Today

- Threads vs Processes
- Thread Features and types
- Hello world with threads
- Comparison and applications
- Concurrent Programming – Intro
- **Sharing among threads**

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,
  - 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 truly separate and protected, but…
  - Any thread can read and write the stack of any other thread

The mismatch between the conceptual and operation model is a source of confusion and errors

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.**

Synchronizing Threads

- **Shared variables are handy...**

- **...but introduce the possibility of nasty *synchronization* errors.**

badcnt.c: Improper Synchronization

```c
/* Global shared variable */
long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
   long niters;
   pthread_t tid1, tid2;

   niters = atoi(argv[1]);
   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=%ld\n", cnt);
   else
      printf("OK cnt=%ld\n", cnt);
   exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
   long i, niters =
            *((long *)vargp);

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

   return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

`cnt` should equal 20,000.

What went wrong?

Enforcing Mutual Exclusion

- **Need to guarantee *mutually exclusive access* for each critical section.**

- **Classic solution:**
  - Semaphores

- **Other approaches (out of our scope)**
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

Semaphores

- ***Semaphore:*** **non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**
- **P(s)**
  - If *s* is nonzero, then decrement *s* by 1 and return immediately.
    - Test and decrement operations occur atomically (indivisibly)
  - If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s):***
  - Increment *s* by 1.
    - Increment operation occurs atomically
  - If there are any threads blocked in a P operation waiting for *s* to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing *s*.

- **Semaphore invariant: *(s >= 0)***

Semaphore

- Assume Semaphore *S* – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `post()`
    - Originally called `P()` and `V()`
- Definition of the `wait()` operation

```
wait(S) {
     while (S <= 0)
         ; // busy wait
     S--;
}
```

- Definition of the `post()` operation

```
post(S) {
     S++;
}
```

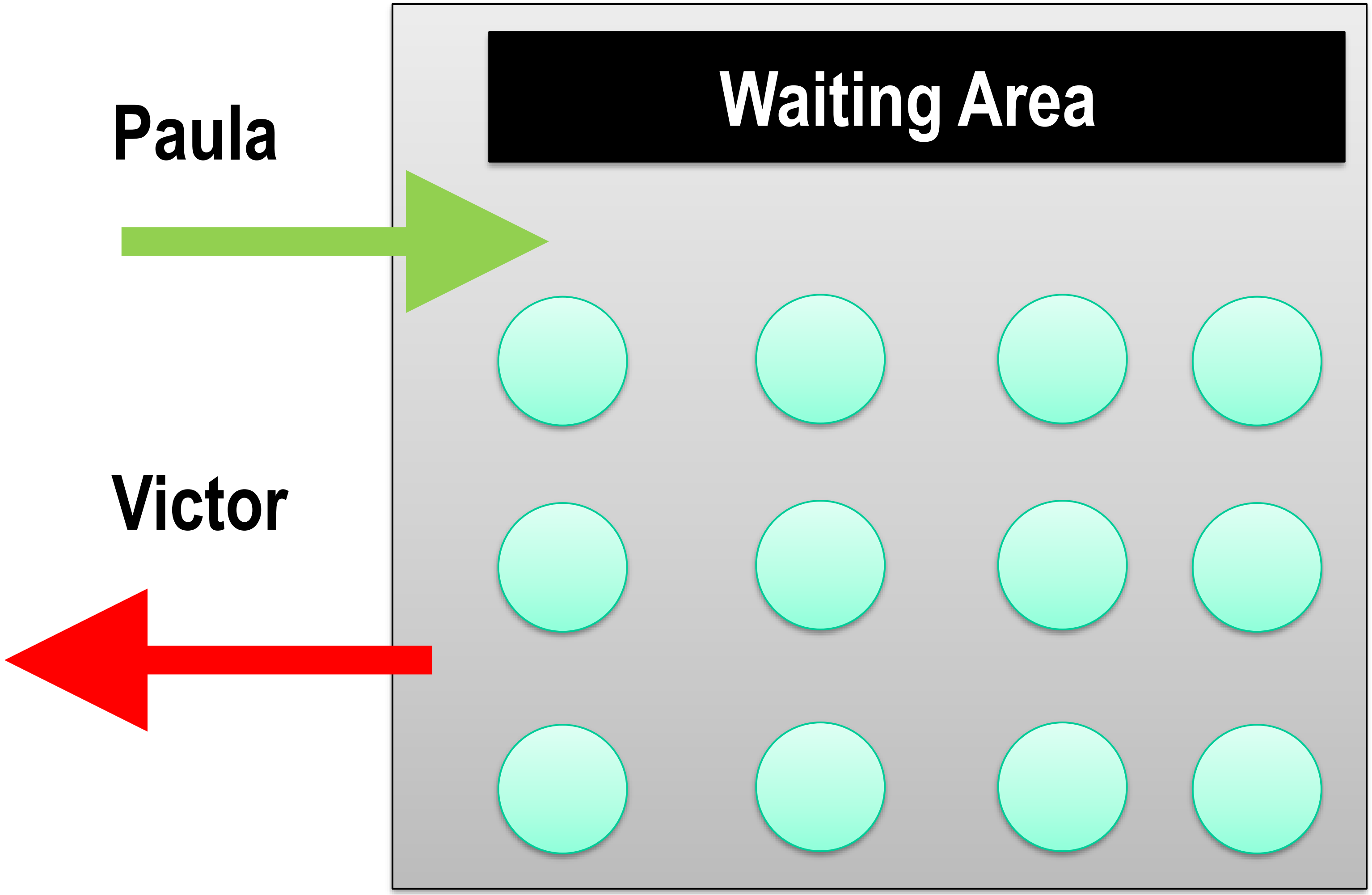
Inside csapp.c

```
void P(sem_t *sem)
{
    if (sem_wait(sem) < 0)
    unix_error("P error");
}

void V(sem_t *sem)
{
    if (sem_post(sem) < 0)
    unix_error("V error");
}
```

An analogy

- **Paula and Victor work in a restaurant:**
- **Paula handles customer arrivals:**
  - Prevents people from entering the restaurant when all tables are busy.
- **Victor handles departures**
  - Notifies people waiting for a table when one becomes available
- **The semaphore represents the number of available tables**
  - Initialized with the ***total number of tables*** in restaurant


Paula
Victor
Waiting Area

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

CS:APP wrapper functions:

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

Using Semaphores for Mutual Exclusion

- **Basic idea:**
  - Associate a unique semaphore with each shared variable (or related set of shared variables).
  - Surround corresponding critical sections with *P(mutex)* and *V(mutex)* operations.

- **Terminology:**
  - *Binary semaphore*: semaphore whose value is always 0 or 1
  - *Mutex:* binary semaphore used for mutual exclusion
    - P operation: "locking" the mutex
    - V operation: "unlocking" or "releasing" the mutex
    - *"Holding"* a mutex: locked and not yet unlocked.
  - *Counting semaphore*: used as a counter for set of available resources.

goodcnt.c: Proper Synchronization

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

```
long cnt = 0;  /* Counter */
sem_t mutex;          /* Semaphore that protects cnt , global*/

sem_init(&mutex, 0, 1); /* mutex = 1 , within main function*/
```

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

```
for (i = 0; i < niters; i++) {
   P(&mutex);
   cnt++;
   V(&mutex);
}
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Warning: It's orders of magnitude slower than `badcnt.c`.

Review: Using semaphores to protect shared resources via mutual exclusion

- **Basic idea:**
  - Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables)
  - Surround each access to the shared variable(s) with *P(mutex)* and *V(mutex)* operations

```
sem_init(mutex,0,1);

 P(&mutex)
 cnt++
 V(&mutex)
```

Producer-Consumer on an *n*-element Buffer

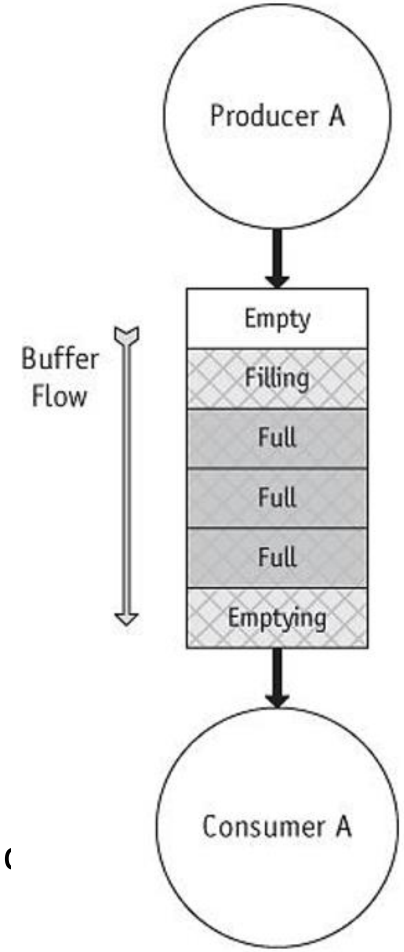
- **Common synchronization pattern:**
  - Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
  - Consumer waits for ***item***, removes it from buffer, and notifies producer

- **Requires a mutex and two counting semaphores:**
  - `mutex (binary semaphore)`: enforces mutually exclusive access to the the buffer
  - `slots (counting semaphore)`: counts the available slots in the buffer
  - `Items (counting semaphore)`: counts the available items in the buffer

- **Text book has an implementation using a shared buffer package called `sbuf`.**
- **We will just illustrate it here**

Producer-Consumer Problem

- **Also known as Bounded buffer problem**
  - Number of full positions
    - Semaphore `items` initialized to the value 0
  - Number of empty positions
    - Semaphore `slots` initialized to the value n
- Mutex
  - Third semaphore: ensures mutual exclusion
    - Semaphore `mutex` initialized to the value 1



- **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 correspond events in buffer
    - Consumer retrieves events from buffer and paints the display

Bounded Buffer Problem Pseudocode(Cont.)

- ```
  producer() {
     struct x item;
     for(;;) {
       produce(&item);
       P(&slots);
       P(&mutex);
       put(item);
       V(&mutex);
       V(&items);
     } // for
  } // producer
  ```

  ```
  consumer() {
     struct x item;
     for(;;) {
       P(&items);
       P(&mutex);
       take(item);
       V(&mutex);
       V(&slots);
       eat(item);
     } // for
  } // consumer
  ```
  ```

Crucial concept: Thread Safety

- **Functions called from a thread must be *thread-safe***

- ***Def:* A function is *thread-safe* iff it will always produce correct results when called repeatedly from multiple concurrent threads**

- **Potentially thread-unsafe functions:**
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer
  - Class 4: Functions that call thread-unsafe functions ☺

- **Fix: Use *P* and *V* semaphore operations**

Another worry: Deadlock

- **Def: A process is *deadlocked* iff it is waiting for a condition that will never be true**

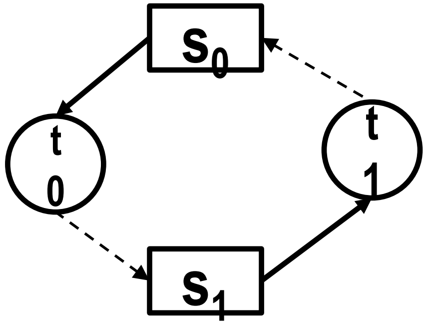
- **Typical Scenario**
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!

Deadlocking With Semaphores

```
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;
}
```

Tid[0]:	Tid[1]:
$P(s_0)$;	$P(s_1)$;
$P(s_1)$;	$P(s_0)$;
cnt++;	cnt++;
$V(s_0)$;	$V(s_1)$;
$V(s_1)$;	$V(s_0)$;



At some point t_0 might be waiting for s_1 while t_1 might be waiting for s_0

Avoiding Deadlock

Acquire shared resources in same order

```
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]:	Tid[1]:
P(s0);	P(s0);
P(s1);	P(s1);
cnt++;	cnt++;
V(s0);	V(s1);
V(s1);	V(s0);

Total ordering is a method to prevent deadlocks.

Laundromat

- A Laundromat has 30 washing machines and 15 dryers. Assuming that all customers use one washing machine to wash their clothes then one dryer to dry them, add the necessary semaphore calls to the following program segment.

```
semaphore _____________ = ______;
semaphore _____________ = ______;
customer (int who) {
       _________________________;
       wash_clothes();
       __________________________;
       __________________________;
       dry_clothes();
       __________________________; }
```

Laundromat

- A Laundromat has 30 washing machines and 15 dryers. Assuming that all customers use one washing machine to wash their clothes then one dryer to dry them, add the necessary semaphore calls to the following program segment.

```
semaphore ____washer __= __30__;
semaphore ____dryer____ = __15__;
customer (int who) {
        ________________________;
        wash_clothes();
        _________________________;
        _________________________;
        dry_clothes();
        _________________________; }
```

Laundromat

- A Laundromat has 30 washing machines and 15 dryers. Assuming that all customers use one washing machine to wash their clothes then one dryer to dry them, add the necessary semaphore calls to the following program segment.

```
semaphore ____washer __= __30__;
semaphore ____dryer____ = __15__;
customer (int who) {
        P(& washer)____________;
        wash_clothes();
        ________________________;
        ________________________;
        dry_clothes();
        ________________________; }
```

Laundromat

- A Laundromat has 30 washing machines and 15 dryers. Assuming that all customers use one washing machine to wash their clothes then one dryer to dry them, add the necessary semaphore calls to the following program segment.

```
semaphore ____washer __= __30__;
semaphore ____dryer____ = __15__;
customer (int who) {
        P(& washer)____________;
        wash_clothes();
        V(& washer)____________;
        _______________________;
        dry_clothes();
        _______________________; }
```

Laundromat

- A Laundromat has 30 washing machines and 15 dryers. Assuming that all customers use one washing machine to wash their clothes then one dryer to dry them, add the necessary semaphore calls to the following program segment.

```
semaphore ____washer __= __30__;
semaphore ____dryer____ = __15__;
customer (int who) {
        P(& washer)_____________;
        wash_clothes();
        V(& washer)_____________;
        P(& dryer)  ______________;
        dry_clothes();
        V(& dryer)______________; }
```

The ice-cream parlor

- An ice-cream parlor has two employees selling ice cream and six seats for its customers. Each employee can only serve one customer at a time and each seat can only accommodate one customer at a time. Add the required semaphores to the following program skeleton to guarantee that customers will never have to wait for a chair with a melting ice-cream in their hand.

```
semaphore _______________ = ______;
semaphore _______________ = ______;
customer (int who) {
    ______________________________
    order_ice_cream();
    ______________________________
    eat_it();
    ______________________________
} // customer
```

Sketching a solution

- **Two resources are shared by all customers**
  - Six seats
  - Two employees

- **Questions to ask are**
  - When should we request a resource?
  - In which order? (***very important***)
  - When should we release it?

Solution

```
semaphore _seats___________ = __6___;
semaphore _employees______ = _2___;
customer (int who) {
        ________________________________
        order_ice_cream();
        ________________________________
        eat_it();
        ________________________________
} // customer
```

Solution (cont'd)

"customers will never have to wait for a chair with a melting ice-cream in their hand.

```
semaphore _seats___________ = __6___;
semaphore _employees______  = __2___;
customer (int who) {
    P(&seats); P(&employees);______
    order_ice_cream();
    ___________________________________;
    eat_it();
    ___________________________________;
// customer
```

Get seat fir

Solution (cont'd)

```
semaphore _seats___________ = __6___;
semaphore _employees_______ = __2___;
customer (int who) {
        P(&seats); P(&employees);_________
        order_ice_cream();
        V(&employees);___________________
        eat_it();
        ________________________
} // customer
```

What is missing?

Solution (cont'd)

```
semaphore _seats___________ = __6___;
semaphore _employees______ = __2___;
customer (int who) {
        P(&seats); P(&employees);_________
        order_ice_cream();
        V(&employees);___________________
        eat_it();
        V(&seat);_________________________
} // customer
```

The pizza oven

A pizza oven can contain nine pizzas but the oven narrow opening allows only one cook at a time to either put a pizza in the oven or to take one out. Given that there will be more than one cook preparing pizzas at any given time, complete the missing lines in the following C procedure.

```
semaphore oven   = ______;
semaphore access = _______;
make_pizza(int size, int toppings) {
        prepare_pizza(size, toppings);
        ________________________________
        put_into_oven();
        ________________________________
        wait_until_done();
        ________________________________
        take_from_oven();
        ________________________________
} // make_pizzac
```

Sketching a solution

- **The two resources are already identified**
  - The oven
  - Access to the oven (mutex)
- **We ask the usual questions**
  - And take care of avoiding
    mutex-induced deadlocks

The pizza oven

A pizza oven can contain nine pizzas but the oven narrow opening allows only one cook at a time to either put a pizza in the oven or to take one out. Given that there will be more than one cook preparing pizzas at any given time, complete the missing lines in the following C procedure.

```
semaphore oven   = ___9__;
semaphore access = __1___;  // the mutex
make_pizza(int size, int toppings) {
        prepare_pizza(size, toppings);
        P(&oven); P(&access);____________________
        put_into_oven();
        __________________________________________
        wait_until_done();
        __________________________________________
        take_from_oven();
        __________________________________________
} // make_pizza
```

Order matters

The pizza oven

A pizza oven can contain nine pizzas but the oven narrow opening allows only one cook at a time to either put a pizza in the oven or to take one out. Given that there will be more than one cook preparing pizzas at any given time, complete the missing lines in the following C procedure.

```
semaphore oven   = ___9__;
semaphore access = __ 1__;  // the mutex
make_pizza(int size, int toppings) {
        prepare_pizza(size, toppings);
        P(&oven); P(&access); ____________________
        put_into_oven();
        V(&access);______________________________
        wait_until_done();
        P(&access);______________________________
        take_from_oven();
        V(&oven); V(&access);  // IN ANY ORDER!____
} // make_pizza
```

The pizza oven

A pizza oven can contain nine pizzas but the oven narrow opening allows only one cook at a time to either put a pizza in the oven or to take one out. Given that there will be more than one cook preparing pizzas at any given time, complete the missing lines in the following C procedure.

```
semaphore oven   = ___9__;
semaphore access = __ 1__;  // the mutex
make_pizza(int size, int toppings) {
        prepare_pizza(size, toppings);
        P(&oven); P(&access); ______________________
        put_into_oven();
        V(&access);________________________________
        wait_until_done();
        P(&access);________________________________
        take_from_oven();
        V(&oven); V(&access);  // IN ANY ORDER!____
} // make_pizza
```

Parallelization with threads

- Semaphores enable synchronization

- Efficient for protection of critical section, in many cases
- May be inefficient for parallelization jobs, if job requires too frequent critical section accesses.

- For parallelization, better prevent critical section.

Parallelization Example: Parallel Summation

- **Sum numbers *0, …, n-1***
  - Should add up to *((n-1)*n)/2*
- **Partition values *1, …, n-1* into *t* ranges**
  - *n/t* values in each range
  - Each of *t* threads processes 1 range
  - For simplicity, assume *n* is a multiple of *t*

- **Let's consider different ways that multiple threads might work on their assigned ranges in parallel**

First attempt: psum-mutex

- **Simplest approach: Threads sum into a global variable protected by a semaphore mutex.**

```
void *sum_mutex(void *vargp); /* Thread routine */

/* Global shared variables */
long gsum = 0;              /* Global sum */
long nelems_per_thread;     /* Number of elements to sum */
sem_t mutex;                /* Mutex to protect global sum */

int main(int argc, char **argv)
{
    long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
    pthread_t tid[MAXTHREADS];

     /* Get input arguments */
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems); /* nelems will be 2to the power log_nelems */
    nelems_per_thread = nelems / nthreads;
    sem_init(&mutex, 0, 1);
```

psum-mutex.c

psum-mutex (cont)

- **Simplest approach: Threads sum into a global variable protected by a semaphore mutex.**

```
    /* Create peer threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        Pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    /* Check final answer */
    if (gsum != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", gsum);

    exit(0);
}
```

psum-mutex.c

psum-mutex Thread Routine

- **Simplest approach: Threads sum into a global variable protected by a semaphore mutex.**

```c
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        P(&mutex);
        gsum += i;
        V(&mutex);
    }
    return NULL;
}
```

psum-mutex.c

`psum-mutex` Performance

- **Sample run with 8 cores, $n=2^{31}$**

Threads (Cores)	1 (1)
psum-mutex (secs)	51

Is it going to run faster with more threads?

You can run as: time ./psum-mutex 2 31

For 2 threads and , $n=2^{31}$

`psum-mutex` Performance

- **Sample run with 8 cores, $n=2^{31}$**

Threads (Cores)	1 (1)	2 (2)	4 (4)	8 (8)	16 (8)
psum-mutex (secs)	51	456	790	536	681

You can run as: time ./psum-mutex 2 31
For 2 threads and , $n=2^{31}$

- **Nasty surprise:**
  - Single thread is very slow
  - Gets slower as we use more cores

Next Attempt: `psum-array`

- Peer thread `i` sums into global array element `psum[i]`
- Main waits for theads to finish, then sums elements of `psum`
- Eliminates need for mutex synchronization

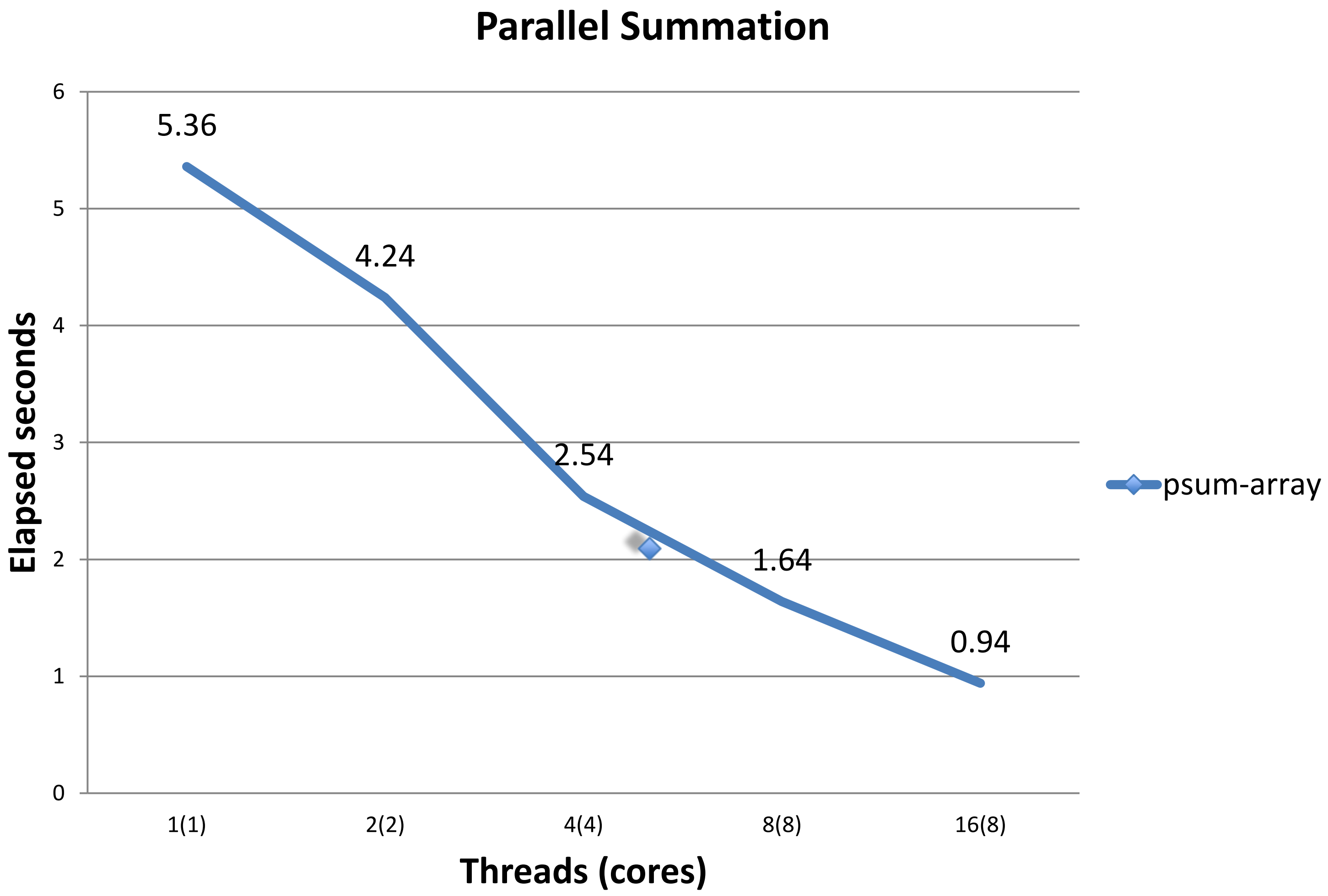
```
/* Thread routine for psum-array.c */
void *sum_array(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        psum[myid] += i;
    }
    return NULL;
}
```

psum-array.c

`psum-array` Performance

- Orders of magnitude faster than `psum-mutex`



Remember psum-mutex took 51 secs to complete with 1 thread and 456 seconds to complete with 2 threads

Next Attempt: `psum-local`

- Reduce memory references by having peer thread i sum into a local variable (register)

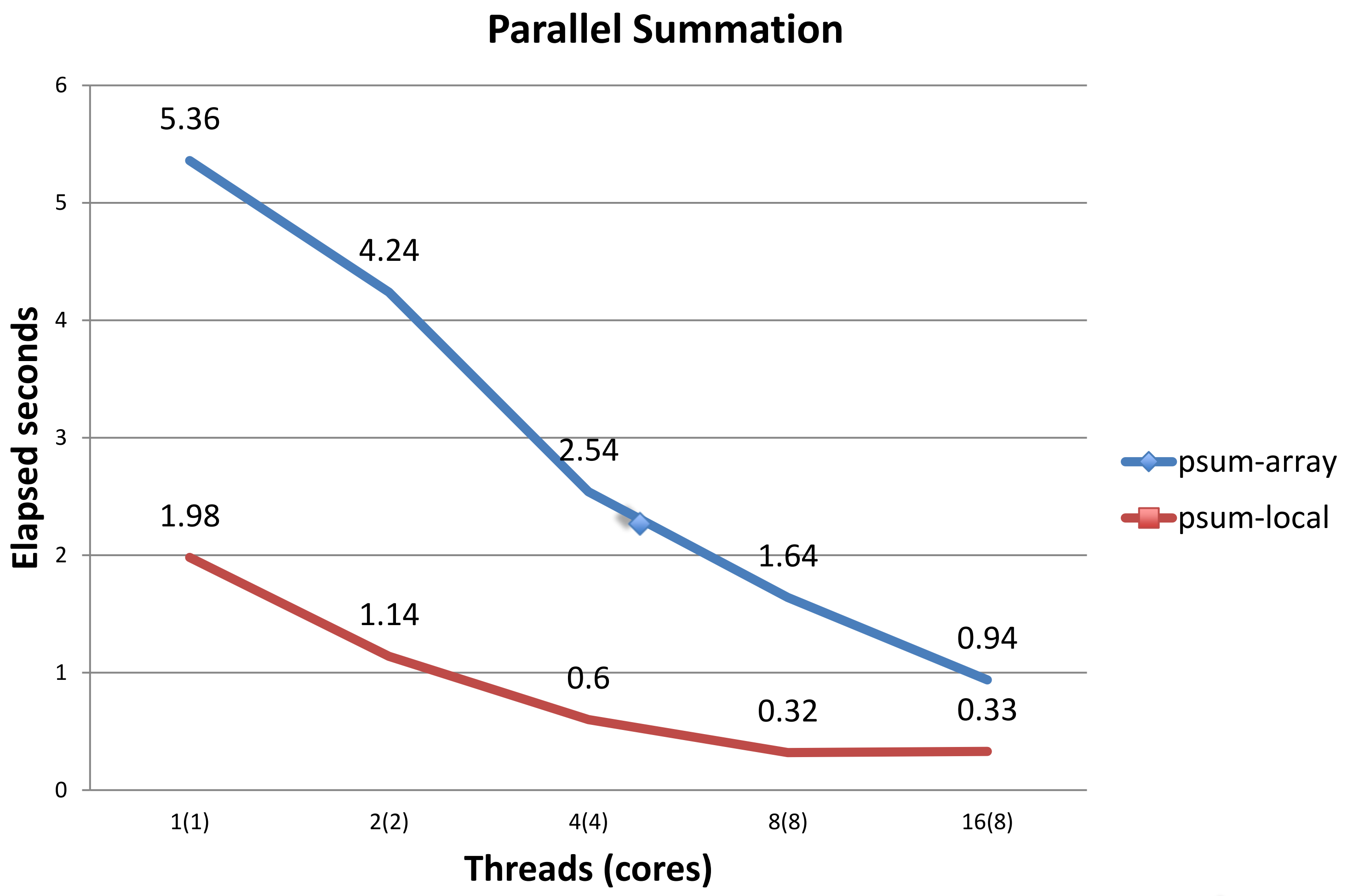
```
/* Thread routine for psum-local.c */
void *sum_local(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;
    return NULL;
}
```

psum-local.c

psum-local Performance

- Significantly faster than psum-array



Characterizing Parallel Program Performance

- ***p*** **processor cores,** T_k **is the running time using** ***k*** **cores**

- ***Def. Speedup:*** $S_p = T_1 / T_p$
  - S_p is *relative speedup* if T_1 is running time of parallel version of the code running on 1 core.
    T_p is running time with parallelization

- ***Def. Efficiency:*** $E_p = S_p / p = T_1 / (pT_p)$
  - Reported as a percentage in the range [0, 100].
  - Measures the overhead due to parallelization

Performance of `psum-local`

Threads (t)	1	2	4	8	16
Cores (p)	1	2	4	8	8
Running time (T_p)	1.98	1.14	0.60	0.32	0.33
Speedup (S_p)	1	1.74	3.30	6.19	6.00
Efficiency (E_p)	100%	87%	82%	77%	75%

- **Efficiencies OK, not great**
- **Our example is easily parallelizable**
- **Real codes are often much harder to parallelize**