Introduction to Computer Systems
15-213/18-243, spring 2009
27th Lecture, Apr. 30th

Instructors:
Gregory Kesden and Markus Püschel
Today

- Races, deadlocks, thread safety
- Multi-code
- Thread Level Parallelism (TLP)
- Simultaneous Multi-Threading (SMT)
Another worry: Deadlock

- Processes wait for condition that will never be true

**Typical Scenario**
- Processes 1 and 2 need 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;
}
```
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 $S_0$ or $S_1$ to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunate 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;
}
```

Tid[0]:
P(s0);
P(s1);
cnt++;
V(s0);
V(s1);

Tid[1]:
P(s0);
P(s1);
cnt++;
V(s1);
V(s0);
Avoided Deadlock in Progress Graph

No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Forbidden region for $s_1$

Forbidden region for $s_2$
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

```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;
}

/* 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 (Class 3)

- 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;
}

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

```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 (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>
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; /* 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 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;
}
Counting with Semaphores

- Remember, it’s a non-negative integer
  - So, values greater than 1 are legal
- Lets repeat thing_5() 5 times for every 3 of thing_3()

```c
/* thing_5 and thing_3 */
#include “csapp.h”

sem_t five;
sem_t three;

void *five_times(void *arg);
void *three_times(void *arg);

int main() {
    pthread_t tid_five, tid_three;

    /* initialize the semaphores */
    Sem_init(&five, 0, 5);
    Sem_init(&three, 0, 3);

    /* create threads and wait */
    Pthread_create(&tid_five, NULL,
                   five_times, NULL);
    Pthread_create(&tid_three, NULL,
                   three_times, NULL);
    .
    .
    .
    }
```
Counting with semaphores (cont)

Initially: five = 5, three = 3

```c
/* thing_5() thread */
void *five_times(void *arg) {
    int i;

    while (1) {
        for (i=0; i<5; i++) {
            /* wait & thing_5() */
            P(&five);
            thing_5();
        }
        V(&three);
        V(&three);
        V(&three);
    }
    return NULL;
}

/* thing_3() thread */
void *three_times(void *arg) {
    int i;

    while (1) {
        for (i=0; i<3; i++) {
            /* wait & thing_3() */
            P(&three);
            thing_3();
        }
        V(&five);
        V(&five);
        V(&five);
        V(&five);
        V(&five);
    }
    return NULL;
}
```
Producer-Consumer on a Buffer That Holds More than One Item

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

#define NITERS 5
#define NITEMS 7

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

struct {
    void *buf[NITEMS];
    int cnt;
    sem_t full; /* sems */
    sem_t empty;
    sem_t mutex;
} shared;

int main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;

    /* initialization */
    Sem_init(&shared.empty, 0, NITEMS);
    Sem_init(&shared.full, 0, 0);
    Sem_init(&shared.mutex, 0, 1);
    shared.cnt = 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 = all slot, full = no slots, e.g. 0

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

    for (i=0; i<NITERS; i++) {
        /* write item to buf */
        P(&shared.empty);
        P(&shared.mutex);

        shared.buf[shared.cnt++] = produceItem();

        V(&shared.mutex);
        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);
        P(&shared.mutex);

        item=shared.buf[shared.cnt--];

        V(&shared.mutex);
        V(&shared.empty);

        /* consume item */
        printf("consumed %d\n", item);
    }
    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
  - Tread carefully with threads!

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

- CPU chip
  - register file
  - ALU
  - bus interface
- system bus
- I/O bridge
- memory bus
- main memory
- I/O bus
- USB controller
  - mouse keyboard
- graphics adapter
  - monitor
- disk controller
  - disk

Expansion slots for other devices such as network adapters.
Single-core CPU chip

![Diagram of a single-core CPU chip with components labeled: CPU chip, register file, ALU, system bus, and bus interface. The diagram highlights the single core.](image-url)
Multi-core architectures

- This lecture is about a new trend in computer architecture: Replicate multiple processor cores on a single die.
Within each core, threads are time-sliced (just like on a uniprocessor)
Interaction With the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, ...
Why multi-core?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)
Instruction-level parallelism

• Parallelism at the machine-instruction level
• The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
• Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years
Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP
A technique complementary to multi-core: Simultaneous multithreading

- Problem addressed: The processor pipeline can get stalled:
  - Waiting for the result of a long floating point (or integer) operation
  - Waiting for data to arrive from memory

Other execution units wait unused

Source: Intel
Simultaneous multithreading (SMT)

- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads” on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units
Without SMT, only a single thread can run at any given time

Thread 1: floating point
SMT processor: both threads can run concurrently

Thread 1: floating point
Thread 2: integer operation
But: Can’t simultaneously use the same functional unit

This scenario is impossible with SMT on a single core (assuming a single integer unit)
SMT Dual-core: all four threads can run concurrently
Comparison: multi-core vs SMT

- **Multi-core:**
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism

- **SMT**
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism