Parallel Programming
POSIX Threads Programming

Pham Quang Dung

Hanoi, 2012

Outline

Process

- Computer program is a passive collection of instructions
- **Process** is an instance of a computer program that is being executed
- Several processes may be associated with the same program
- A process consists of following resources
  - Image of the executable machine code
  - Memory :
    - executable code,
    - A call stack (keep track of actives routines)
    - Heap (to hold intermediate data generated during run time)
    - etc.
  - Descriptors of resources allocated to the process : file descriptors (Unix terminology), handles (Windows)
  - Security attributes : process owner and set of permissions of the process
  - Process state (context) : content of registers, physical memory addressing, program counter, etc.

Thread

- **Thread** is an independent stream of instructions that can be scheduled to run by the OS
- Threads are contained inside a process : use and exists within the process resources
- A thread maintains its own
  - Stack pointer
  - Registers
  - Scheduling properties (policy or priority)
  - Set of pending and blocked signals
  - Thread specific data
- A thread has its own independent flow of control as long as its parent process exists and the OS supports it
- A thread dies if its parent process dies

Thread

- Threads within the same process share resources
  - Changes made by one thread to shared system resources (e.g., closing a file) are seen by all other threads of the same process
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory location is possible, but synchronization must be made by programmers
- All threads within a process share the same address space
- Inter-thread communication is more efficient than inter-process communication

Pthreads

- In modern, multi-cpu machines, pthreads are suited for parallel programming
- In order for a program to take advantages, it must be organized into discrete, independent tasks which can execute concurrently

Pthreads

- Set of C language programming types and procedure calls, implemented with a pthread.h header/include file
- Pthreads API was defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard
- Library :
  - IBM Library functions C/C++ : http ://publib.boulder.ibm.com/infocenter/zos/v1r11/index.jsp ?topic=/
  - POSIX Threads Programming : https ://computing.llnl.gov/tutorials/pthreads/#Misc
  - about 100 subroutines, but we consider only fundamental functions
- Pthreads API is organized into four groups
  - Thread management
  - Mutexes
  - Condition variables
  - Synchronization

Compiling threaded programs

Compiler / Platform	Compiler Command	Description
INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PathScale Linux	`pathcc -pthread`	C
	`pathCC -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, BG/L, BG/P	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++
IBM BG/L and BG/P	`bgxlc_r / bgcc_r`	C (ANSI / non-ANSI)
	`bgxlC_r, bgxlc++_r`	C++

source : https ://computing.llnl.gov/tutorials/pthreads/

Outline

Thread management

- pthread_create(pthread_t * thrd, pthread_attr_t* attr, void* start_routine, void* arg)
- pthread_exit(void *value_ptr)
- pthread_cancel(pthread_t thr)
- pthread_join(pthread_t thr, void** status)
- pthread_detach(pthread_t thr)

Thread management : Creating a thread

```
1 int pthread_create(pthread_t * thrd,
                    pthread_attr_t* attr,
3                   void* start_routine,
                    void* arg)
```

- Attributes specified by attr
- The created thread will execute the start_routine with arg as its argument
- Once created, threads are peers, and may create other threads. No dependency between threads

Thread management : Terminating a thread

```
void pthread_exit(void *value_ptr)
```

A thread is terminated in several cases

- The thread finishes normally from its starting routine. Its work is done
- The thread makes a call to pthread_exit(...) subroutine whether its work is done or not
- The thread is cancelled by another thread via the pthread_cancel(...) subroutine
- The entire process terminates due to making a call to exec() or exit()
- The main finishes without explicitly calling pthread_exit(...)

Thread management : Terminating a thread

- The pthread_exit(...) does not close any files
- If main finishes before the threads it spawned, all these threads will be terminated because main() is done and no longer exists to support the threads.
- By explicitly calling to pthread_exit(...) , main() will block until all threads it created terminate

Thread management : example

```
1  void *proc(void *threadid)
   {
3      long tid = (long)threadid;
       printf("Hello World! I am thread %ld!\n", tid);
5      pthread_exit(NULL);
   }
7  int main (int argc, char *argv[])
   {
9      int NUM_THREADS = atoi(argv[1]);
       pthread_t threads[NUM_THREADS];
11     for(long t=0; t < NUM_THREADS; t++){
           printf("Main: creating thread %ld\n", t);
13         int rc = pthread_create(&threads[t], NULL, proc, (void *)t);
           if (rc){
15             printf("ERROR; return code  is %d\n", rc);    exit(-1);
           }
17     }
       pthread_exit(NULL);
19 }
```

Thread management : example without call to pthread_exit(NULL) of main()

```
1  void* proc(void* arg){
     long tid = (long) arg;
3    for(int i = 1; i <= 100; i++){
       cout << "thread " << tid << "  has i = " << i << endl;
5      sleep(1);
     }
7    pthread_exit(NULL);
   }
9  int main(int argc, char** argv){
     int nbThreads = atoi(argv[1]);
11   pthread_t thrd[nbThreads];
     for(long t = 0; t < nbThreads; t++){
13     int rc = pthread_create(&thrd[t], NULL, proc, (void*)t);
       if(rc){
15       cout << "Main Error when create thread with return code is =
             " << rc << endl;
         exit(-1);
17     }
     }
19   sleep(3);
     //pthread_exit(NULL);
21 }
```

Thread management : passing arguments

- Argument will be a pointer to a structure encapsulating all variables that threads want to handle
- The structure should contain the information about the id of the thread

```
1 struct Param{
    long tid;
3   int* a;
    int sz;
5   int* r;
};
```

Thread management : example pthread_cancel

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>

using namespace std;

void* proc(void* arg){
  long tid = (long) arg;
  for(int i = 0; i < 10; i++){
    cout << "My id is " << tid << " having i = " << i << endl;
    sleep(1);
  }
}

main(){
  pthread_t T;
  int rc = pthread_create(&T,NULL,proc,(void*)123);
  sleep(3);
  cout << "main terminates the thread" << endl;
  rc = pthread_cancel(T);
  pthread_exit(NULL);
}
```

Thread management : example pthread_cancel

- Function main() creates 3 threads (0, 1, 2), each thread increments its own local variable *i* from 1 to 20
- If the value of *i* of the thread 0 reach 10, then it cancel the execution of thread 1

```
struct Param{
  long tid;
  pthread_t* arr;
};
```

Thread management : example pthread_cancel

```
1  void* proc(void* arg){
     int old_cancel_state;
3    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &old_cancel_state);

5    Param* p = (Param*)arg;
     long tid = p->tid;
7    for(int i = 1; i <= 20; i++){
       cout << "thread " << tid << " has i = " << i << endl;
9      if(tid == 0 && i == 10){
         int rc = pthread_cancel(p->arr[1]);
11         if(rc) cout << "thread " << tid << " cancel thread 1
               returning error code " << rc << endl;
       }
13     sleep(1);
     }
15   pthread_exit(NULL);
   }
```

Thread management : example pthread_cancel

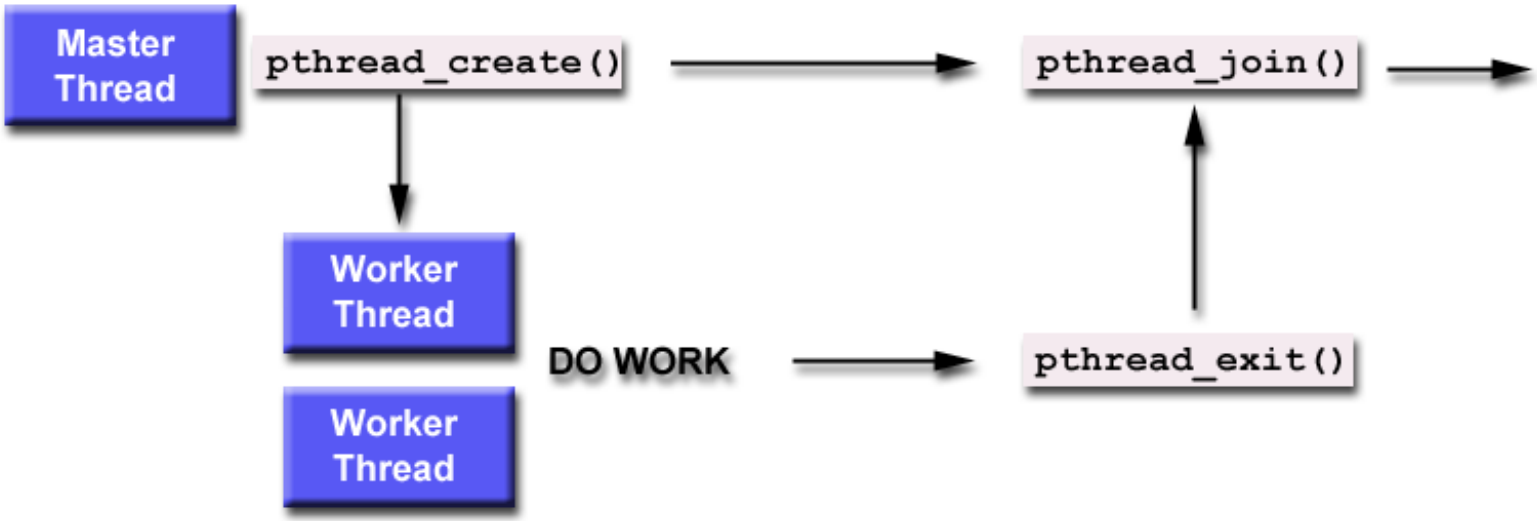
```
int main(int argc, char** argv){
  int nbThreads = atoi(argv[1]);
  pthread_t thrd[nbThreads];
  Param p[nbThreads];

  for(long t = 0; t < nbThreads; t++){
    p[t].tid = t;
    p[t].arr = thrd;

    int rc = pthread_create(&thrd[t], NULL, proc, (void*)&p[t]);
    if(rc){
      cout << "Main Error when create thread with return code is =
          " << rc << endl;
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Thread management - shared global variables

- A created thread can access variables declared as global variables

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
int A;
void* proc(void* arg){
  long tid = (long) arg;
  for(int i = 0; i < 10; i++){
    cout << "My id is " << tid << " having i = " << i << " A = " <<
        A << endl;
    sleep(2);
  }
}
main(){
  pthread_t T;
  A = 0;
  int rc = pthread_create(&T,NULL,proc,(void*)123);
  for(int i = 0; i <= 100; i++){
    A++;
    sleep(1);
  }
  pthread_exit(NULL);
```

Thread management : joining and detaching threads



source : https ://computing.llnl.gov/tutorials/pthreads/

- Joining is one way to accomplish synchronization between threads

Thread management : joining and detaching threads

```
int pthread_join(pthread_t thrd, void **status)
```

- The subroutine pthread_join() blocks the calling thread until the thread thrd terminates
- We can obtain the target thread's termination return status if it is specified in the target thread's call to pthread_exit
- To explicitly create a thread as joinable or detached, we use the attr in the method pthread_create(...)

Thread management : joining and detaching threads

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
void* proc(void* arg){
  long tid = (long) arg;
  for(int i = 0; i < 5*tid; i++){
    cout << "My id is " << tid << " having i = " << i << endl;
    sleep(tid);
  }
}
```

Thread management : joining and detaching threads

```
main(){
  int sz = 2;
  pthread_t T[sz+1];
  void* status;
  for(long i = 1; i <= sz; i++)
    int rc = pthread_create(&T[i],NULL,proc,(void*)i);

  for(long i = 1; i <= sz; i++){
    int rc = pthread_join(T[i], &status);
    cout << "Main joins with the created thread " << i << " with rc
         = " << rc << " status = " << status << endl;
  }
  cout << "Main continues" << endl;
  pthread_exit(NULL);
}
```

Thread management : joining and detaching threads

```
My id is 1 having i = 0
My id is 1 having i = 1
My id is 2 having i = 0
My id is 1 having i = 2
My id is 2 having i = 1
My id is 1 having i = 3
My id is 2 having i = 2
My id is 1 having i = 4
My id is 2 having i = 3
Main joins with the created thread 1 with rc = 0 status = 0
My id is 2 having i = 4
My id is 2 having i = 5
My id is 2 having i = 6
My id is 2 having i = 7
My id is 2 having i = 8
My id is 2 having i = 9
Main joins with the created thread 2 with rc = 0 status = 0
Main continues
```

Thread management : joining and detaching threads

```
#define MAX_SIZE 100000000

int x[MAX_SIZE];
int r[MAX_SIZE];

struct Param{
  long tid;
  int* a;
  int sz;
  int* r;
};
```

Thread management : joining and detaching threads

```
1  void* sum(void* arg){
     Param*  p;
3    p = (Param*) arg;

5    long tid;
     tid = p->tid;
7
     *(p->r) = -1;
9    for(int i = 0; i < p->sz; i++){
       *(p->r) = *(p->r) > *(p->a+i) ? *(p->r) : *(p->a+i);
11   }
     pthread_exit(NULL);
13 }
```

Thread management : joining and detaching threads

```
1  int main(int agrc, char** argv){
     pthread_t threads[NUM_THREADS];
3    pthread_attr_t attr;
     pthread_attr_init(&attr);
5    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
     void* status;
7
     int sz = N/NUM_THREADS;
9    Param pa[NUM_THREADS];
     for(int i = 0; i < NUM_THREADS; i++){
11     pa[i].tid = i; pa[i].a = x+i*sz; pa[i].sz = sz; pa[i].r = r+i;
     }
13   for(long t = 0; t < NUM_THREADS; t++){
       int rc = pthread_create(&threads[t],NULL,sum,(void*)&pa[t]);
15     if(rc){
         printf("ERROR, return code %d\n",rc); exit(-1);
17     }
     }
19   pthread_attr_destroy(&attr);
```

Thread management : joining and detaching threads

```
1   for(long t = 0; t < NUM_THREADS; t++){
      int rc = pthread_join(threads[t], &status);
3     if(rc){
        printf("ERROR, return code %d\n",rc);
5       exit(-1);
      }
7     printf("main finishes joining with thread %ld having a status
          of %ld\n",t,(long)status);
    }
9
    printf("main finishes\n");
11  int M = -1;
    for(int i=0;i<NUM_THREADS;i++) {
13    M = M > r[i] ? M : r[i];
      printf("r[%d] = %d\n",i,r[i]);
15  }
    pthread_exit(NULL);
17 }
```

Thread management : detaching threads

```
1 int pthread_detach(pthread_t t)
```

- The pthread_detach(pthread_t t) routine can be used to explicitly detach a thread even though it was created as joinable
- Storage for the thread t can be reclaimed when that thread terminates
- If we know in advance that a thread will never need to join with other threads, then consider creating it in a detached state. Some system resources may be able to be freed
- If the thread t does not terminate, the subroutine pthread_detach(pthread_t t) does not cause t to terminate
- Why pthread_detach(pthread_t t) ?
  - To detach threads on which pthread_join was waiting to avoid unbounded waiting periods

Thread management : detaching threads

```
1  void* proc(void* arg){
     printf("thread start with argument %s\n", arg);
3    for(int i = 1; i <= 10; i++){
       cout << "thread proc " << arg << " has i = " << i << endl;
5      if(i == 5){
         cout << "i = 5, detach self "  << endl;
7        pthread_detach(pthread_self());
       }
9      sleep(1);
     }
11   pthread_exit(NULL);
   }
13 int main(int argc, char** argv){
     pthread_t t;
15   void *status;
     int rc = pthread_create(&t,NULL,proc,(void*)"thread 2");
17   cout << "main() starts joining..." << endl;
     rc = pthread_join(t,&status);
19   if(rc)
      cout << "main() join thread returns error code " << rc << endl;
21   else
      cout << "main() Join -> OK " << endl;
23   pthread_exit(NULL);
   }
```

Thread management : detaching threads

execution result : on MAC OS

```
main() starts joining...
thread start with argument thread 2
thread executing proc has i = 1
thread executing proc has i = 2
thread executing proc has i = 3
thread executing proc has i = 4
thread executing proc has i = 5
i = 5, detach self
main() join thread returns error code 3
thread executing proc has i = 6
thread executing proc has i = 7
thread executing proc has i = 8
thread executing proc has i = 9
thread executing proc has i = 10
```

Thread management : Miscellaneous routines

- pthread_self() : returns the unique, system assigned thread ID of the calling thread
- pthread_equal(pthread_t t1, pthread_t t2) : return a non-value if thread t1 and t2 are equal. Otherwise, 0 will be returned

Thread management : Miscellaneous routines

```
int pthread_once(pthread_once_t *once_control, void(*init_routine)
    ())
```

- There may be several threads calling init_routine
- But only the first call causes the routine init_routine to run
- Other threads that reach the same point will be delayed until the execution of init_routine of the first thread calling it finishes
- The mechanism is used when we want the data structure is initialized by only one thread. An alternative is to use mutex but it is waste of resource.

Thread management : pthread_once

```
int main(int argc, char** argv){
  int nbThrds = 4;
  pthread_t t[nbThrds];
  void* status;

  for(long i = 0;  i <= nbThrds; i++){
    int rc = pthread_create(&t[i],NULL,proc,(void*) i);
    if(rc){
      cout << "pthread_create in main() returns error code " << rc
           << endl;
      exit(-1);
    }
  }
  for(int i = 0; i <= nbThrds; i++){
    int rc = pthread_join(t[i],&status);
    if(rc){
      cout << "main(), pthread_join thread " << i << " returns
          error code " << rc << " status = " << status << endl;
      exit(-1);
    }
  }
  cout << "global count = " << global_count << endl;
  pthread_exit(NULL);
}
```

Thread management : pthread_once

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int global_count = 0;

void init_routine(void ){
  cout << "init is called...." << endl;
  global_count++;
}

void* proc(void* arg){
  long tid = (long)arg;
  cout << "thread " << tid << " is running self = " << pthread_self
      () << endl;

  init_routine();

  pthread_exit(NULL);
}
```

Thread management : pthread_once

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int global_count = 0;

void init_routine(void ){
  cout << "init is called...." << endl;
  global_count++;
}

void* proc(void* arg){
  long tid = (long)arg;
  cout << "thread " << tid << " is running self = " << pthread_self
      () << endl;

  pthread_once(&once_control,init_routine);

  pthread_exit(NULL);
}
```

Thread management - calculate sum

- Input : a text file contains an array A
- Write a parallel problem using pthread for calculating the sum of all items of A

Thread management - calculate sum

First and not correct implementation (line 16)

```
#define MAX 100000000
#define MAX_THREADS 10
int A[MAX];
long n;
long d;
int k; // so luong thread
long tong;

void* subSum(void* arg){
  long tid = (long)arg;
  long start = tid*d;
  long end = (tid+1)*d -1;
  if(tid == k-1) end = n-1;

  for(long j = start; j <= end; j++)
    tong = tong + A[j];
}
```

Thread management - calculate sum

First and not correct implementation

```
int main(int argc, char** argv){
  k = atoi(argv[1]);
  docdulieu(argv[2]);
  Timer ti;
  double t0 = ti.getElapsedTime();
  d = n/k;
  pthread_t T[k];

  for(long i = 0; i < k; i++)
    pthread_create(&T[i],NULL,subSum,(void*)i);

  printf("Tong = %ld  time = %lf\n",tong,ti.getElapsedTime()-t0);
  pthread_exit(NULL);
}
```

Thread management - calculate sum

Second and not efficient implementation

```
#define MAX 100000000
#define MAX_THREADS 10
int A[MAX];
long n;
long d;
int k; // so luong thread
long tong;
long S[MAX_THREADS];//S[i] la tong con ma thread i tinh toan

void* subSum(void* arg){
  long tid = (long)arg;
  long start = tid*d;
  long end = (tid+1)*d -1;
  if(tid == k-1) end = n-1;

  S[tid] = 0;
  for(long j = start; j <= end; j++)
    S[tid] = S[tid] + A[j];
}
```

Thread management - calculate sum

Second and not efficient implementation (line 13)

```
1  int main(int argc, char** argv){
     k = atoi(argv[1]);
3    docdulieu(argv[2]);
     Timer ti;
5    double t0 = ti.getElapsedTime();
     d = n/k;
7
     pthread_t T[k];
9    void* status;

11   for(long i = 0; i < k; i++){
       pthread_create(&T[i],NULL,subSum,(void*)i);
13     pthread_join(T[i],&status);
     }
15   tong = 0;
     for(int i = 0; i < k; i++)
17     tong = tong + S[i];
     printf("Tong = %ld  time = %lf\n",tong,ti.getElapsedTime()-t0);
19   pthread_exit(NULL);
   }
```

Outline

Mutex variables

- One of the primary means for
  - implementing threads synchronization
  - protecting shared data when multiple writes occur
- Acts as a lock
  - Only one thread can lock (or own) a mutex variable at any given time
  - No other threads can lock a mutex until the owning thread unlocks it
- Use to prevent race condition

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

source = https ://computing.llnl.gov/tutorials/pthreads/

Mutex variables

- A typical sequence of use of a mutex is as follows
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one thread succeeds and it owns the mutex
  - The owner thread performs some actions
  - The owner thread unlocks the mutex
  - Another thread acquires the mutex and performs its actions
  - Finally, the mutex is destroyed
- We can use "trylock" (unblocking call) instead of "lock" to avoid blocking at the call

Mutex variables

- Initialize a mutex pthread_mutex_init(pthread_mutex_t* mutex, pthread_attr_t attr)
- Destroy a mutex pthread_mutex_destroy(pthread_mutex_t* mutex)
- Locking a mutex pthread_mutex_lock(pthread_mutex_t* mutex)
- NonBlocking lock pthread_mutex_trylock(pthread_mutex_t* mutex)
- Unlocking a mutex pthread_mutex_unlock(pthread_mutex_t* mutex)

Mutex variables - sum calculation

```
#define MAX 100000000
#define MAX_THREADS 10
int A[MAX];
long n;
long d;
int k; // so luong thread
long tong;
pthread_mutex_t mux;

void* subSum(void* arg){
  long tid = (long)arg;
  long start = tid*d;
  long end = (tid+1)*d -1;
  if(tid == k-1) end = n-1;
  long Si = 0;
  for( long j = start; j <= end; j++)
    Si = Si + A[j];
  pthread_mutex_lock(&mux);
  tong = tong + Si;
  printf("Thread %ld compute Si = %ld and update tong = %ld\n",tid,
      Si,tong);
  pthread_mutex_unlock(&mux);
}
```

Mutex variables

```
int main(int argc, char** argv){
  k = atoi(argv[1]);
  tong = 0;
  d = n/k;
  Timer ti;
  double t0 = ti.getElapsedTime();
  pthread_mutex_init(&mux,NULL);
  pthread_t T[k];
  for(long i = 0; i < k; i++)
    pthread_create(&T[i],NULL,subSum,(void*)i);
  void* status;
  for(long i = 0; i < k; i++)
    pthread_join(T[i],&status);
  printf("Tong = %ld  time = %lf\n",tong,ti.getElapsedTime()-t0);
  pthread_mutex_destroy(&mux);
  pthread_exit(NULL);
}
```

Outline

Condition variables

- Another way for threads synchronization
- A condition variable is used in conjunction with a mutex lock

Condition variables

Thread A	Thread B
• Do work up to the point where a certain condition occurs • Lock the mutex variable and check the value of a global variable • Call pthread_cond_wait() to perform a blocking wait for signal from thread **B**. Note that a call to pthread_cond_wait() automatically and atomically unlocked the associated mutex so that it can be used by thread **B** • When signaled, wake up. Mutex is automatically and atomically locked • Explicitly unlock the mutex • Continue	• Do work • Locked the associated mutex • Change the value of a global variable that thread **A** is waiting upon • Check the value of the global variable. If it fulfills the desired condition, then signal thread **A** • Unlock mutex • Continue

Condition variables

- Initializing and destroying condition variables

```
int pthread_cond_init(pthread_cond_t* cond,
                      pthread_condattr_t* attr);

int pthread_cond_destroy(pthread_cond_t *cond)
```

Condition variables

- Waiting and signaling on condition variables

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

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);
```

Condition variables : example

- thread main() will create three threads 1, 2, 3 which share a global variable count
  - thread 2 and 3 augment the value of count each time by one
  - thread 1 wait until the value of count reaches COUNT_LIMIT , it then augment the value of count by 125 and finishes
  - threads 2 and 3 continue augmenting the value of count by one at each step

Condition variables : example

```
int count = 0;

pthread_mutex_t count_mutex;

pthread_cond_t count_threshold_cv;
```

Condition variables : example

```
1  void* inc_count(void* t){
     long my_id = (long)t;
3    for(int i = 0; i < TCOUNT; i++){
       pthread_mutex_lock(&count_mutex);
5      count++;

7      if(count == COUNT_LIMIT){
         pthread_cond_signal(&count_threshold_cv);
9        printf("inc_count(): thread %ld, count = %d, Threshold
             reached\n",my_id,count);
       }
11     printf("inc_count(): thread %ld, count = %d, Unlock mutex\n",
           my_id,count);

13     pthread_mutex_unlock(&count_mutex);

15     /* do some work, so thread can alternate on mutex lock*/
       sleep(2);
17   }

19   pthread_exit(NULL);
   }
```

Condition variables : example

```
void* watch_count(void* t){
  long my_id = (long)t;
  printf("Start watch count, thread %ld\n",my_id);

  pthread_mutex_lock(&count_mutex);
  pthread_cond_wait(&count_threshold_cv, &count_mutex);
  printf("Watch_count(): thread %ld condition singal received\n",
      my_id);
  count += 125;
  printf("watch_count(): thread %ld count is now %d\n",my_id,count)
      ;

  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

Condition variables : example

```
int main(){
  long t1 = 1, t2 = 2, t3 = 3;
  pthread_t threads[3];
  pthread_attr_t attr;

  pthread_mutex_init(&count_mutex,NULL);
  pthread_cond_init(&count_threshold_cv,NULL);

  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0],&attr,watch_count,(void*) t1);
  pthread_create(&threads[1],&attr,inc_count,(void*) t2);
  pthread_create(&threads[2],&attr,inc_count,(void*) t3);

  for(int i = 0; i < NUM_THREADS; i++)
    pthread_join(threads[i],NULL);

  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```