Pthreads Programming

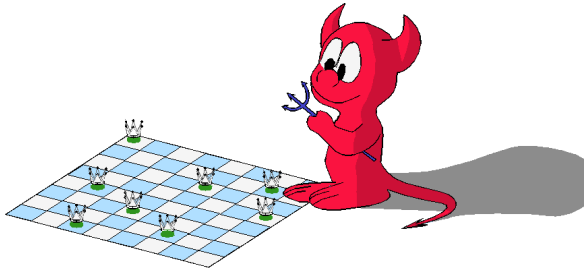
Jeong-Gun Lee

Dept. of Computer Engineering, Hallym University

Email: Jeonggun.Lee@hallym.ac.kr

What are pthreads?

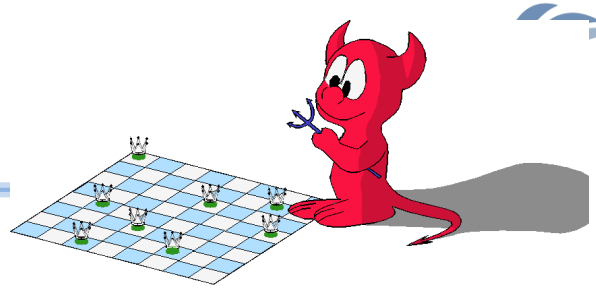
- POSIX 1003.1c defines a thread interface
  - pthreads
  - defines “how threads should be created, managed, and destroyed”



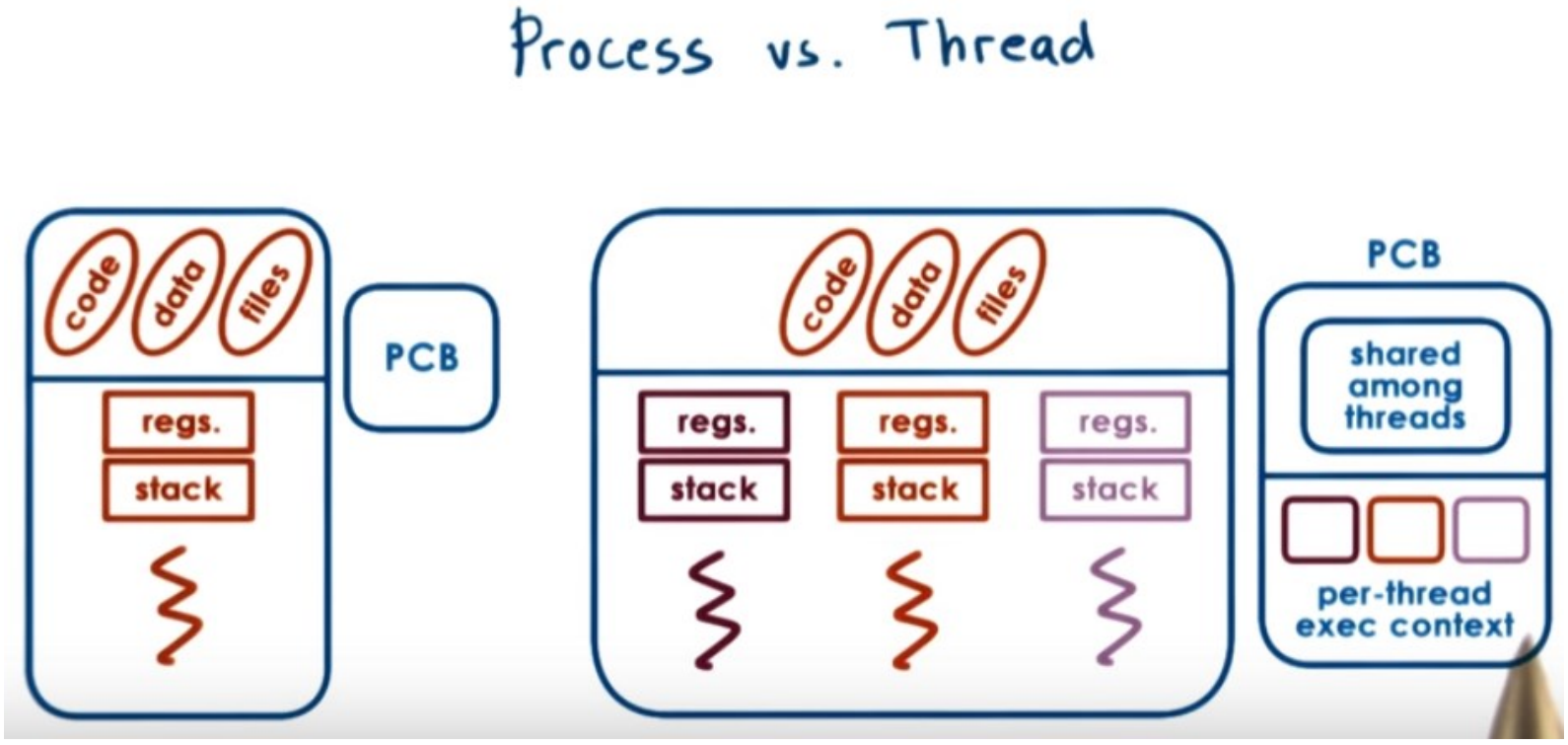
- Unix provides a “pthreads” library
  - APIs to *create and manage threads*
  - you don’t need to worry about the implementation details

POSIX (**포직스**, /ˈpɒzɪks/)는 **이식 가능 운영 체제 인터페이스**(移植可能運營體制 interface, **p**ortable **o**perating **s**ystem **i**nterface)의 약자로, 서로 다른 UNIX OS의 공통 API를 정리하여 이식성이 높은 유닉스 응용 프로그램을 개발하기 위한 목적으로 IEEE가 책정한 애플리케이션 인터페이스 규격이다.
POSIX의 마지막 글자 X는 유닉스 호환 운영체제에 보통 X가 붙는 것에서 유래한다.

Threads



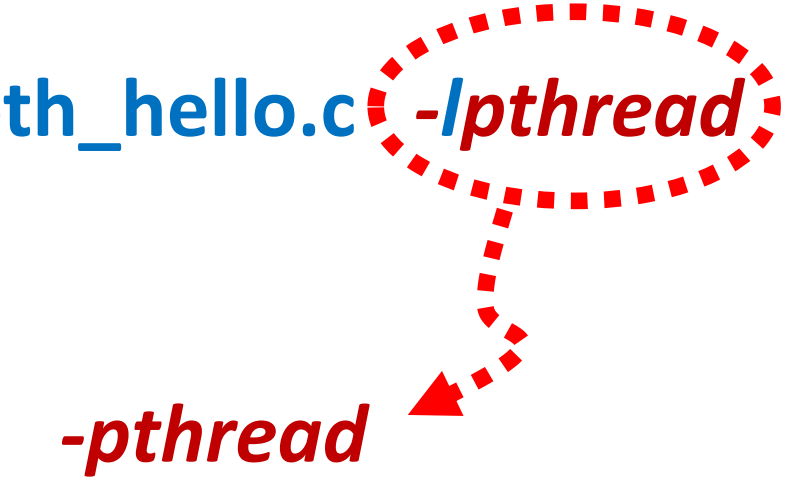
- Light Weight Process



The Basics

- Include the header file:
  - ***<pthread.h>***

- Compile using:
  - **gcc -o pth_hello pth_hello.c *-lpthread***



Let's Dive in- pth_hello.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   long* tidPtr = (long*) threaded;
   tid = *tidPtr;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Output

```
jeong-gun@jeonggun-desktop:~/PTHREAD$ ./pth_hello
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #1!
In main: creating thread 2
Hello World! It's me, thread #2!
In main: creating thread 3
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
Hello World! It's me, thread #5!
```

Some disorder is possible

Creating Threads

- Prototype:
  - int **pthread_create**(**pthread_t *tid**, const pthread_attr_t *tattr, void*(***start_routine**)(void *), void ***arg**);
    - *tid*: an unsigned long integer that indicates a threads id
    - *tattr*: attributes of the thread – usually **NULL** for default
    - *start_routine*: the name of the function the thread starts executing
    - *arg*: the argument to be passed to the start routine – **only one**
  - after this function gets executed, a new thread has been created and is executing the function indicated by ***start_routine***

Waiting for a Thread

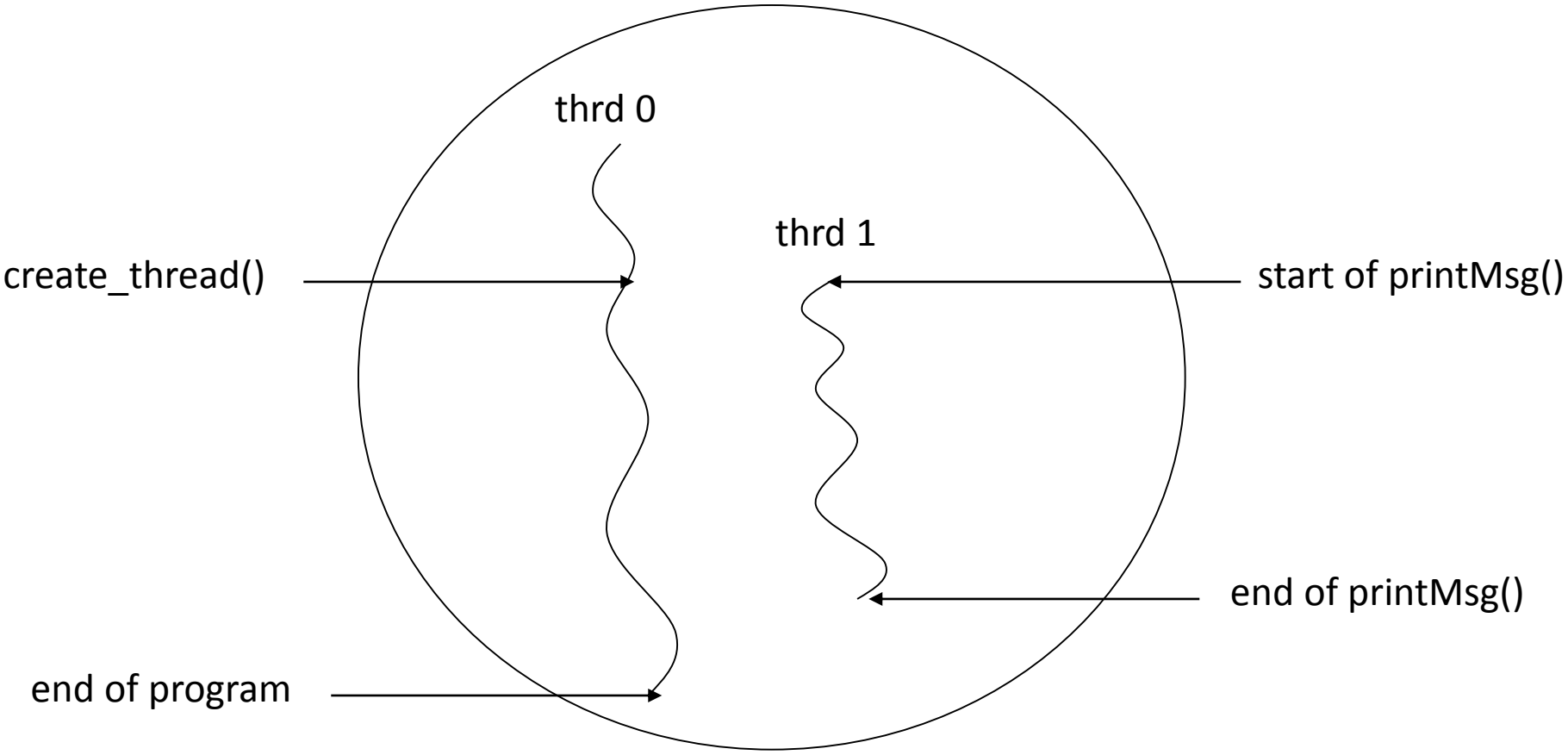
- Prototype:
  - int **pthread_join**(thread_t tid, void **status);
    - *tid*: identification of the thread to wait for
    - *status:* the exit status of the terminating thread – can be **NULL**
  - the thread that calls this function blocks its own execution until the thread indicated by *tid* terminates its execution
    - finishes the function it started with or
    - issues a *pthread_exit()* command – more on this in a minute

Example - pth1.c

```
#include <stdio.h>
#include <pthread.h>

void printMsg(void* arg_msg) {
        char *msg = (char *) arg_ms
        printf("%s\n", msg);
}

int main(int argc, char** argv) {
        pthread_t thrdID;

        printf("creating a new thread\n");
        pthread_create(&thrdID, NULL, (void*)printMsg, (void *) argv[1]);
        printf("created thread %d\n", thrdID);
        pthread_join(thrdID, NULL);

        return 0;
}
```

```
jeong-gun@jeonggun-desktop: ~/PTHREAD
jeong-gun@jeonggun-desktop:~/PTHREAD$ !vi
vi pth1.c
jeong-gun@jeonggun-desktop:~/PTHREAD$ !g
gcc -o pth1 pth1.c -pthread -Wformat
jeong-gun@jeonggun-desktop:~/PTHREAD$ ./pth1 Hello
creating a new thread
created thread 1994953840
Hello
jeong-gun@jeonggun-desktop:~/PTHREAD$
```

Example



Note: thrd 0 is the function that contains *main()* – only one *main()* per program

Exiting a Thread

- To have a thread exit, use *pthread_exit()*

- Prototype:
  - void **pthread_exit**(void *status);
    - *status:* the exit status of the thread – passed to the *status* variable in the *pthread_join()* function of a thread waiting for this one

Example with Return 1

pth_returnErr.c

```
#include <stdio.h>
#include <pthread.h>

void *thread(void *vargp) {
        int value = 42;
        pthread_exit((void *)&value);
}

int main() {
        int i;
        pthread_t tid;
        void *vptr_return;

        pthread_create(&tid, NULL, thread, NULL);
        pthread_join(tid, &vptr_return);

        i = *((int *)vptr_return);
        printf("%d\n",i);
}
```

```
jeong-gun@jeonggun-desktop: ~/PTHREAD
jeong-gun@jeonggun-desktop:~/PTHREAD$ vi pth_returnErr.c
jeong-gun@jeonggun-desktop:~/PTHREAD$ ./pth_returnErr
8384116
jeong-gun@jeonggun-desktop:~/PTHREAD$
```

Example with Return 2

pth_returnErr2.c

```
#include <stdio.h>
#include <pthread.h>

void *thread(void *vargp) {
        int value = 42;
        pthread_exit((void *)&value);
}

int main() {
        int i;
        pthread_t tid;
        void *vptr_return = malloc(sizeof(int));

        pthread_create(&tid, NULL, thread, NULL);
        pthread_join(tid, &vptr_return);

        i = *((int *)vptr_return);
        printf("%d\n",i);
}
```

```
jeong-gun@jeonggun-desktop: ~/PTHREAD
jeong-gun@jeonggun-desktop:~/PTHREAD$ ./pth_returnErr2
8384116
jeong-gun@jeonggun-desktop:~/PTHREAD$
```

Example with Return 3

```
#include <stdio.h>
#include <pthread.h>

void *thread(void *vargp) {
        int *value = (int *)malloc(sizeof(int));
        *value = 42;
        pthread_exit(value);
}

int main() {
        int i;
        pthread_t tid;
        void *vptr_return;

        pthread_create(&tid, NULL, thread, NULL);
        pthread_join(tid, &vptr_return);

        i = *((int *)vptr_return);
        free(vptr_return);
        printf("%d\n",i);
}
```

pth_return.c

```
jeong-gun@jeonggun-desktop: ~/PTHREAD
jeong-gun@jeonggun-desktop:~/PTHREAD$ ./pth_return
42
jeong-gun@jeonggun-desktop:~/PTHREAD$
```

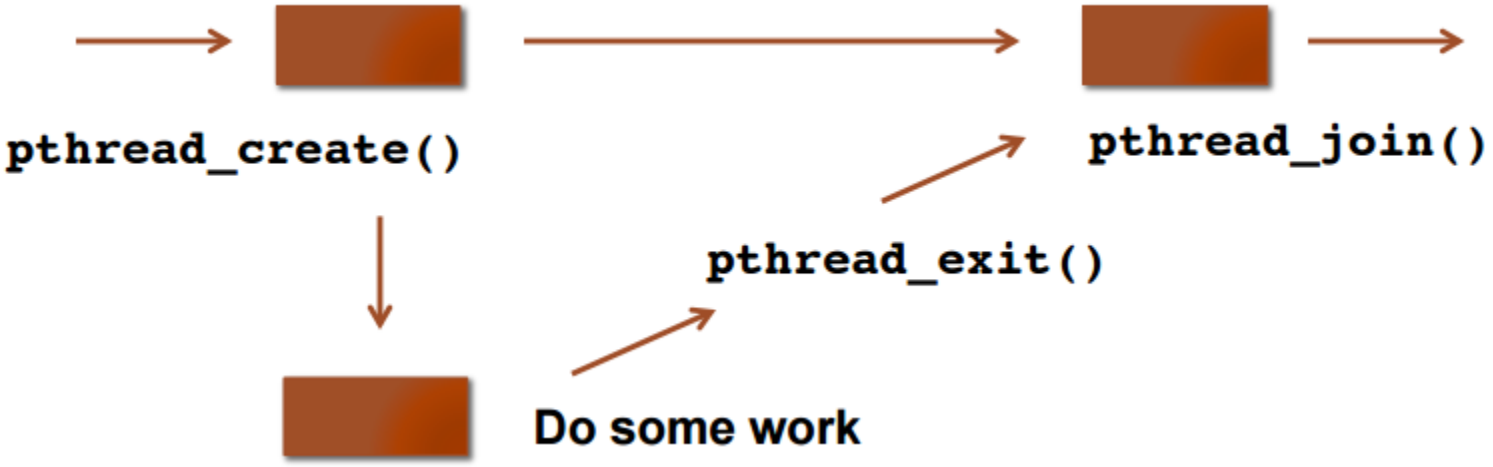
More Example with Return

sum_many_threads.c

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

struct sum_runner_struct {
    long long limit;
    long long answer;
};

// Thread function to generate sum of 0 to N
void* sum_runner(void* arg)
{
    struct sum_runner_struct *arg_struct
                = (struct sum_runner_struct*) arg;

    long long sum = 0;
    for (long long i = 0; i <= arg_struct->limit; i++) {
        sum += i;
    }
    arg_struct->answer = sum;

    pthread_exit(0);
}
```

```
int main(int argc, char **argv)
{
    if (argc < 2) {
        printf("Usage: %s <num 1> <num 2> ... <num-n>₩n", argv[0]);
        exit(-1);
    }
    int num_args = argc - 1;

    struct sum_runner_struct args[num_args];

    // Launch thread
    pthread_t tids[num_args];
    for (int i = 0; i < num_args; i++) {
        args[i].limit = atoll(argv[i + 1]);
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&tids[i], &attr, sum_runner, &args[i]);
    }

    // Wait until thread is done its work
    for (int i = 0; i < num_args; i++) {
        pthread_join(tids[i], NULL);
        printf("Sum for thread %d is %lld₩n", i, args[i].answer);
    }
}
```

Thread Flow



Synchronizing Threads

- Three basic ***synchronization*** primitives
  1. **mutex locks**
  2. **condition variables**
  3. **Semaphores**

- Mutexes and condition variables will handle most of the cases you need in this class
  - but feel free to use semaphores if you like

Mutex Locks

- A Mutex lock is created like a normal variable
  - ***pthread_mutex_p*** *mutex;*

- Mutexes must be initialized before being used
  - a mutex can only be initialized once
  - prototype:
    - int **pthread_mutex_init**(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
      - *mp:* a pointer to the mutex lock to be initialized
      - *mattr:* attributes of the mutex – usually **NULL**

Locking a Mutex

- To insure mutual exclusion to a critical section, a thread should lock a mutex
  - when locking function is called, it does not return until the current thread owns the lock
  - if the mutex is already locked, calling thread blocks
  - if multiple threads try to gain lock at the same time, the return order is based on priority of the threads
    - higher priorities return first
    - no guarantees about ordering between same priority threads
  - prototype:
    - int **pthread_mutex_lock**(pthread_mutex_t *mp);
      - *mp:* mutex to lock

Unlocking a Mutex

- When a thread is finished within the critical section, it needs to release the mutex
  - calling the unlock function releases the lock
  - then, any threads waiting for the lock compete to get it
  - very important to remember to release mutex
  - prototype:
    - int **pthread_mutex_unlock**(pthread_mutex_t *mp);
      - *mp:* mutex to unlock

Example: pth_mutex.c

```
#include <stdio.h>
#include <pthread.h>

#define MAX_SIZE  5
pthread_mutex_t bufLock;
int count;

void producer(char* buf) {
   for(;;) {
     while(count == MAX_SIZE);
     pthread_mutex_lock(&bufLock);
     buf[count] = getChar();
     count++;
     pthread_mutex_unlock(&bufLock);
   }
}
```

```
void consumer(char* buf) {
   for(;;) {
     while(count == 0);
     pthread_mutex_lock(&bufLock);
     useChar(buf[count-1]);
     count--;
     pthread_mutex_unlock(&bufLock);
   }
}

int main() {
   char buffer[MAX_SIZE];
   pthread_t p;
   count = 0;
   pthread_mutex_init(&bufLock, NULL);
   pthread_create(&p, NULL, (void*)producer, &buffer);
   consumer(buffer);
   return 0;
}
```

Condition Variables (CV)

- Notice in the previous example a ***spin-lock*** was used to wait for a condition to be true
  - the buffer to be full or empty
  - spin-locks require CPU time to run
    - ***waste of cycles***

- Condition variables allow a thread to block until a specific condition becomes true
  - recall that a blocked process cannot be run
    - ***doesn't waste CPU cycles***
  - blocked thread goes to wait queue for condition
- When the condition becomes true, some other thread signals the blocked thread(s)

Condition Variables (CV)

- A CV is created like a normal variable
  - ***pthread_cond_t*** *condition;*

- CVs must be initialized before being used
  - a CV can only be initialized once
  - prototype:
    - int **pthread_cond_init**(pthread_cond_t *cv, const pthread_condattr_t *cattr);
      - *cv:* a pointer to the conditon variable to be initialized
      - *cattr:* attributes of the condition variable – usually NULL

Blocking on CV

- A wait call is used to block a thread on a CV
  - puts the thread on a *wait queue* until it gets signaled that the condition is true
  - blocked thread does not compete for CPU
  - the wait call should occur under the protection of a mutex
    - this mutex is automatically **released** by the wait call
    - the mutex is automatically **reclaimed** on return from wait call
- prototype:
  - int **pthread_cond_wait**(pthread_cond_t *cv, pthread_mutex_t *mutex);
    - *cv*: condition variable to block on
    - *mutex:* the mutex to release while waiting

Signaling a Condition

- A signal call is used to "wake up" a single thread waiting on a condition
  - multiple threads may be waiting and there is no guarantee as to which one wakes up first
  - thread to wake up does not actually wake until the lock indicated by the wait call becomes available
  - condition thread was waiting for may not be true when the thread actually gets to run again
    - should always do a wait call inside of a while loop
  - if no waiters on a condition, signaling has no effect
  - prototype:
    - int **pthread_cond_signal**(pthread_cond_t *cv);
      - *cv:* condition variable to signal on

Example: Producer-consumer

```
#include <stdio.h>
#include <pthread.h>

#define MAX_SIZE  5
pthread_mutex_t lock;
pthread_cond_t notFull, notEmpty;
int count;

void producer(char* buf) {
   for(;;) {
      pthreads_mutex_lock(lock);
      while(count == MAX_SIZE)
               pthread_cond_wait(notFull, lock);
      buf[count] = getChar();
      count++;
      pthread_cond_signal(notEmpty);
      pthread_mutex_unlock(lock);
   }
}
```

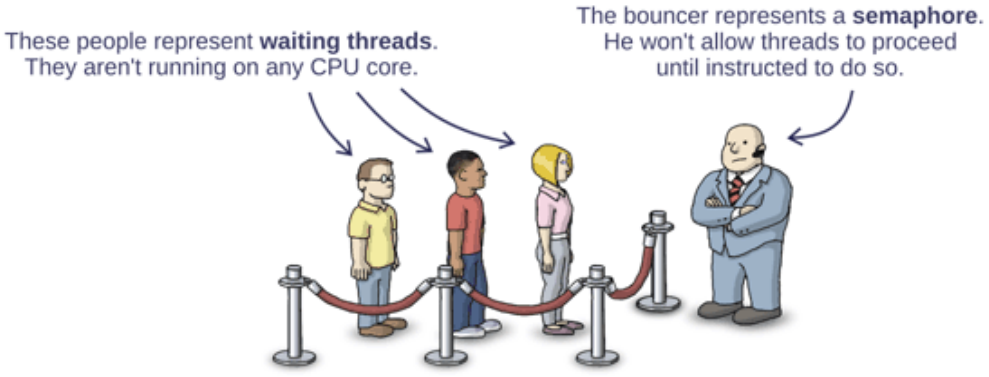
```
void consumer(char* buf) {
   for(;;) {
      pthread_mutex_lock(lock);
      while(count == 0)
         pthread_cond_wait(notEmpty, lock);
      useChar(buf[count-1]);
      count--;
      pthread_cond_signal(notFull);
      pthread_mutex_unlock(lock);
   }
}

int main() {
   char buffer[MAX_SIZE];
   pthread_t p;
   count = 0;
   pthread_mutex_init(&bufLock);
   pthread_cond_init(&notFull);
   pthread_cond_init(&notEmpty);
   pthread_create(&p, NULL, (void*)producer, &buffer);
   consume(&buffer);
   return 0;
}
```

More on Signaling Threads

- The previous example only wakes a single thread
  - not much control over which thread this is
- Perhaps all threads waiting on a condition need to be woken up
- Prototype:
  - int **pthread_cond_broadcast**(pthread_cond_t *cv);
    - *cv:* condition variable to signal all waiters on

Semaphores

- pthreads allows the specific creation of **semaphores**
  - can do increments and decrements of semaphore value
  - semaphore can be initialized to any value
  - thread blocks if semaphore value is less than or equal to zero when a decrement is attempted
  - as soon as semaphore value is greater than zero, one of the blocked threads wakes up and continues
    - no guarantees as to which thread this might be



Creating Semaphores

- Semaphores are created like other variables
  - **sem_t** semaphore;
- Semaphores must be initialized
  - Prototype:
    - int **sem_init**(sem_t *sem, int pshared, unsigned int value);
      - *sem:* the semaphore value to initialize
      - *pshared:* share semaphore across processes – usually 0
      - *value:* the initial value of the semaphore

Decrementing a Semaphore

- Prototype:
  - int **sem_wait**(sem_t *sem);
    - *sem:* semaphore to try and decrement
- If the semaphore value is greater than 0, the *sem_wait* call return immediately
  - otherwise it blocks the calling thread until the value becomes greater than 0

Incrementing a Semaphore

- Prototype:
  - int **sem_post**(sem_t *sem);
    - *sem:* the semaphore to increment
- Increments the value of the semaphore by 1
  - if any threads are blocked on the semaphore, they will be unblocked
- Be careful
  - doing a post to a semaphore always raises its value – even if it shouldn't!

Example

```
#include <stdio.h>
#include <semaphore.h>

#define MAX_SIZE  5
sem_t empty, full;

void producer(char* buf) {
   int in = 0;
   for(;;) {
     sem_wait(&empty);
     buf[in] = getChar();
     in = (in + 1) % MAX_SIZE;
     sem_post(&full);
   }
}
```

```
void consumer(char* buf) {
   int out = 0;
   for(;;) {
     sem_wait(&full);
     useChar(buf[out]);
     out = (out + 1) % MAX_SIZE;
     sem_post(&empty);
   }
}

int main() {
   char buffer[MAX_SIZE];
   pthread_t p;
   sem_init(&empty, 0, MAX_SIZE);
   sem_init(&full, 0, 0);
   pthread_create(&p, NULL, (void*)producer, &buffer);
   consumer(buffer);
   return 0;
}
```

Parting Notes

- Very important to get all the ordering right
  - one simple mistake can lead to problems
    - no progress
    - mutual exclusion violation
- Comparing primitives
  - Using mutual exclusion with CV's is faster than using semaphores
  - Sometimes semaphores are intuitively simpler

Matrix/Vector Multiplication

a_{00}	a_{01}	$\cdots$	$a_{0,n-1}$
a_{10}	a_{11}	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
a_{i0}	a_{i1}	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

x_0
x_1
$\vdots$
x_{n-1}

$=$

y_0
y_1
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
y_{m-1}

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

Matrix/Vector Multiplication

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

π-Calculation

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

π-Calculation

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;
```

```
   return NULL;
}  /* Thread_sum */
```