CIS 631
Parallel Processing

Lecture 9: Shared Memory Parallel Programming

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon

Acknowledgements

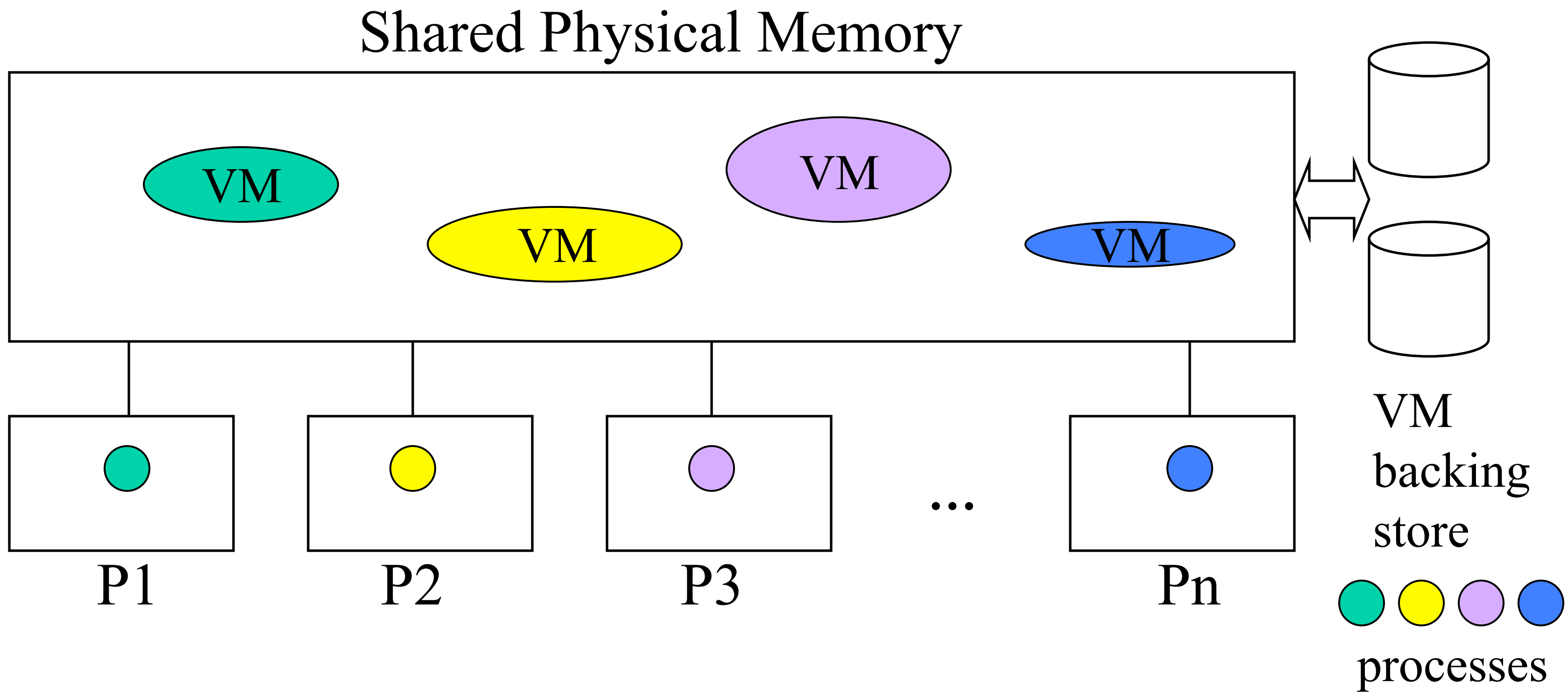
- Portions of the lectures slides were adopted from:
  - I. Foster, "Designing and Building Parallel Programs," 1995
  - Vijay Pai, COMP 422, "Parallel Programming," Rice University, 2002
  - John Mellor-Crummey, COMP 422, "Parallel Programming," Rice University, 2010
  - A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Introduction to Parallel Computing," 2003

Outline

- Shared memory parallelism and programming
- Process-based vs. thread-based programming
- Pthreads programming

Shared Memory Multiprocessors

- All processes share the same physical memory space
- Processes generally do not share virtual memory space



- How to program for parallel execution?

Shared Address Space Programming Taxonomy

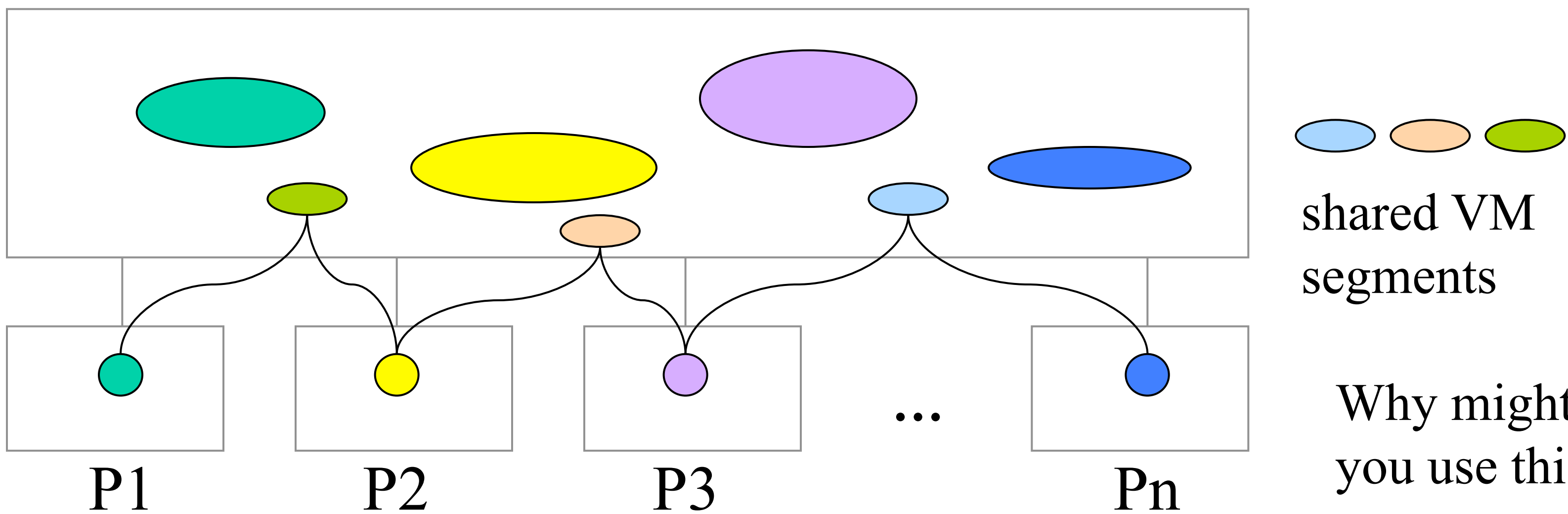
- Process model
  - Each process's data is private, but can create shared space
  - Example: Linux shared memory segments
- Lightweight process (LWP) / thread model
  - Virtual memory is global, shared between LWP / threads
  - Example: Pthreads, Cilk (lazy, lightweight threads)
- Language-based model
  - Threads have shared /private data, built on runtime system
  - Example: OpenMP, Java
- Global address space (for distributed memory)
  - Language or library supported

Shared Memory Programming – IPC (Really!?)

- Multiple processes can run in parallel concurrently
- Multiple processes can be used in a parallel execution
- How?
  - Inter-process communication (IPC)
    - Sockets
  - Message passing (MPI)
    - IPC
    - Memory mapping implementation
  - Through file system (yucky, but possible)
- Effectively programming as a multi-computer system
- BUT, get advantage of shared file system, devices, ... !!!

Shared Memory Programming – Process

- Shared memory parallel programming possible only if processes can access same virtual memory space (some)
- One way is to use OS VM address sharing mechanisms
  - Linux (*shm*) memory mapping and synchronization
  - Incur overheads of using OS support
    - Less than going through IPC and network interfaces though!



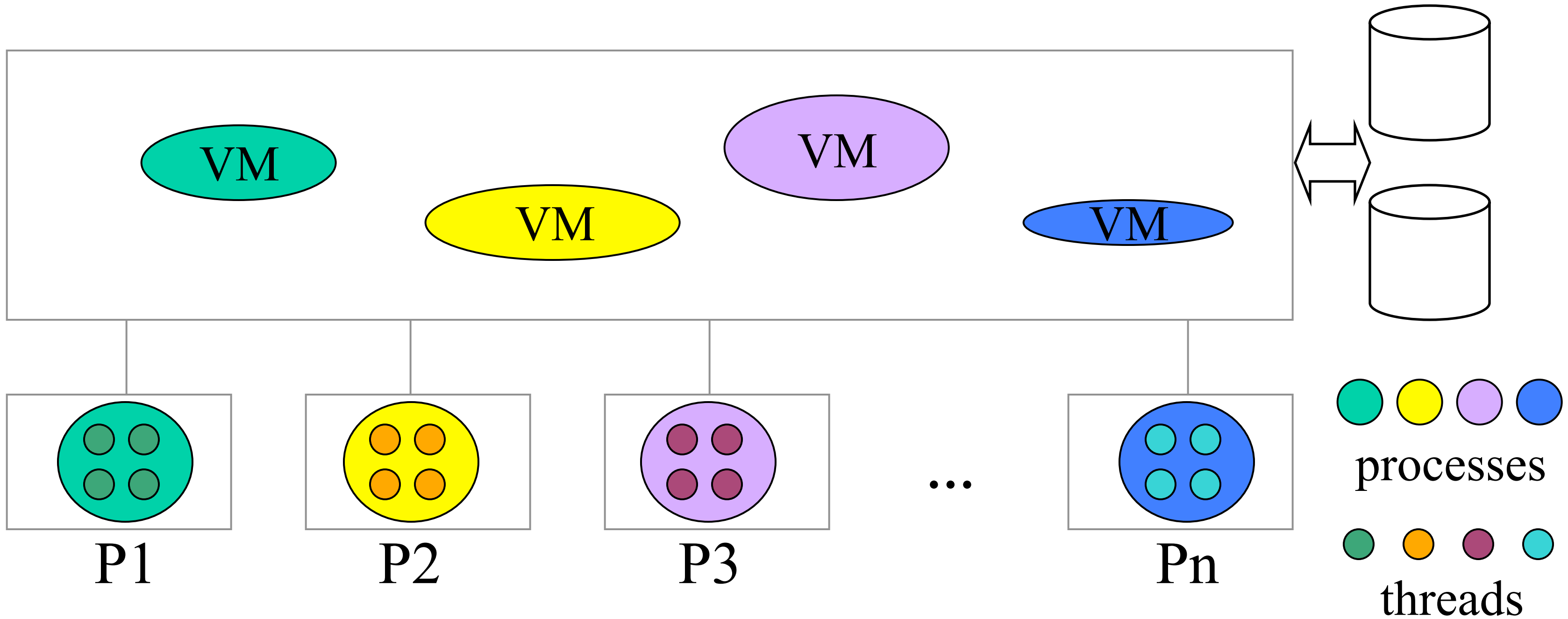
Why might you use this?

Shared Memory Programming – Multithreading

- Process-level parallel programming via virtual memory space sharing suffers from overheads
  - Processes are heavy-weight and cost more to switch
  - Interaction with OS mechanisms not very clean
- Desire way to:
  - Create lighter-weight units of computation
  - Program light-weight computation interaction
- Must be able to address shared data directly
  - Object *x* is the same object *x* no matter which computation unit references it
  - The virtual memory space (or some portion) is shared

Multi-threading

- Creation of "threads of execution" that share process VM



- Threads inherit process's VM address space
- All global data AND instructions are shared
- Thread-private data possible
  - Through memory allocation and local variables

Multi-threading: Advantages and Disadvantages

- Advantages
  - Light-weight computation
  - Fast thread switching
  - Shared data through memory addressing
  - Sychronization support through shared memory
  - Low overhead
  - User has control over threads of execution
- Disadvantages
  - User has control over threads of execution and memory
  - Prone to concurrent programming (synchronization) errors
  - Performance of memory system may be harder to optimize

Shared Memory Parallel Programming

- What does the user have to do?
  - Decide how to decompose computation into parallel parts
  - Create (and destroy) threads to support decomposition
  - Add synchronization to satisfy dependences
- In some sense, similar to message passing programming
  - Think distributed memory, then program share memory
- Execution on a shared memory multiprocessor
  - OS can run threads on available processors
  - Threads run concurrently and can run in parallel
  - Cache coherency kicks in to support consistency model
- Synchronization programming is the hard part!

What's a thread?

- Control perspective
  - A thread is a single stream of control in a program
  - A thread can execute an instruction stream
- State perspective
  - A thread embodies a state of execution
  - It contains an instruction pointer, a stack, registers, …
- "Thread of execution"
  - Control plus state
- Threads inherit from parent process
  - Virtual address space, file descriptors, …
  - Threads are peers, only have parent relationship to process

General Thread Structure

- Typically, a thread is the execution of a piece of code
  - Represents a portion of the program (light-weight)
  - Given a well-defined entry point (e.g., routine)
  - Inherits process's symbol table (e.g., to get to libraries)
- Task-based parallelism
  - Parallel parts form separate procedures or functions
  - Kick off thread with specific routine
  - Kick off thread with a common routine and then specialize
- Data-based parallelism
  - Invoke threads to work on different data
  - Automatic with shared memory support

Why threads?

- Portable, widely-available programming model
  - Requires OS support, but practically all OSes do
- Easier to program (some say)
- Efficiencies in scheduling
  - Versus processes (Why?)
- Efficiencies in latency hiding
  - I/O, communication (How?)
- More dynamic concurrency
- Requires code to be *thread-safe*

POSIX Pthreads

- POSIX standard multi-threading interface
  - For general multi-threaded concurrent programming
  - Largely independent across implementations
  - Broadly supported on different platforms
  - Common target for library and language implementation
- Provides primitives for
  - Thread creation and management
  - Synchronization
- We will only look at a subset of Pthreads
- See webpage for links to Pthreads programming

Thread Creation

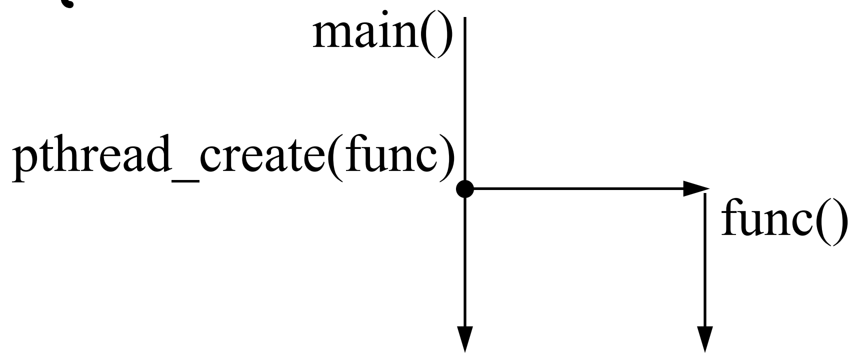
```
#include <pthread.h>
int pthread_create(
  pthread_t *thread_id,
  const pthread_attr_t *attribute,
  void *(*thread_function) (void *),
  void *arg);
```

- `thread_id`
  - thread's unique identifier
- `attribute`
  - contain details on scheduling policy, priority, stack, ...
- `thread_function`
  - function to be run in parallel (entry point)
- `arg`
  - arguments for function `func`

Example of Thread Creation

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

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



Pthread Termination

```
void pthread_exit(void *status)
```

- Terminates the currently running thread
- Implicitly called when function called in `pthread_create` returns

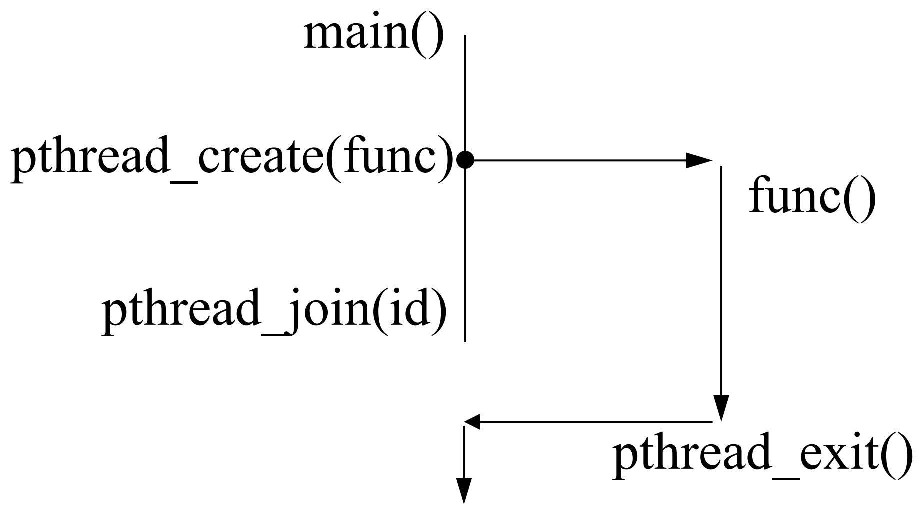
Thread Joining

```
int pthread_join(
  pthread_t thread_id,
  void **status);
```

- Waits for thread `thread_id` to terminate
  - Either by returning
  - Or by calling `pthread_exit()`
- Status receives the return value or the value given as argument to `pthread_exit()`

Thread Joining Example

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

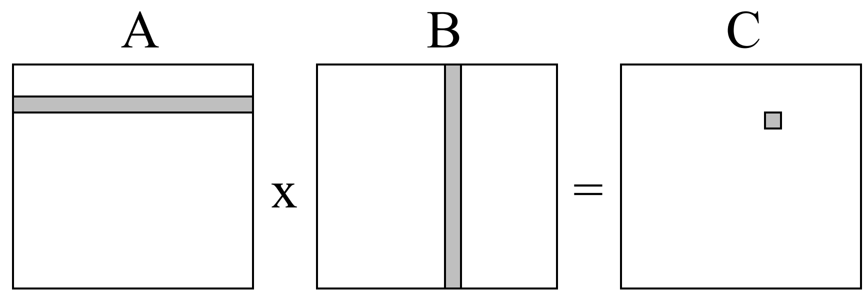


General Program Structure

- Encapsulate parallel parts in functions
- Use function arguments to parameterize thread behavior
- Call `pthread_create()` with the function
- Call `pthread_join()` for each thread created
- Need to take care to make program "thread safe"

Matrix Multiply

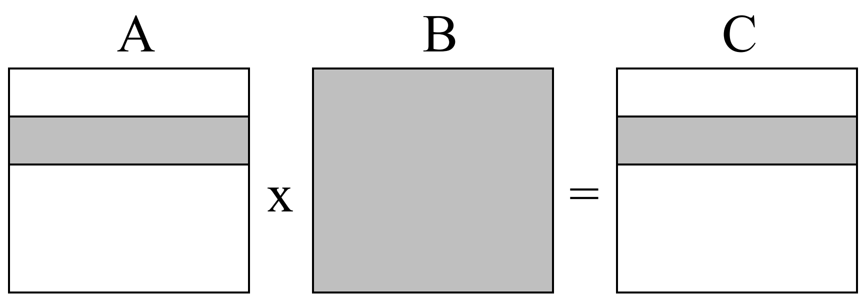
- A x B = C
- A[i,:] • B[:,j] = C[i,j]




```
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
- If we have p processors, n/p rows to each processor
- Corresponds to partitioning i-loop



Matrix Multiply: Parallel Part

```
void mmult(void* s)
{
 int slice = (int) s;
 int from = (slice*n)/p;
 int to = ((slice+1)*n)/p;
 for(i=from; i<to; 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];
 }
}
```

Matrix Multiply: Main

```
int main()
{
 pthread_t thrd[p];

 for( i=0; i<p; i++ )
   pthread_create(&thrd[i], NULL,
                               mmult,(void*) i);

 for( i=0; i<p; i++ )
   pthread_join(thrd[i], NULL);
}
```

Pthread Process Management

- `pthread_create()`
  - Creates a parallel thread executing a given function
  - Passes function arguments
  - Returns thread identifier
- `pthread_exit()`
  - terminates thread.
- `pthread_join()`
  - waits for particular thread to terminate

Pthreads Synchronization

- Create/exit/join
  - Provide some coarse form of synchronization
  - "Fork-join" parallelism
  - Requires thread creation/destruction
- Need for finer-grain synchronization
  - Mutex locks
  - Condition variables

Pthreads “Hello World”

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS     5

void *TaskCode(void *argument)
{
   int tid;

   tid = *((int *) argument);
   printf("Hello World! It's me, thread %d!\n", tid);

   /* optionally: insert more useful stuff here */

   return NULL;
}

int main(void)
{
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int rc, i;

   /* create all threads */
   for (i=0; i<NUM_THREADS; ++i) {
      thread_args[i] = i;
      printf("In main: creating thread %d\n", i);
      rc = pthread_create(&threads[i], NULL, TaskCode, (void *) &thread_args[i]);
      assert(0 == rc);
   }

   /* wait for all threads to complete */
   for (i=0; i<NUM_THREADS; ++i) {
      rc = pthread_join(threads[i], NULL);
      assert(0 == rc);
   }

   exit(EXIT_SUCCESS);
}
```

Mutex Locks – Create, Destroy

- Creates a new mutex lock `mutex`

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

- Destroys the mutex specified by mutex.

```
pthread_mutex_destroy(
   pthread_mutex_t *mutex);
```

Mutex Locks – Lock

- Tries to acquire the lock specified by mutex

```
pthread_mutex_lock(
   pthread_mutex_t *mutex)
```

- If mutex is already locked
  - Calling thread blocks until mutex is unlocked
- If mutex is not locked
  - Mutex is locked and calling thread returned

- Mutually exclusive

Mutex Locks – Unlock

- Unlock mutex lock

```
pthread_mutex_unlock(
    pthread_mutex_t *mutex);
```

- If calling thread has mutex currently locked
  - Mutex will be unlocked
  - If other threads are blocked waiting on this mutex
    - one will unblock and acquire mutex
    - which one is determined by the scheduler

Use of Mutex Locks

- Pthreads provides only exclusive locks
- Other systems allow other types of locks
  - Shared-read, exclusive-write locks
- Critical sections
  - Code sections only to be executed by one thread
- Applications
  - Update of shared variables
  - Queues
  - Stacks
- Problem is that mutex locks can be inefficient
  - Excessive polling

Condition Variables

- Condition variables are objects for thread sychronization
- Allows a thread to block itself until specified data reaches a predefined state
- A condition variable is associated with a predicate
  - When predicate become true, the condition variable is used to signal thread(s) waiting on the condition
- A condition variable always has an associated mutex
- A thread locks the mutex and tests the predicate
  - If not true, threads waits on condition variable
- A blocked thread is released on a signal
  - It then acquires the mutex before resuming

Condition variables – Init, Destroy

- Creates a new condition variable `cond`

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

- Destroys the condition variable `cond`

```
pthread_cond_destroy(
    pthread_cond_t *cond)
```

Condition Variables – Wait

- Wait on `cond`

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

- Blocks the calling thread
- Unlocks the mutex on success

Condition Variables – Signal

- Unblocks one thread waiting on `cond`

```
pthread_cond_signal(
    pthread_cond_t *cond)
```

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

Condition Variables – Broadcast

- Unblocks all threads waiting on `cond`

```
pthread_cond_broadcast(
    pthread_cond_t *cond)
```

- If no thread waiting, then broadcast is a no-op

Use of Condition Variables

- To implement signal-wait synchronization
- Be careful
  - A signal is "forgotten" if there is no corresponding wait that has already happened.

PIPE Example

- Send picture images into a pipeline
- Transformations performed at each pipeline stage
- Separate flag for each picture image

```
P1:for( i=0; i<num_pics, read(in_pic); i++ ) {
     int_pic_1[i] = trans1( in_pic );
     signal( event_1_2[i] );
   }
P2: for( i=0; i<num_pics; i++ ) {
     wait( event_1_2[i] );
     int_pic_2[i] = trans2( int_pic_1[i] );
     signal( event_2_3[i] );
   }
```

PIPE Using Pthreads

- Replace by Pthreads condition variable wait/signal
  - Will not work
  - Signals before a wait are forgotten
  - Need to remember a signal
- Create a signal semaphore

```
semaphore_signal(i) {
  pthread_mutex_lock(&mutex_rem[i]);
  arrived [i]= 1;
  pthread_cond_signal(&cond[i]);
  pthread_mutex_unlock(&mutex_rem[i]);
}
```

PIPE Using Pthreads

```
sempahore_wait(i) {
  pthreads_mutex_lock(&mutex_rem[i]);
  if( arrived[i] == 0 ) {
   pthreads_cond_wait(&cond[i],
                                  mutex_rem
  [i]);
  }
  arrived[i] = 0;
  pthreads_mutex_unlock(&mutex_rem[i]);
}
```

PIPE with Pthreads

```
P1:
for( i=0; i<num_pics, read(in_pic); i++ ) {
    int_pic_1[i] = trans1( in_pic );
    semaphore_signal( event_1_2[i] );
  }
P2:
for( i=0; i<num_pics; i++ ) {
    semaphore_wait( event_1_2[i] );
    int_pic_2[i] = trans2( int_pic_1[i] );
    semaphore_signal( event_2_3[i] );
  }
```

Note on Semaphores

- Many shared memory programming systems (other than Pthreads) have semaphores as basic primitive
- If they do, you should use it, not construct it yourself
- Implementation may be more efficient than what you can do yourself

Reality Bites ...

- Thread create/exit/join is not so cheap
- More efficient if could have a parallel program where
  - Create/exit/join would happen rarely (once!)
  - Cheaper synchronization were used
- We need something that makes all threads wait
  - Barrier synchronization

Barrier Synchronization

- A wait at a barrier causes a thread to wait until all threads have performed a wait at the barrier
- At that point, they all proceed

Implementing Barriers in Pthreads

- Count the number of arrivals at the barrier
- Wait if this is not the last arrival
- Make everyone unblock if this is the last arrival
- Since the arrival count is a shared variable, enclose the whole operation in a mutex lock-unlock

Implementing Barriers in Pthreads

```
void barrier()
{
 pthread_mutex_lock(&mutex_arr);
 arrived++;
 if (arrived<N) {
     pthread_cond_wait(
          &cond, &mutex_arr);
 }
 else {
     pthread_cond_broadcast(&cond);
     arrived=0; /* next barrier */
     }
 pthread_mutex_unlock(&mutex_arr);
}
```

Note on Barriers

- Many shared memory programming systems (other than Pthreads) have barriers as basic primitive
- If they do, you should use it, not construct it yourself
- Implementation may be more efficient than what you can do yourself

Busy Waiting

- Not an explicit part of the API
- Available in shared memory programming environment

```
initially:  flag = 0;

P1:     produce data;
        flag = 1;

P2:     while( !flag ) ;
        consume data;
```

Use of Busy Waiting

- On the surface, simple and efficient
- In general, not a recommended practice
- Often leads to messy and unreadable code
  - Blurs data/synchronization distinction
- On some architectures, may be inefficient
- May not even work as intended
  - Depending on consistency model

Private Data in Pthreads

- To make a variable private in Pthreads, you need to make an array out of it
- Index the array by thread identifier, which you can get by the pthreads_self() call
- Not very elegant or efficient

Other Primitives in Pthreads

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

Next Class

- Shared memory parallel programming
- OpenMP