Introduction to Computer Systems
15-213/18-243, fall 2009
22nd Lecture, Nov. 17

Instructors:
Roger B. Dannenberg and Greg Ganger

Today
- Threads: review basics
- Synchronization
- Races, deadlocks, thread safety

Process: Traditional View
- Process = process context + code, data, and stack

Process: Alternative View
- Process = thread + code, data, and kernel context

Process with Two Threads

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

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

- Thread-based servers:
  - Use "detached" threads to avoid memory leaks
  - At any point in time, a thread is either joinable or detached
  - Joinable thread can be reaped and killed by other threads
  - Detached thread cannot be reaped or killed by other threads

- Must be careful to avoid unintended sharing
  - For example, what happens if we pass the address of connfd to the thread routine?
    - Pthread_create(tid, NULL, thread, (void *)&connfd);

Pros and Cons of Thread-Based Designs

- + Easy to share data structures between threads
- + Threads are more efficient than processes

- Unintentional sharing can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads

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 each memory instance?
  - How many threads might reference each of these instances?

Threads Memory Model

- Conceptual model:
  - Multiple threads run within the context of a single process
  - Each thread has its own separate thread 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:
  - Register values are truly separate and protected, but
  - 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

Thread Accessing Another Thread’s Stack

```c
char **ptr; /* global */
int main()
{
  int i;
  pthread_t tid;
  char *msgs[2] = {
    "Hello from foo",
    "Hello from bar"
  };
  ptr = msgs;
  pthread_create(&tid, NULL, thread, (void *)i);
  for (i = 0; i < 2; i++)
    pthread_create(&tid, NULL, thread, (void *)&(si));
  pthread_exit(NULL);
}

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

Peer threads access main thread’s stack indirectly through global ptr variable
Mapping Variables to Memory Instances

Global var: 1 instance (ptr [data])
Local var: 1 instance (cnt, msgs, x)

badcnt.c: Improper Synchronization

Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable</th>
<th>Referenced by</th>
<th>Referenced by</th>
<th>Referenced by</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>avar</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Myid_p0</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Myid_p1</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

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

- ptr, avar, and msgs are shared
- i and myid are not shared

Assembly Code for Counter Loop

C code for counter loop in thread i

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

Corresponding assembly code

Key: Head (H) Load cnt (L) Store cnt (S) Tail (T)

i(thread) instr | %eax | %eax | %eax | cnt
1 M1 - - - 0
1 L1 0 - - 0
1 S1 0 - - 0
1 T1 - 1 - 1
2 L2 - - 2 1
2 S2 - 2 - 2
2 T2 - 2 - 2
1 L1 1 - - 2

Key: Load Update Store

Concurrent Execution

Key idea: In general, any sequentially consistent interleaving is possible, but some give an unexpected result!

- I denotes that thread i executes instruction I
- %eax is the content of %eax in thread i’s context

OK

Concurrent Execution (cont)

Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2
Concurrent Execution (cont)

How about this ordering?

<table>
<thead>
<tr>
<th>(thread)</th>
<th>instr</th>
<th>Stmtx1</th>
<th>Stmtx2</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H1</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>U1</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>2</td>
<td>T2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We can analyze the behavior using a process 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 (Inst1, Inst2).

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

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.

Definition: A trajectory is safe iff it does not enter any unsafe region.

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

Critical Sections and Unsafe Regions

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 global integer synchronization variable
  - P(s): \[\text{while} (s == 0) \text{wait}(); s--;\]
    - 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: \(s >> 0\)
badcnt.c: Improper Synchronization

```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;
}
```

How to fix using semaphores?

```c
/* Semaphore s is initially 1 */
/* Thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

Safe Sharing with Semaphores

- One semaphore per shared variable
- Initially set to 1
- Here is how we would use P and V operations to synchronize the threads that update cnt:

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");
}
```

Warning:
It's really slow!

Sharing With POSIX Semaphores

```c
/* properly sync'd counter program */
#include "csapp.h"
#define NITERS 100000000
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 */
    ...
    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);
        P(&sem);
    }
    return NULL;
}
```

Today

- Threads: basics
- Synchronization
- Races, deadlocks, thread safety
One worry: races

- A race occurs when 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;
}
```

Where is the race?

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*) 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[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

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[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```
Avoided Deadlock in Progress Graph

Thread 2
V(s0) P(s0) V(s1) P(s1)

Thread 1
V(s0) P(s0) V(s1) P(s1)

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
- Meaning: it must always produce correct results when called repeatedly from multiple concurrent threads

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

Avoided Deadlock in Progress Graph

Thread 2
V(s0) P(s0) V(s1) P(s1)

Thread 1
V(s0) P(s0) V(s1) P(s1)

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
- Meaning: it must always produce correct results when called repeatedly from multiple concurrent threads

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 (Class 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 (Class 2)

- Relying on persistent state across multiple function invocations
  - Example: Random number generator (RNG) that relies on static state

/* 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;
}

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

Thread-Unsafe Functions (Class 3)

- Returning a ptr to a static variable
  - Fix:
    - 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

Making Thread-Safe RNG

- Pass state as part of argument
  - and, thereby, eliminate static state

/* 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 (Class 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 (Class 2)

- Relying on persistent state across multiple function invocations
  - Example: Random number generator (RNG) that relies on static state

/* 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;
}

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

Thread-Unsafe Functions (Class 3)

- Returning a ptr to a static variable
  - Fix:
    - 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

Making Thread-Safe RNG

- Pass state as part of argument
  - and, thereby, eliminate static state

/* 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 (Class 4)

- 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_ntoa</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>

Threads Summary

- Threads provide another mechanism for writing concurrent programs
- Threads are very popular
  - Somewhat cheaper than processes
  - Easy to share data between threads
  - Make use of multiple cores for parallel algorithms
- However, the ease of sharing has a cost:
  - Easy to introduce subtle synchronization errors
  - Thread carefully with threads!

- For more info:
  - D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997