Introduction to Computer Systems
15-213/18-243, fall 2009
23rd Lecture, Nov 19

Instructors:
Roger B. Dannenberg and Greg Ganger
Introduction

- **Last Time:**
  - Threads
  - Concepts
  - Synchronization with Semaphores

- **Today: Synchronization in Greater Depth**
  - Producer Consumer
  - Buffer Pools and Condition Variables
  - Message Queues
  - Message Passing
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

/* buf1.c - producer-consumer on 1-element buffer */
#include "csapp.h"
#define NITERS 5

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

void main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;

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

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

    /* 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

```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;
}
```
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);
```

```c
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, Circular Buffer

/* bufn.c - producer-consumer on n-element buffer */
#include "csapp.h"

#define NITERS 100
#define N 20 /* buffer size */

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

struct {
  int buf[N]; /* 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, N);
  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, Circular Buffer (cont)

Initially, shared.full = 0, shared.empty = N

```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[i%N] = 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[i%N];
        V(&shared.empty);

        /* consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```
Do You Need Semaphores?

/* data structure */
int shared;

/* initially */
shared = 0;

/* producer */
while (shared != 0) ;
    shared = any_non_zero_data;

/* consumer */
while (shared == 0) ;
data_to_consume = shared;
shared = 0;

Notes:
  ▪ Producer notifies Consumer by writing non-zero
  ▪ Consumer notifies Producer by writing zero
  ▪ Don’t try this at home! (loops, hot-spot)
  ▪ This is just a mental warm-up
Do You Need Semaphores? (cont)

/* data structure */
#define N 20
struct {
    int head;
    int tail;
    int buf[N];
} shared;

/* initialize */
shared.head = 0;
Shared.tail = 0;

/* producer */
int i = (shared.tail + 1) % N;
if (i == shared.head) return FAIL;
shared.buf[i] = produced_data;
tail = i; /* atomic update */

/* consumer */
if (shared.tail == shared.head)
    return FAIL;
data_to_consume = shared.buf[shared.head];
/* atomic update: */
shared.head = (shared.head + 1) % N;

Notes:
- Producer notifies Consumer by updating tail
- Consumer notifies Producer by updating head
- Data copied in before notifying consumer
- Data copied out before notifying producer
- What about out-of-order writes?
Sharing With Dependencies

- Threads interact through a resource manager
- (Standard terminology: monitor)
- One thread may need to wait for something another thread provides
- Example 1: previous circular buffer
  - But that had a simple semaphore solution
- Example 2: memory buffer pool
  - Multiple threads checking out and returning buffers
  - Might have to wait for buffer of the right size to show up

- Ultimately, our solution will be: mutex and condition variable
Naïve Solution

```c
/* protected access to buffer structures */
mutex_t mutex;
Mutex_init(&mutex);

/* get a buffer */
Mutex_lock(&mutex);
Find a suitable buffer
Mutex_unlock(&mutex);

/* return a buffer */
Mutex_lock(&mutex);
Return buffer to pool
Mutex_unlock(&mutex);
```

### Notes:
- `mutex_lock` is functionally similar to `P()`
- `mutex_unlock` is similar to `V()`
- Analog of `Mutex_init()` is initializing semaphore to 1
What If Resource Is Unavailable?

```c
/* protected access to buffer structures */
mutex_t mutex;
Mutex_init(&mutex);

/* get a buffer */
Mutex_lock(&mutex);
while (!Find_a_suitable_buffer()) {
    Mutex_unlock(); Mutex_lock();
}
Reserve the suitable buffer
Mutex_unlock(&mutex);

/* return a buffer */
Mutex_lock(&mutex);
Return buffer to pool
Mutex_unlock(&mutex);
```

- **Bad**: spins rather than yielding processor to threads holding the very resources we are waiting for!
What If Resource Is Unavailable? (cont)

```c
/* protected access to buffer structures */
mutex_t mutex;
Mutex_init(&mutex);

/* get a buffer */
Mutex_lock(&mutex);
while (!Find_a_suitable_buffer()) {
    Mutex_unlock(); Yield(); Mutex_lock();
}
Reserve the suitable buffer
Mutex_unlock(&mutex);

/* return a buffer */
Mutex_lock(&mutex);
Return buffer to pool
Mutex_unlock(&mutex);
```

Better, but not really acceptable. Polls until buffer is available.
How About Waiting for a Notification?

/* protected access to buffer structures */
mutex_t mutex;
Mutex_init(&mutex);
int waiting = FALSE;
sem_t rsrc;
Sem_init(&rsrc, 0, 0);

/* get a buffer */
Mutex_lock(&mutex);
while (!Find_a Suitable_buffer()) {
    waiting = TRUE;
    Mutex_unlock(&mutex);
    P(&rsrc);
    Mutex_lock(&mutex);
}
Reserve the suitable buffer
Mutex_unlock(&mutex);

/* return a buffer */
Mutex_lock(&mutex);
Return buffer to pool
if (waiting) { waiting = FALSE; V(&rsrc); }
Mutex_unlock(&mutex);
How About Waiting for a Notification?

```c
/* protected access to buffer structures */
mutex_t mutex;
Mutex_init(&mutex);

/* get a buffer */
Mutex_lock(&mutex);

Problems:
• This will not work with multiple waiting threads
  • waiting is only a Boolean
• Hard to analyze
  • Need a more general approach
  • Special-case “hacks” invite obscure bugs

/* return a buffer */
Mutex_lock(&mutex);
Return buffer to pool
if (waiting) { waiting = FALSE; V(&rsrc); }
Mutex_unlock(&mutex);
```
Condition Variable

- Yet another synchronization mechanism
  - Not a semaphore, not a mutex
  - Not really a “variable”
- Essentially a queue of waiting/sleeping/suspended threads
- Operations:
  - Cond_wait puts thread on the queue
  - Cond_signal wakes up one thread on the queue (if any)
  - Cond_broadcast wakes up all threads on the queue
- Special feature of Condition Variables:
  - Cond_wait atomically puts thread on queue and releases a mutex lock
  - Waking up automatically reacquires the lock (!)
Sharing with Condition Variable

```c
/* protected access to buffer structures */
mutex_t mutex;
Mutex_init(&mutex);
cond_t bufcv;
Cond_init(&bufcv);

/* get a buffer */
Mutex_lock(&mutex);
while (!Find_a_suitable_buffer()) {
  Cond_wait(&condcv, &mutex);
}
Reserve the suitable buffer
Mutex_unlock(&mutex);

/* return a buffer */
Mutex_lock(&mutex);
Return buffer to pool
Cond_broadcast(&condcv);
Mutex_unlock(&mutex);
```

- This is a general pattern: enter mutex, loop testing for what you need to proceed, cond_wait() if you cannot, finally finish up and leave mutex;

- Other operations within the monitor always: cond_signal or cond_broadcast if they possibly enable a blocked thread

- Note: posix allows spurious wakeups to occur, so always retest condition in a loop
Another Problem: Readers and Writers

- Consider multiple readers and multiple writers of some shared data.
- Writers must access the data exclusively (no other readers or writers at the same time)
- Readers must exclude writers, but multiple readers can read at the same time
- How would you implement Readers and Writers?
- How would you give priority to Writers?
- How would you prevent Writers from preventing Readers from (ever) making progress?
Synchronization and Message Passing

- Advantages of threads over processes seem to require shared memory (and all the associated problems)
- There’s no requirement that threads share variables
- Threads can communicate through messages even in a shared address space
- In fact, shared address space allows for lightweight message passing
- Let’s consider:
  - How to implement message passing
  - Solutions to some common synchronization problems
- Note: message passing is not covered in the textbook
Message Passing Implementation

- Many implementations are possible
- Rare to see primitives in languages or systems
- Usually built using synchronization primitives
- **Design Issues:**
  - Do you send actual message data or just a pointer to it?
  - Data structures to use for messages and queues
  - How to name recipients of messages
Simple Message Implementation

- Messages are send to and received from a *mailbox*
- Mailbox is just a linked list of pointers to messages
- Implementation (interim version, no synchronization):

```c
/* assume Queue datatype */
typedef struct { queue_t q; } mbox_t;

mbox_t my_mb;

/* sending a message */
void mb_init(mbox_t *mb) {
    queue_init(&(mb->q)); }

void mb_snd(mbox_t *mb, void *msg) {
    mb->q.enqueue(msg); }

/* receiving a message */
void *mb_snd(mbox_t *mb) {
    if (mb->q.empty()) return NULL;
    return mb->q.dequeue(); }

int mb_empty(mbox_t *mb) {
    return mb->q.empty(); }

void *msg = malloc(MSG_SIZE);
... fill in msg with data ...
mb_snd(&my_mb, msg);
... do not free msg! ...

/* receiving a message */
void *msg = mb_rcv(&my_mb);
if (msg) {
    ... use data in *msg ...
    free(msg);
}
```

/* assume Queue datatype */
typedef struct { queue_t q; } mbox_t;

mbox_t my_mb;

/* sending a message */
void mb_init(mbox_t *mb) {
    queue_init(&(mb->q)); }

void mb_snd(mbox_t *mb, void *msg) {
    mb->q.enqueue(msg); }

/* receiving a message */
void *mb_snd(mbox_t *mb) {
    if (mb->q.empty()) return NULL;
    return mb->q.dequeue(); }

int mb_empty(mbox_t *mb) {
    return mb->q.empty(); }

void *msg = malloc(MSG_SIZE);
... fill in msg with data ...
mb_snd(&my_mb, msg);
... do not free msg! ...

/* receiving a message */
void *msg = mb_rcv(&my_mb);
if (msg) {
    ... use data in *msg ...
    free(msg);
}
Note About Message Types

- We can’t really work with messages of type `void *`.
- Typically use something like this:

```c
enum Msg_type {start, stop, task1, task2 };

typedef struct {
    enum Msg_type tag;
    union {
        struct { ... } start_data;
        struct { ... } stop_data;
        struct { ... } task1_data;
        struct { ... } task2_data;
    }
} Message;
```
Synchronization

Problem 1: Shared access to Mailboxes (and Queues)

```c
/* assume Queue datatype */
typedef struct { Queue q;    sem_t s; } mbox_t;

void mb_init(mbox_t *mb) {
    queue_init(&((mb)->q));
    Sem_init(&((mb)->s, 0, 1));
}

void mb_snd(mbox_t *mb, void *msg) {
    P(&((mb)->s));
    mb->q.enqueue(msg);
    V(&((mb)->s));
}

int mb_empty(mbox_t *mb) {
    int empty;
    P(&((mb)->s));
    empty = mb->q.empty();
    V(&((mb)->s));
    return empty;
}

void *mb_rcv(mbox_t *mb) {
    void *m;
    P(&((mb)->s));
    if ((mb)->q.empty()) {
        V(&((mb)->s));
        return NULL;
    }
    m = mb->q.dequeue();
    V(&((mb)->s));
}
```
Synchronization (cont)

Problem 2: Waiting for a message: rewrite with condition var

```c
/* assume Queue datatype */
typedef struct { Queue q;
    mutex_t s;
    cond_t rdy; } mbox_t;

void mb_init(mbox_t *mb) {
    queue_init(&(mb->q));
    Mutex_init(&(mb->s));
    Cond_init(&(mb->rdy));
}

void mb_snd(mbox_t *mb, void *msg) {
    mutex_lock(&(mb->s));
    mb->q.enqueue(msg);
    cond_signal(&(mb->rdy));
    mutex_unlock(&(mb->s));
}

int mb_empty(mbox_t *mb) {
    int empty;
    mutex_lock(&(mb->s));
    empty = mb->q.empty();
    mutex_unlock(&(mb->s));
    return empty;
}

void *mb_rcv(mbox_t *mb) {
    void *m;
    mutex_lock(&(mb->s));
    while (mb->q.empty()) {
        cond_wait(&(mb->rdy),
                   &(mb->s));
    }
    m = mb->q.dequeue();
    mutex_unlock(&(mb->s));
}
```
Example Message Passing Application

- Consider an Audio Player
  - User interface
    - sends filename, EQ, volume, position to audio thread
    - displays song pos. and spectrum
  - Audio thread
    - reads, decodes, plays audio
    - computes spectrum data

- Possible implementation
  - 2 mailboxes: one for UI, one for Audio thread
  - Each thread: check for msgs, do work, repeat
  - No shared variables except for mailboxes:
    - No shared access to screen
    - Display updates unlikely to block time-critical audio thread
Beware of Optimizing Compilers!

**Code From Book**

```c
#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**

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

- Global variable `cnt` shared between threads
- Multiple threads could be trying to update within their iterations
- 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?
#define NITERS 100000000 /* shared counter variable */
volatile unsigned int cnt = 0; /* thread routine */

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

- Declaring variable as volatile forces it to be kept in memory
- Shared variable read and written each iteration
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
  - Tread carefully with threads!

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