```c
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Awakens *all* threads waiting on condition `cond`.
- if no threads waiting on condition, then it does nothing.

```c
int pthread_cond_timedwait(pthread_cond_t *cond,
        pthread_mutex_t *mutex,
        struct timespec *abstime)
```

Waits for condition `cond` until absolute wall clock time is `abstime`
- Unlocks `mutex` on entry, locks `mutex` on awakening.
- Use of absolute time rather than relative time is strange.
Signaling and waiting on conditions

Basic pattern for signaling

Pthread_mutex_lock(&mutex);
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&mutex);

A mutex is always associated with a condition variable.

Guarantees that the condition cannot be signaled (and thus ignored) in the interval when the waiter locks the mutex and waits on the condition.

Basic pattern for waiting

Pthread_mutex_lock(&mutex);
Pthread_cond_wait(&cond, &mutex);
Pthread_mutex_unlock(&mutex);
#include <ics.h>

static pthread_mutex_t mutex;
static pthread_cond_t cond;
static int nthreads;
static int barriercnt = 0;

void barrier_init(int n) {
    nthreads = n;
    Pthread_mutex_init(&mutex, NULL);
    Pthread_cond_init(&cond, NULL);
}

void barrier() {
    Pthread_mutex_lock(&mutex);
    if (++barriercnt == nthreads) {
        barriercnt = 0;
        Pthread_cond_broadcast(&cond);
    } else
    Pthread_cond_wait(&cond, &mutex);
    Pthread_mutex_unlock(&mutex);
}

Barrier synchronization

Call to barrier will not return until every other thread has also called barrier.

Needed for tightly-coupled parallel applications that proceed in phases. E.g., physical simulations.
timebomb.c: timeout waiting example

A program that explodes unless the user hits a key within 5 seconds.

```c
#include <ics.h>
#define TIMEOUT 5

/* function prototypes */
void *thread(void *vargp);
struct timespec *maketimeout(int secs);

/* condition variable and its associated mutex */
pthread_cond_t cond;
pthread_mutex_t mutex;

/* thread id */
pthread_t tid;
```
A routine for building a timeout structure for `pthread_cond_timewait`.

```c
/*
 * maketimeout - builds a timeout object that times out
 *      in secs seconds
 *
  struct timespec *maketimeout(int secs) {
    struct timeval now;
    struct timespec *tp =
      (struct timespec *)malloc(sizeof(struct timespec));

    gettimeofday(&now, NULL);
    tp->tv_sec = now.tv_sec + secs;
    tp->tv_nsec = now.tv_usec * 1000;
    return tp;
  }
*/
```
int main() {
    int i, rc;

    /* initialize the mutex and condition variable */
    Pthread_cond_init(&cond, NULL);
    Pthread_mutex_init(&mutex, NULL);

    /* start getchar thread and wait for it to timeout */
    Pthread_mutex_lock(&mutex);
    Pthread_create(&tid, NULL, thread, NULL);
    for (i=0; i<TIMEOUT; i++) {
        printf("BEEP\n");
        rc = pthread_cond_timedwait(& cond, &mutex, maketimeout(1));
        if (rc != ETIMEDOUT) {
            printf("WHEW!\n");
            exit(0);
        }
    }
    printf("BOOM!\n");
    exit(0);
}
Thread routine for timebomb.c

```c
/*
 * thread - executes getchar in a separate thread
 */
void *thread(void *vargp) {

    (void) getchar();

    Pthread_mutex_lock(&mutex);
    Pthread_cond_signal(&cond);
    Pthread_mutex_unlock(&mutex);
    return NULL;
}
```
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.

For more info:
  • man pages (man -k pthreads)