Process Synchronization

Andrew Case

Slides adapted from Mohamed Zahran, Clark Barrett, Jinyang Li, Randy Bryant and Dave O’Hallaron
Topics

- Sharing data
- Race conditions
- Mutual exclusion and semaphores
- Sample problems
- Deadlocks
Multi-Threaded process

- A process can have multiple threads
  - Each thread has its own logical control flow (PC, registers, etc.)
  - Each thread shares the same code, data, and kernel context

![Diagram of multi-threaded process](image)

- Thread 1 (main thread)
  - stack 1
  - Thread 1 context:
    - Data registers
    - Condition codes
    - SP1
    - PC1

- Thread 2 (peer thread)
  - stack 2
  - Thread 2 context:
    - Data registers
    - Condition codes
    - SP2
    - PC2

- Shared code and data
  - run-time heap
  - read/write data
  - read-only code/data

- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer
Threads Memory Model

- Conceptual model:
  - Multiple threads run within the context of a single process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
  - All threads share the remaining process 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 protected, but...
  - Any thread can read and write to entire virtual address space (any threads stack)
Mapping Variable Instances to Memory

- **Global variables**
  - *Def:* Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- **Local variables**
  - *Def:* Variable declared inside function without static attribute
  - Each thread stack contains one instance of each local variable

- **Local static variables**
  - *Def:* Variable declared inside function with the static attribute
  - Virtual memory contains exactly one instance of any local static variable.
Mapping Variable Instances to Memory

**Global var:** 1 instance in ‘data’ memory

**Local vars:** 1 instance in main thread stack

```c
char **ptr; /* global */
int main()
{
    int i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;

    for (i = 0; i < 2; i++)
        pthread_create(&tid, NULL, thread, (void *)i);

    pthread_exit(NULL);
}

/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}
```

**Local var:** 2 instances (myid in peer thread 0’s stack, myid in peer thread 1’s stack)

**Local static var:** 1 instance in ‘data’ memory

`linux> ./a.out [0]: Hello from foo (cnt=1) [1]: Hello from foo (cnt=2)`
Problem: Race conditions

- A *race condition* occurs when correctness of the program depends on one thread or process reaching point \( x \) before another thread or process reaches point \( y \).

- Any ordering of instructions from multiple threads may occur.
Race Condition: Example

```c
int main() {
    ...
    int cnt;
    for (cnt = 0; cnt < 100; cnt++) {
        /* send an cnt to each thread */
        pthread_create(&tid, NULL, thread, &cnt);
    }
}

void *thread(void *p)
{
    int i = *((int *)p); /* cast to an int ptr and de-ref */
    pthread_detach(pthread_self());
    save_value(i); /* do something with the count */
    return NULL;
}
```

- **Race Test**
  - If no race, then each thread would get different value of cnt
  - Set of saved values would consist of one copy each of 0 through 99.
Race Condition: Example

```c
... 
1: int cnt;
2: for (cnt = 0; cnt < 100; cnt++) {
3:   pthread_create(&tid, NULL,
                   thread, &cnt);
4: }
...
5: void *thread(void *p) {
6:   int i = *((int *)p);
7:   pthread_detach(pthread_self());
8:   save_value(i);
9:   return NULL;
10: }
```

<table>
<thead>
<tr>
<th>Line of code run</th>
<th>i (in thread)</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Problem: i in worker thread should have been set to 0
Experimental Results

(Number of times each value is saved)

No Race

Single core laptop

Multicore server

The race can really happen!
int main() {
    ...
    int cnt;
    for (cnt = 0; cnt < 100; cnt++) {
        int *j = malloc(sizeof(int)); /* create separate var */
        *j = cnt;                    /* for each thread */
        pthread_create(&tid, NULL, thread, j);
    }
}

void *thread(void *p)
{
    int i = *((int *)p);
    free(p);                       /* free it when done */
    pthread_detach(pthread_self());
    save_value(i);
    return NULL;
}
Race Condition: Problem

... 1: int cnt;
2: for (cnt = 0; cnt < 100; cnt++) {
3:   int *j = malloc(sizeof(int));
4:   *j = cnt;
5:   pthread_create(&tid, NULL, thread, j);
6: }
...

7: void *thread(void *p) {
8:   int i = *((int *)p);
9:   free(p);
10:  pthread_detach(pthread_self());
11:  save_value(i);
12:  return NULL;
13: }

<table>
<thead>
<tr>
<th>Line of code</th>
<th>i</th>
<th>Cnt (global)</th>
<th>*j</th>
<th>*p</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>0</td>
<td>?</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Topics

- Sharing data
- Race conditions
- Mutual exclusion and Semaphores
- Sample problems
- Deadlocks
Improper Synchronization: badcnt.c

```c
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
    int niters = atoi(argv[1]);
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL,
                    thread, &niters);
    pthread_create(&tid2, NULL,
                    thread, &niters);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

void *thread(void *vargp)
{
    /* local data */
    int i;
    int niters = *((int *)vargp);
    for (i = 0; i < niters; i++)
        cnt++; /* shared data */

    return NULL;
}
```

```bash
linux> ./badcnt 10000
OK cnt=20000

linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt should equal 20,000.

What went wrong?
Assembly Code for Counter Loop

C code for counter loop in thread i

```c
for (i=0; i < niters; i++)
    cnt++; /* changing shared data */
```

Corresponding assembly code

```
.movl (%rdi),%ecx
.movl $0,%edx
.cmpl %ecx,%edx
.jge .L13
.L11:
```

Critical Regions: changing shared data

```
.movl cnt(%rip),%eax
.incl %eax
.movl %eax,cnt(%rip)
```

Head (loop iteration)

```
.incl %edx
.cmpl %ecx,%edx
.jl .L11
.L13:
```

Load cnt
Update cnt
Store cnt

Tail (loop iteration)
Concurrent Execution

Any sequentially consistent interleaving is possible

- $X_i$ denotes that thread $i$ executes instruction $X$
- $\%eax_i$ is the content 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>$U_1$</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>

$H$ – head
$L$ – Load
$U$ – Update
$S$ – Store
$T$ – tail
Concurrent Execution (cont)

- Race condition: two threads increment the counter, but the result is 1 instead of 2

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr&lt;sub&gt;i&lt;/sub&gt;</th>
<th>%eax&lt;sub&gt;1&lt;/sub&gt;</th>
<th>%eax&lt;sub&gt;2&lt;/sub&gt;</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H&lt;sub&gt;1&lt;/sub&gt;</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Thread 1 critical section
Thread 2 critical section

H – head
L – Load
U – Update
S – Store
T – tail

Oops!
Concurrent Execution (cont)

- How about this ordering?

<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></td>
</tr>
<tr>
<td>2</td>
<td>H$_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L$_2$</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U$_2$</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S$_2$</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>U$_1$</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S$_1$</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T$_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T$_2$</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- How do we fix it?!
Enforcing Mutual Exclusion

- **Answer:** We must *synchronize* the execution of the threads so that critical regions are run sequentially.
  - i.e., need to guarantee *mutually exclusive access* to critical regions

- **Classic solution:**
  - Semaphores (Dijkstra)

- **Other approaches (out of our scope)**
  - Pthread Mutexes and condition variables
  - Monitors (Java)
Semaphores

- **Semaphore**: non-negative global integer synchronization variable

- Manipulated by \( P \) and \( V \) operations:
  - \( P(s) \): [ while (\( s == 0 \)) \ wait(); \( s--; \) ]
    - Dutch for "Proberen" (test)
    - Used to gain access to a shared resource
    - AKA: down, \texttt{wait}
  - \( V(s) \): [ \( s++ \); ]
    - Dutch for "Verhogen" (increment)
    - Used to release access of a shared resource
    - AKA: up, \texttt{post}

[...] – indicates an ‘atomic’ operation performed by the OS
- Guarantees these are run as if it was one instructions
- Only one \( P \) or \( V \) operation at a time can modify \( s \).
Using Semaphores for Mutual Exclusion

- **Basic idea:**
  - Associate a unique semaphore *mutex* with each shared resource (e.g. a variable)
  - Surround corresponding critical sections with \( P(mutex)/\text{wait()} \) and \( V(mutex)/\text{post()} \) operations.

- **Terminology:**
  - *Binary semaphore*: semaphore whose value is always 0 or 1
  - *Mutex*: binary semaphore used for mutual exclusion
    - \( P()/\text{wait()} \) operation: “locking” the mutex
    - \( V()/\text{post()} \) operation: “unlocking” or “releasing” the mutex
    - “**Holding**” a mutex/resource: locked and not yet unlocked.
C Semaphore Operations

Pthreads functions:

```c
#include <semaphore.h>

/* creating a semaphore */
int sem_init(sem_t *sem, 0, unsigned int val);} /* s = val */

/* using a semaphore */
int sem_wait(sem_t *sem); /* P(s): try to claim a resource */
int sem_post(sem_t *sem); /* V(s): release a resource */
```

shared between threads
(as opposed to processes)

Number of resources available
(how many threads can access
the data at any given time)
Improper Synchronization: badcnt.c

```c
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
    int niters = atoi(argv[1]);
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread, &niters);
    pthread_create(&tid2, NULL, thread, &niters);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    /* local data */
    int i, niters = *((int *)vargp);
    cnt++;

    return NULL;
}
```

How can we fix this using semaphores?
Proper Synchronization: goodcnt.c

- Define and initialize a mutex for the shared variable `cnt`:

```c
volatile int cnt = 0;  /* Counter */
sem_t mutex;           /* Semaphore that protects cnt */
sem_init(&mutex, 0, 1); /* mutex=1 (only 1 access at a time) */
```

- Surround critical section with `P` and `V`:

```c
for (i = 0; i < niters; i++) {
    sem_wait(&mutex);  /* P() */
    cnt++;
    sem_post(&mutex);  /* V() */
}
```

Note: much slower than badcnt.c
Sample Problems

- Readers-writers problem
- Producer-consumer problem
Readers-Writers Problem

- Problem statement:
  - *Reader* threads only read the object
  - *Writer* threads modify the object
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object

- Examples:
  - Online airline reservation system
  - Multithreaded caching Web proxy
  - Banking software
Variants of Readers-Writers

- **First readers-writers problem** (favors readers)
  - No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - A reader that arrives after a waiting writer gets priority over the writer.

- **Second readers-writers problem** (favors writers)
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting.

- **Starvation** (where a thread waits indefinitely) is possible in both cases.
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both set to 1 */

void reader(void)
{
    while (1) {
        /* am I the first reader in? */
        /* if so, try to lock mutex */
        /* to exclude writer */

        /* do some reading */

        /* am I the last reader out? */
        /* if so, unlock mutex */
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        /* try to lock mutex */
        /* to exclude readers */

        /* do some writing */

        /* unlock mutex */
    }
}
```
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;  /* Initially 0 */
sem_t mutex, w;  /* Both initially 1 */

void reader(void)
{
    while (1) {
        sem_wait(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            sem_wait(&w);
        sem_post(&mutex);

        /* Reading happens here */

        sem_wait(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            sem_post(&w);
        sem_post(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        sem_wait(&w);

        /* Writing here */
        sem_post(&w);
    }
}
```

rw1.c
Sample Problems

- Readers-writers problem
- Producer-consumer problem
Producer-Consumer Problem

- **Common synchronization pattern:**
  - Producer waits for empty *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 1-element Buffer

```c
#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-Consumer on 1-element Buffer

Initially: empty==1, full==0

### Producer Thread

```c
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 */
        sem_wait(&shared.empty);
        shared.buf = item;
        sem_post(&shared.full);
    }
    return NULL;
}
```

### Consumer Thread

```c
void *consumer(void *arg) {
    int i, item;

    for (i = 0; i < NITERS; i++) {
        /* Read item from buf */
        sem_wait(&shared.full);
        item = shared.buf;
        sem_post(&shared.empty);

        /* Consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```
Producer-Consumer on an $n$-element Buffer

- Requires a mutex and two counting semaphores:
  - **mutex**: enforces mutually exclusive access to the buffer
  - **slots**: counts the available slots in the buffer
  - **items**: counts the available items in the buffer

- Implemented using a shared buffer
**Producer-Consumer w/ Shared Buffer**

- **Producer-consumer pattern:**
  - Producer inserts item in buffer (waits if buffer is full)
  - Consumer removes item from buffer (waits if buffer is empty)

- **Examples:**
  - Network server
    - Producer threads read from sockets and put client request on buffer
    - Consumer threads process clients’ requests from buffer
Topics

- Sharing data
- Race conditions
- Mutual exclusion and semaphores
- Sample problems
- Deadlocks
Problem: Deadlocks

- Def: A process is *deadlocked* if it is waiting for a condition that will never be occur.

- Typical Scenario
  - Processes 1 and Process 2 both need the same two resources (A and B)
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!
Deadlocking With 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, worker, (void*) 0); /* Lock 0-1 */
    pthread_create(&tid[1], NULL, worker, (void*) 1); /* Lock 1-0 */
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    exit(0);
}

void *worker(void *x)
{
    int id = (int) x;
    sem_wait(&mutex[id]); sem_wait(&mutex[1-id]);
    // Do something...
    sem_post(&mutex[id]); sem_post(&mutex[1-id]);
    return NULL;
}
```

Tid[0]:
P(s_0);
P(s_1);
...
V(s_0);
V(s_1);

Tid[1]:
P(s_1);
P(s_0);
...
V(s_1);
V(s_0);
Avoiding Deadlock with 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, worker, (void*) 0); /* Lock 0-1 */
    pthread_create(&tid[1], NULL, worker, (void*) 0); /* Lock 0-1 */
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    exit(0);
}

void *worker(void *x)
{
    int id = (int) x;
    sem_wait(&mutex[id]); sem_wait(&mutex[1-id]);
    // Do something...
    sem_post(&mutex[id]); sem_post(&mutex[1-id]);
    return NULL;
}
```

Acquire shared resources in same order
Release resources in same reverse order
Threads Summary

- Threads are were growing in popularity

- Pros:
  - Somewhat cheaper than processes
  - Easy to share data between threads

- Cons:
  - Easy to share data between threads
    - Easy to introduce subtle synchronization errors

- Shares variables must be protected to ensure mutually exclusive access
  - Semaphores provide one solution