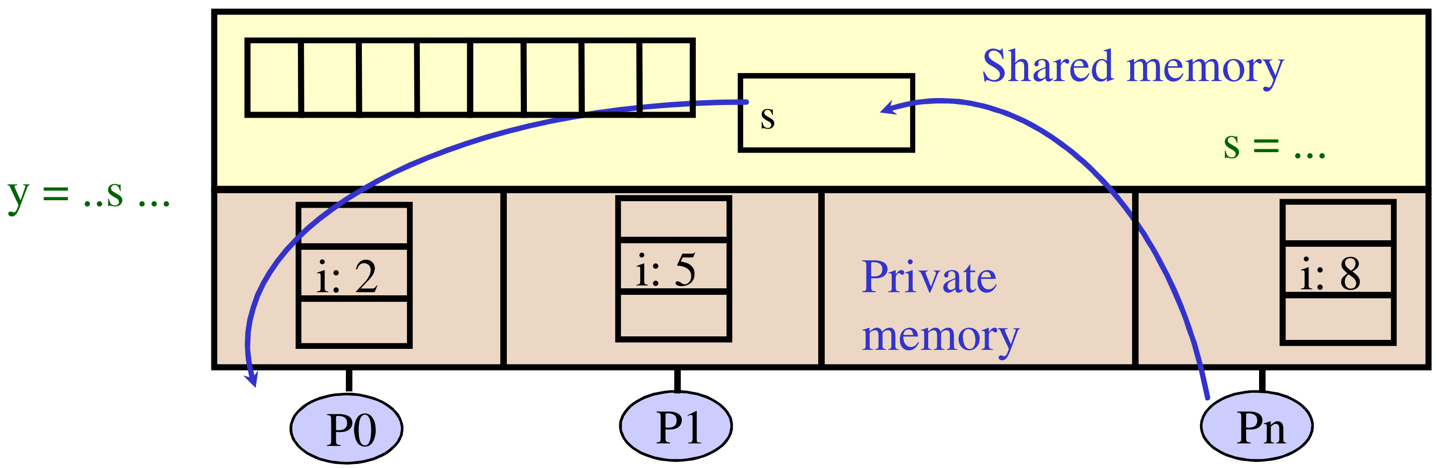
Shared Memory Programming - Pthreads

Outline

- Shared Memory Hardware
- Memory consistency: the dark side of shared memory
  - Hardware review and a few more details
  - What this means to shared memory programmers
- Thread creation
- Thread termination
- Thread join
- Synchronization primitives
  - Semaphore
  - Mutex locks
  - Conditional variables
  - Barrier
  - Busy waiting

Programming Model 1: Shared Memory

- Program is a collection of threads of control
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

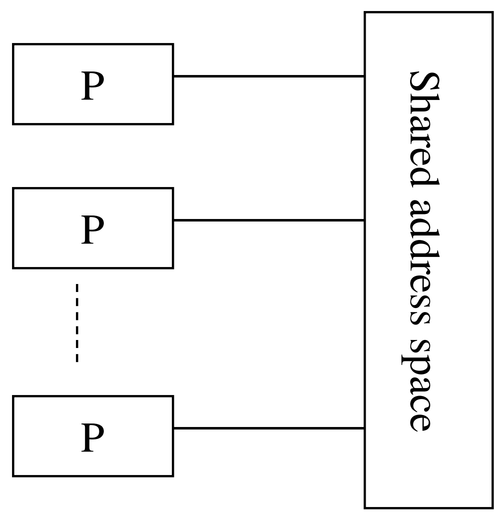


Shared Memory Programming Models

- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight
- Directive-based model, OpenMP standard for application level programming
  - Support for scientific programming on shared memory
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C “ilk”
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language

Thread

- A single, sequential stream of control in a program
- Logical machine model
  - Flat global memory shared among all threads
  - Local stack of frames for each thread's active procedures



Why Threads?

- Portable, widely-available programming model
  - Use on both serial and parallel systems
- Useful for hiding latency
  - E.g., latency due to IO, communication
- Useful for scheduling and load balancing
  - Especially for dynamic concurrency
- Relatively easy to program
  - Significantly easier than message-passing

POSIX Thread (Pthreads)

- IEEE had a POSIX 1003 group that defined an interface to multithreaded programming
  - This is called Pthreads, and is similar to Solaris Threads from Sun
  - Not just for parallel programming, but for general multithreaded programming.
  - Provides primitives for thread management and synchronization
- Standard threads API supported by most vendors
- Concepts behind Pthreads are broadly applicable
  - Largely independent of the API
  - Useful for programming with other thread API like Solaris threads and Java threads
- Threads are peers, no parent/child relationship

Thread Creation

```
#include <pthread.h>
int pthread_create
   (pthread_t *new_id,
   const pthread_attr_t *attr,
   void *(*func) (void *),
   void *arg)
```

- new_id: the thread id or handle (used to halt, etc.)
- attr: various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attribute: minimum stack size
- func: function to be run in parallel
- arg: an argument can be passed to func when it starts

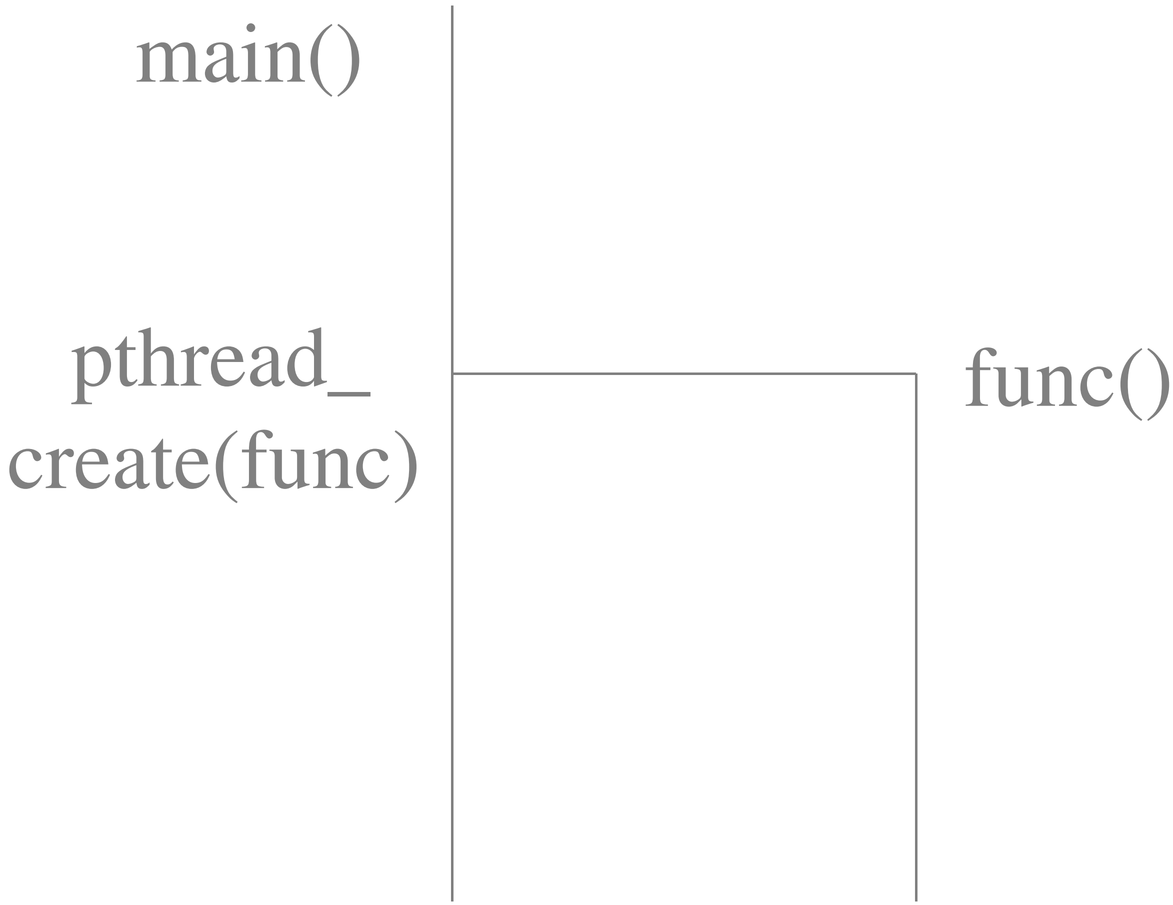
- The function creates and starts a new thread

Example of Thread Creation

```
void *func(void *arg) {
    int         *i=arg;
    …..
}

void main()
{
    int X;      pthread_t         id;
    ….
    pthread_create(&id, NULL, func, &X);
    …
}
```

Example of Thread Creation (cont.)



Pthread Termination

*void pthread_exit(void *status)*

- Terminates the currently running thread.

Thread Joining

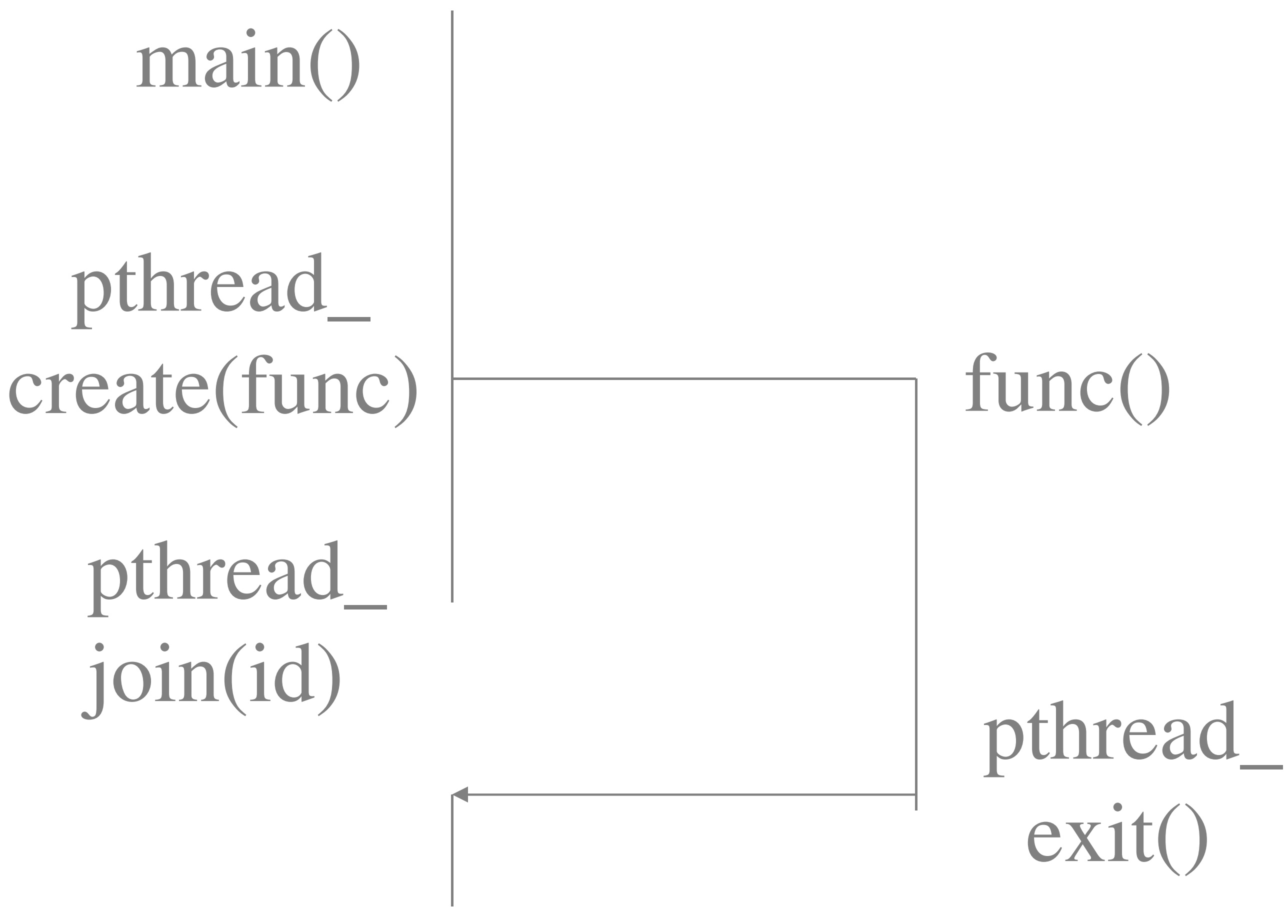
int pthread_join(
pthread_t new_id,
*void **status)*

- Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit()

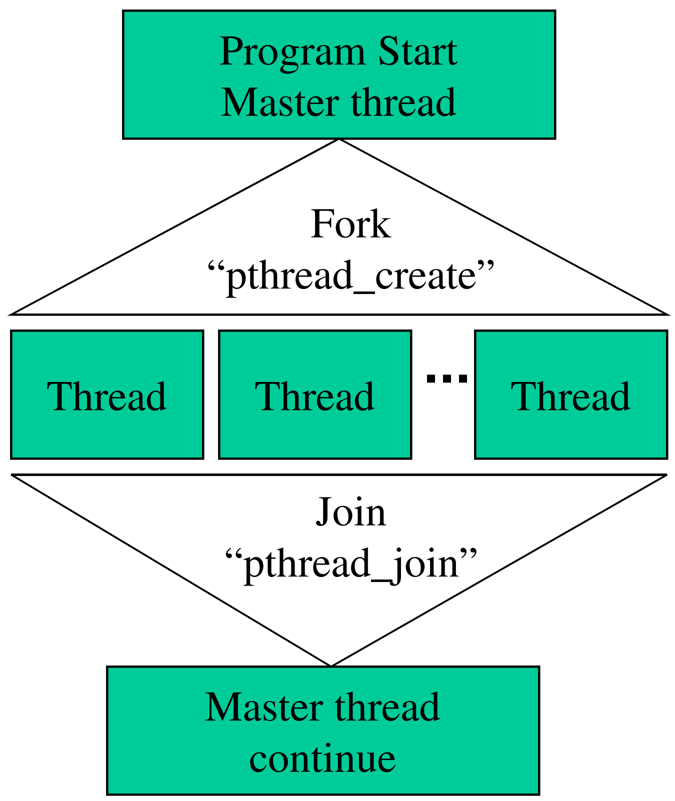
Thread Joining Example

```
void *func(void *) { ….. }
pthread_t     id;  int X;
pthread_create(&id, NULL, func, &X);
…..
pthread_join(id, NULL);
…..
```

Example of Thread Join (contd.)



Pthreads Programming Model



Pthread ID

pthread_t pthread_self(void);

- To determine the thread ID of the calling thread

Thread Attributes

- Detach state
  - PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE
  - reclaim storage at termination (detached) or retain (joinable)
- Scheduling policy
  - SCHED_OTHER: standard round robin (priority must be 0)
  - SCHED_FIFO, SCHED_RR: real time policies
    - FIFO: re-enter priority list at head; RR: re-enter priority list at tail
- Scheduling parameters
  - only priority
- Inherit scheduling policy
  - PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED
- Thread scheduling scope
  - PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS
- Stack size

Simple Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Matrix Multiply

```
for( i=0; i<n; i++ )
   for( j=0; j<n; j++ ) {
            c[i][j] = 0.0;
            for( k=0; k<n; k++ )
                        c[i][j] += a[i][k]*b[k][j];
   }
```

Parallel Matrix Multiply

- All i- or j-iterations can be run in parallel.
- P threads: n/p rows to each thread.
- Corresponds to partitioning i-loop.

Parallel Matrix Multiplication

How to Program with PTHREADS

- To program a parallel application with PTHREADS, need to add this statement to the source file:

    #include <pthread.h>

- Compile a C program using:

    % cc –lpthread input.c

Data Race in Pthreads Program

```
static int s = 0;
```

```
Thread 1

   for i = 0, n/2-1
        s = s + f(A[i])
```

```
Thread 2

   for i = n/2, n-1
        s = s + f(A[i])
```

- Problem is a race condition on variable ***s*** in the program
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Pthreads Synchronization

- Create/exit/join
  - provide some form of synchronization,
  - at a very coarse level,
  - requires thread creation/destruction.
- Need for finer-grain synchronization
  - mutex locks
  - condition variables
  - ….
- PTHREADS provides a variety of synchronization facilities for threads to cooperate in accessing shared resources

Semaphore

- Semaphore?
  - wait operation: ?
  - post operation: ?
- It is not as efficient as a mutex lock. They need not be acquired and released by the same thread, so they may be used in asynchronous event notification
- The header file *semaphore.h* contains definitions and operation prototypes for semaphores.

Semaphore (1 of 3)

*int sem_init(sem_t *sem, int pshared, unsigned value);*

- Initialize the semaphore descriptor

Semaphore (2 of 3)

*int sem_wait(sem_t *sem);*

- Lock a semaphore

Semaphore (3 of 3)

*int sem_post(sem_t *sem);*

- unlock a semaphore

Use of Semaphores

```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1
#include <stdio.h>

void *Producer(void *);
Void *Consumer(void *)

sem_t empty, full; /* global semaphore*/
int data; /* shared buffer */
int numIters;

int main(int argc, char *argv[]) {
    pthread_t pid,cid; /* thread and attributes */

    sem_init(&empty, SHARED, 1); /* sem empty=1 and full=0 */
    sem_init(&full, SHARED, 0);

    numIters=atoi(argv[1]);
    pthread_create(&pid,NULL,Producer,NULL);
    pthread_create(&cid,NULL,Consumer,NULL);
    pthread_join(pid, NULL);
    pthread_join(cid,NULL);
}
```

Use of Semaphores

```
/* deposit 1, 2, …, numIters into the data buffer */
void *Producer(void *arg) {

}

/* fetch numIters items from the buffer and sum them*/
void *Consumer(void *arg) {

}
```

Mutex Locks (1 of 4)

pthread_mutex_init(
*pthread_mutex_t * mutex,*
*const pthread_mutex_attr *attr);*

- Creates a new mutex lock.
- Attribute: normal, recursive, errorcheck

Mutex Types

- Normal
  - Thread deadlocks if tries to lock a mutex it already has locked
- Recursive
  - Single thread may lock a mutex as many times as it wants
    - Increments a count on the number of locks
  - Thread relinquishes lock when mutex count becomes zero
- Errorcheck
  - Report error when a thread tries to lock a mutex it already locked
  - Report error if a thread unlocks mutex locked by another

Mutex Locks (2 of 4)

pthread_mutex_destroy(
*pthread_mutex_t *mutex);*

- Destroys the mutex specified by mutex.

Mutex Locks (3 of 4)

pthread_mutex_lock(
*pthread_mutex_t *mutex)*

- Tries to acquire the lock specified by mutex.
- If mutex is already locked, then calling thread blocks until mutex is unlocked.

Mutex Locks (4 of 4)

pthread_mutex_unlock(
    *pthread_mutex_t *mutex);*

- If calling thread has mutex currently locked, this will unlock the mutex.
- If other threads are blocked waiting on this mutex, one will unblock and acquire mutex.
- Which one is determined by the scheduler.

Example of Use of Locks

```
pthread_mutex_t count_mutex;
pthread_mutex_init(&count_mutex, NULL);
int count;
increment_count() {





}
int get_count() {





}
```

Note on Mutex Locks

- To implement critical sections as needed
- Pthreads provides only exclusive locks.
  - Some other systems allow shared-read, exclusive-write locks
- Locks enforce serialization
  - Threads must execute critical sections one at a time
- Large critical sections can seriously degrade performance
- Reduce overhead by overlapping computation with waiting

  int pthread_mutex_trylock (pthread_mutex_t *mutex_lock)
  - Acquire lock if available
  - Return EBUSY if not available
  - Enables a thread to do something else if lock unavailable

Condition variables (1 of 5)

pthread_cond_init(
    *pthread_cond_t *cond,*
    *pthread_cond_attr *attr)*

- Creates a new condition variable cond.
- Attribute: ignore for now.

Condition Variables (2 of 5)

pthread_cond_destroy(
*pthread_cond_t *cond)*

- Destroys the condition variable cond.

Condition Variables (3 of 5)

pthread_cond_wait(
    *pthread_cond_t *cond,*
    *pthread_mutex_t *mutex)*

- Blocks the calling thread, waiting on cond.
- Unlocks the mutex.

Condition Variables (4 of 5)

pthread_cond_signal(
    *pthread_cond_t *cond)*

- Unblocks one thread waiting on cond.
- Which one is determined by scheduler.
- If no thread waiting, then signal is a no-op.

Condition Variables (5 of 5)

pthread_cond_broadcast(
    *pthread_cond_t *cond)*

- Unblocks all threads waiting on cond.
- If no thread waiting, then broadcast is a no-op.

Condition Variable for Synchronization

- Condition variable: associated with a predicate and a mutex
- Using a condition variable
  - Thread can block itself until a condition becomes true
    - Thread locks a mutex
    - Tests a predicate defined on a shared variable
    - If predicate is false, then wait on the condition variable
    - Waiting on condition variable unlocks associated mutex
  - When some thread makes a predicate true
    - That thread can signal the condition variable to either wake one waiting thread or wake all waiting threads
    - When thread releases the mutex, it is passed to first waiter

Condition Variables Usage

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned int count;
decrement_count(){








}
increment_count() {








}
```

Reality bites ...

- Create/exit/join is not so cheap.
- It would be more efficient if we could come up with a parallel program, in which
  - create/exit/join would happen rarely (once!),
  - cheaper synchronization were used.
- We need something that makes all threads wait, until all have arrived -- a barrier.

Implementing Barriers in Pthreads

```
pthread_mutex_t barrier;
pthread_cond_t go;
int numWorkers;
int numArrived =0;

void barrier()
{
   pthread_mutex_lock(&barrier);
   numArrived++;
   if (numArrived<numWorkers) {
            pthread_cond_wait(&go, &barrier);
   }
   else {
            pthread_cond_broadcast(&go);
            numArrived=0; /* be prepared for next barrier */
   }
   pthread_mutex_unlock(&barrier);
}
```

Other Primitives in Pthreads

- Set the attributes of a thread.
- Set the attributes of a mutex lock.
- Set scheduling parameters.

References

- We have only looked at a subset of Pthreads.
- For complete information, many good references exist:
  - https://computing.llnl.gov/tutorials/pthreads/
  - "*Threads Primer: A Guide to Multithreaded Programming*", Bil Lewis, Daniel J. Berg.
  - "*Pthreads Programming*", Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell, Jackie Farrell.
  - …..

Summary

- Thread creation
- Thread termination
- Thread join
- Synchronization primitives
  - Mutex locks
  - Conditional variables
  - Barrier
  - Busy wating
  - Semaphore