Introduction to Pthreads

Under the hood of OpenMP on UNIX/Linux

Stéphane Zuckerman

Laboratoire ETIS
Université Paris-Seine, Université de Cergy-Pontoise, ENSEA, CNRS
F95000, Cergy, France

January 16, 2019

Outline

Resources

Resources

- . . . The MAN pages!

Resources

- . . . The MAN pages!
  - Seriously, that's pretty much all you need.
- If you *really* want to read a book about it, you can consult W. Richard Stevens' *Advanced Programming in the UNIX Environment* (Stevens and Rago 2013).
- Another excellent resource (in French) is Christophe Blaess' *Programmation système en C sous Linux: Signaux, processus, threads, IPC et sockets* (Blaess 2011).

Introduction

An Introduction to Multithreading

Processes: a Definition

A process is a set of instructions with its own memory space which is accessed privately. A process is composed of a sequence of instructions (its code), as well as input and output sets (its data). Accessing the memory allocated to a process is in general forbidden unless specific mechanisms are being used, such as inter-process communication functions (IPCs).

Threads: a Definition

A thread is a sequence of code that is part of a process. Consequently, processes contain at least one thread. All threads belonging to the same process share the same address space, and thus can access the same memory locations.

Processes and Threads: the Bare Minimum to Know

Process

- A list of instructions
- Some memory to access with the guarantee it is exclusive to the process
  - A stack to store current values with which to compute
  - A heap to store bigger objects that don't fit in the stack

Thread

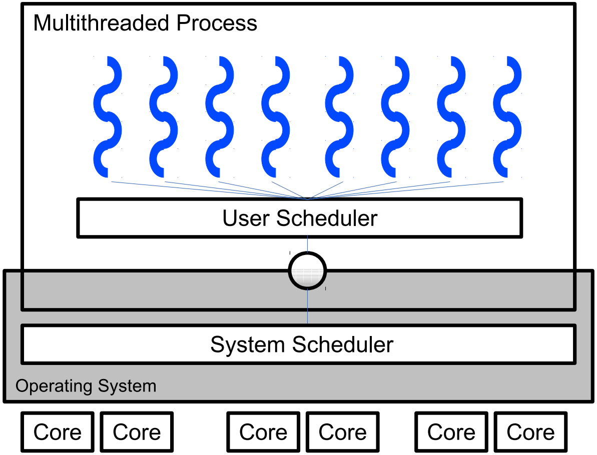
- A list of instructions
- A memory space
  - A stack to store current values with which to compute (private to the thread)
  - Some heap space, shared between threads belonging to the same process

Various Kinds of Multithreading

- User threads
- Kernel threads
- Hybrid ($M \times N$) threads

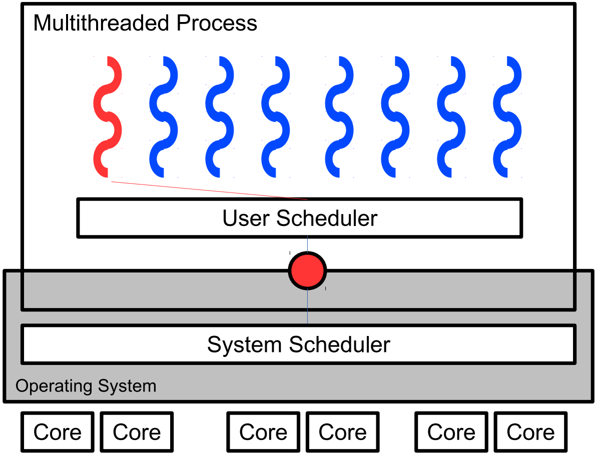
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



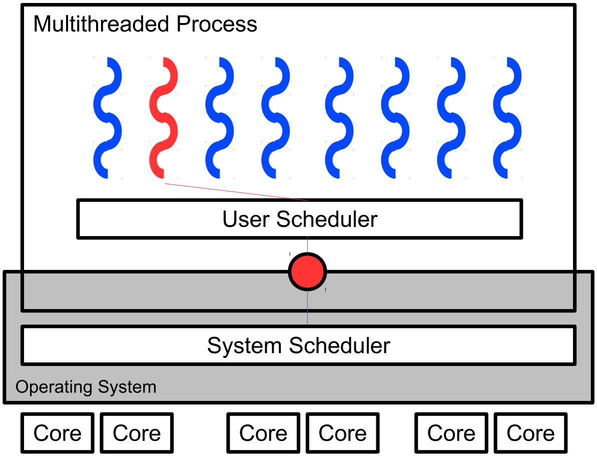
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



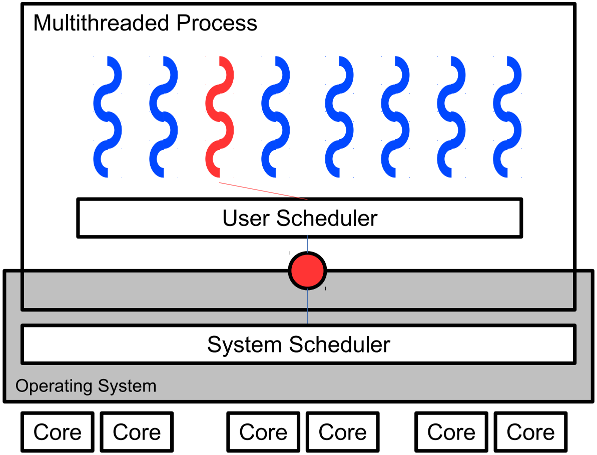
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



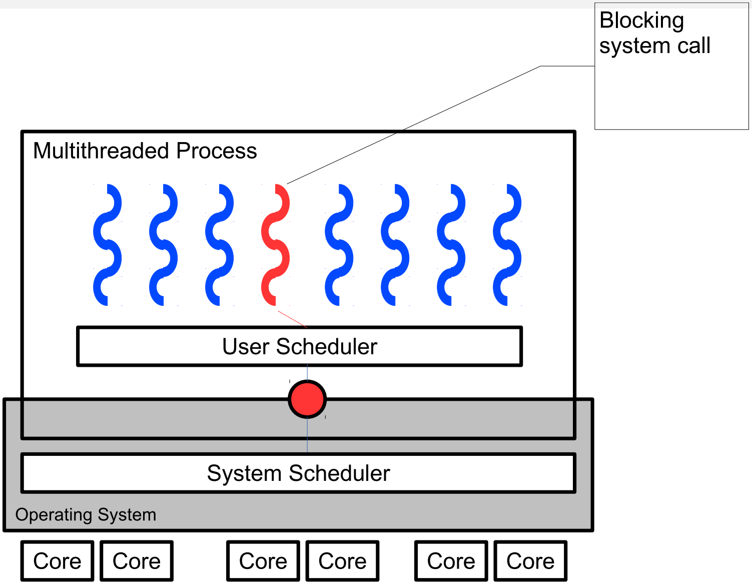
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



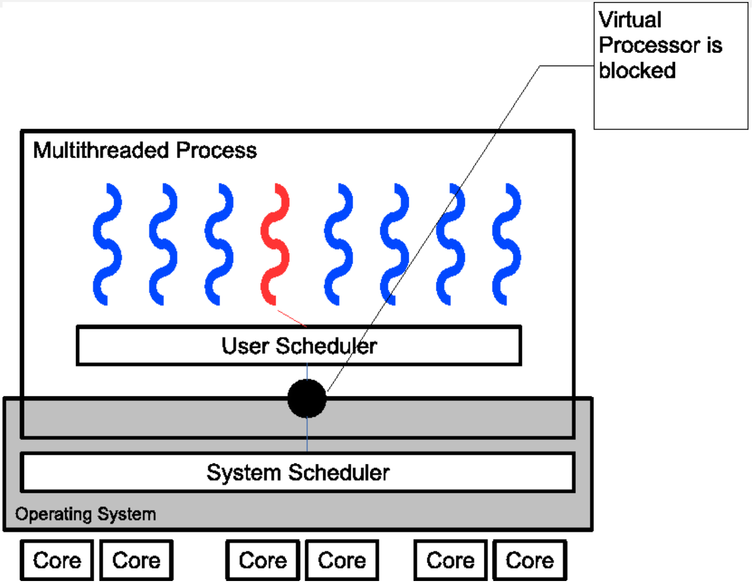
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



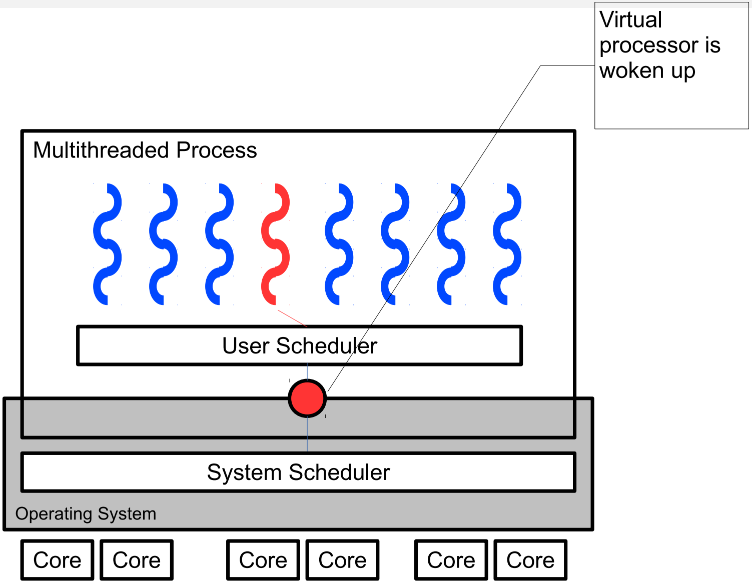
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



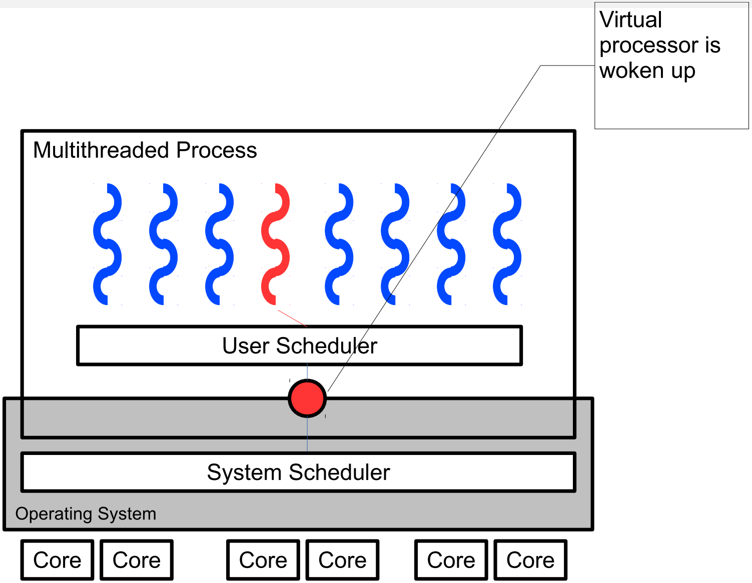
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



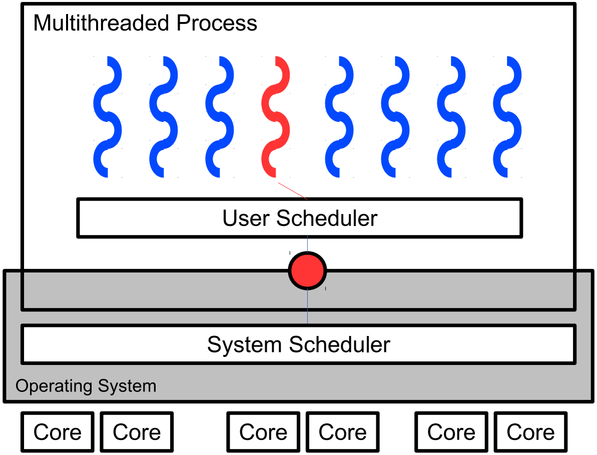
User Thread Libraries

Slides inspired by M. Pérache's multithreading course



User Thread Libraries

Slides inspired by M. Pérache's multithreading course

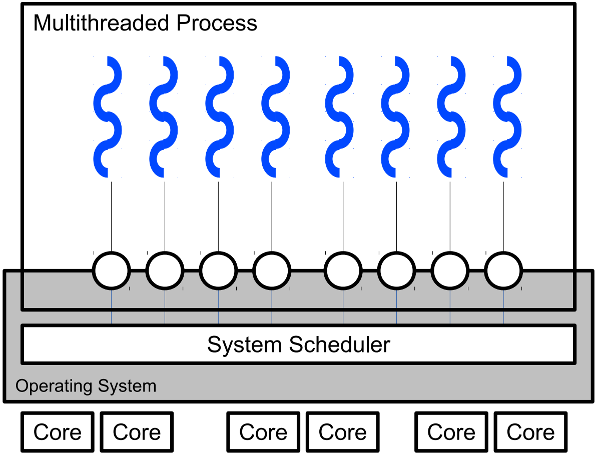


Characteristics of User Threads

- 1 thread per kernel process
- Simple to implement
- Threads libraries were initially implemented this way
- *Very fast:* fully running in user space
- Not really suited to SMP and CMP architectures
- Usually handle system calls badly
- Example of “popular” user thread library: GNU Pth

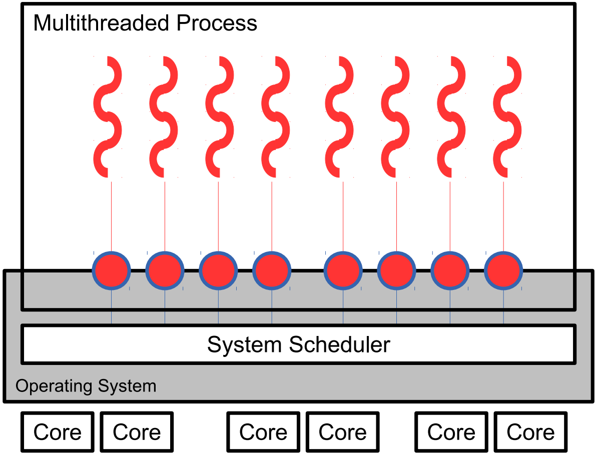
Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course



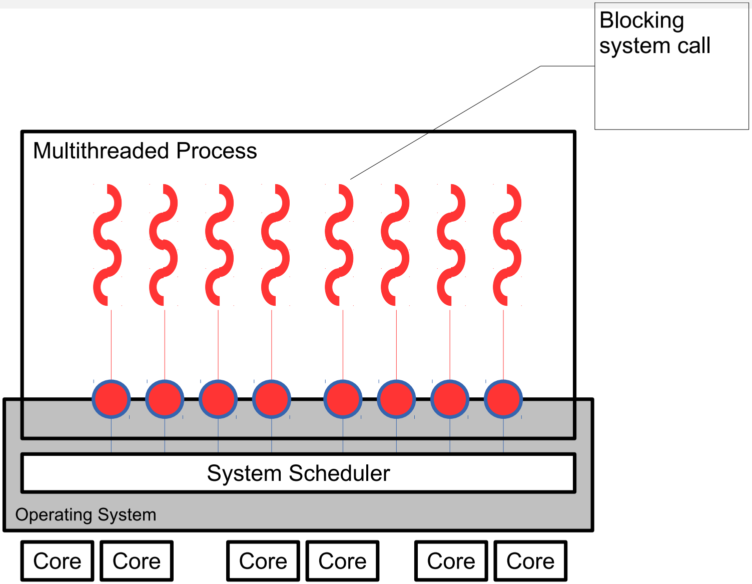
Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course



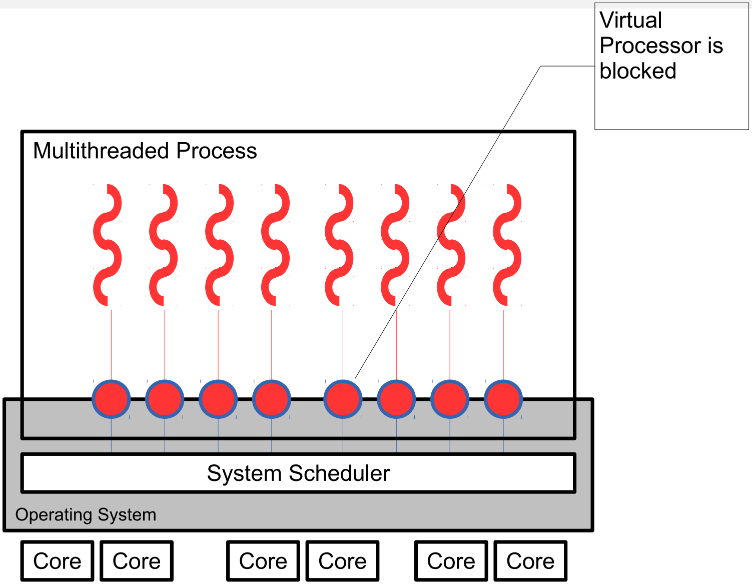
Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course



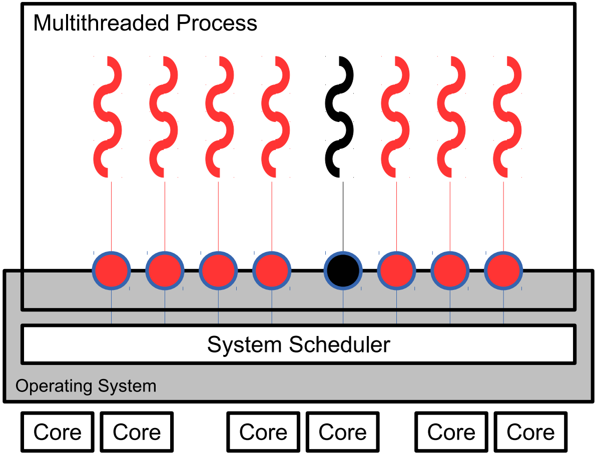
Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course



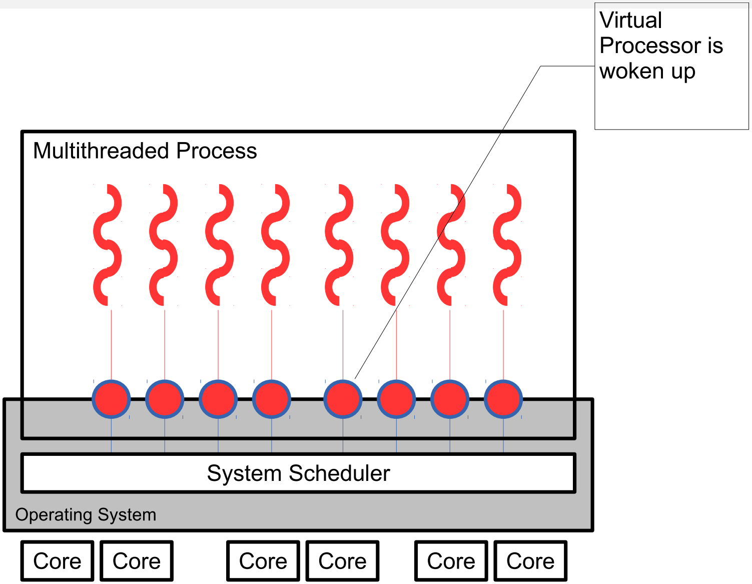
Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course



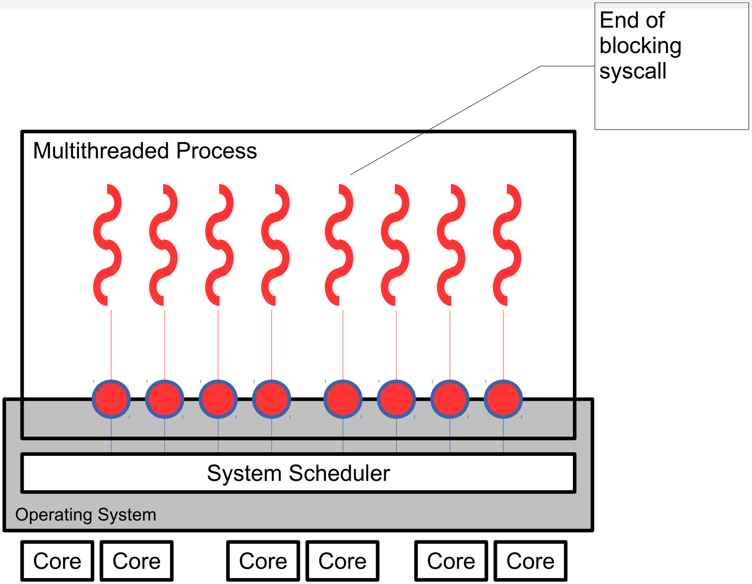
Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course



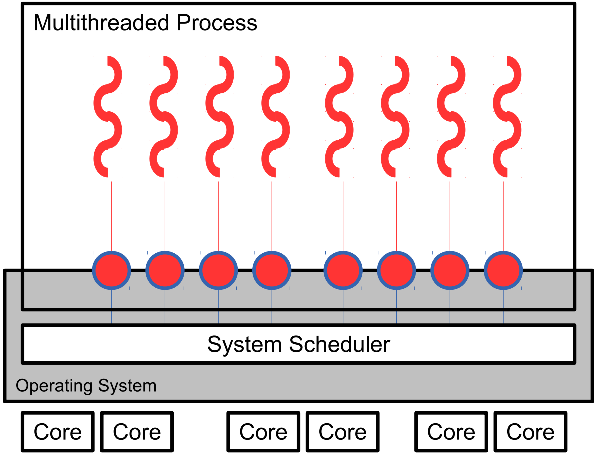
Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course



Kernel Thread Libraries

Slides inspired by M. Pérache's multithreading course

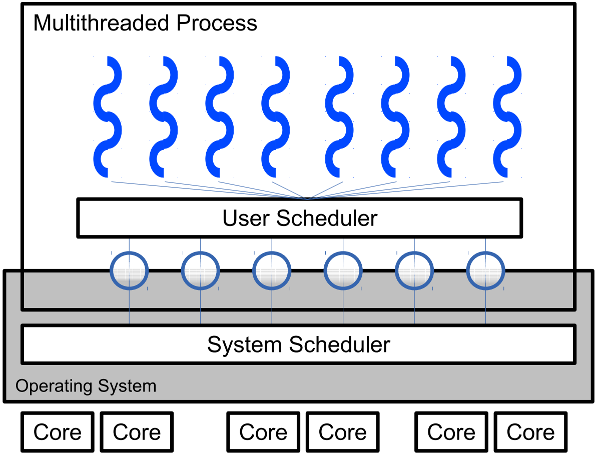


Characteristics of Kernel Threads

- N kernel threads
- Well suited to SMP and CMP architectures
- Handles system calls nicely
- Completely managed at the system level
- Complex to implement
- Slower than user threads (overheads due to entering kernel space)
- Example of “popular” user thread libraries: Windows Threads, LinuxThreads, NPTL

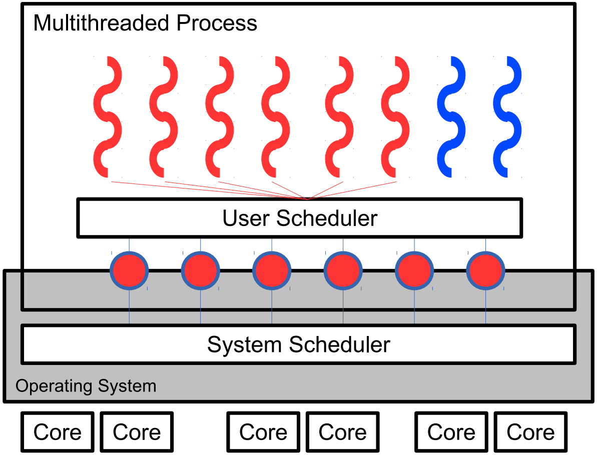
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



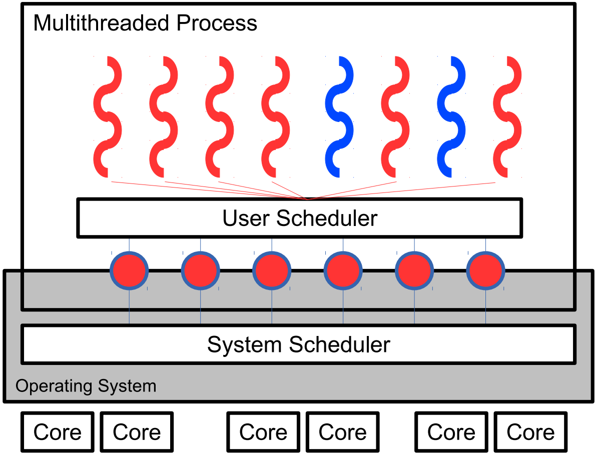
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



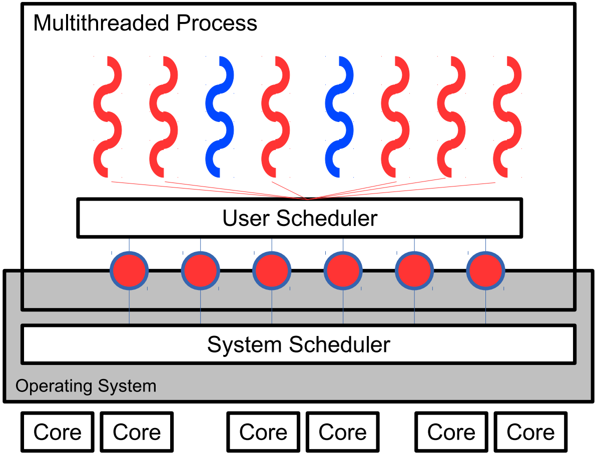
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



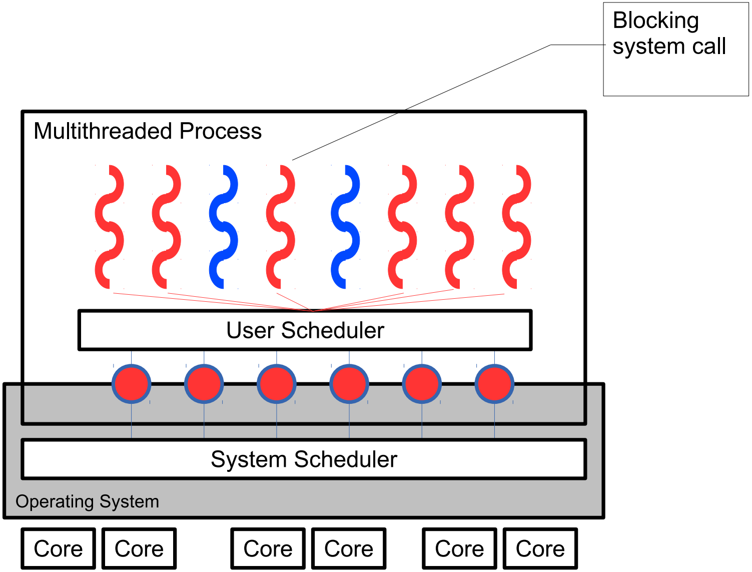
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



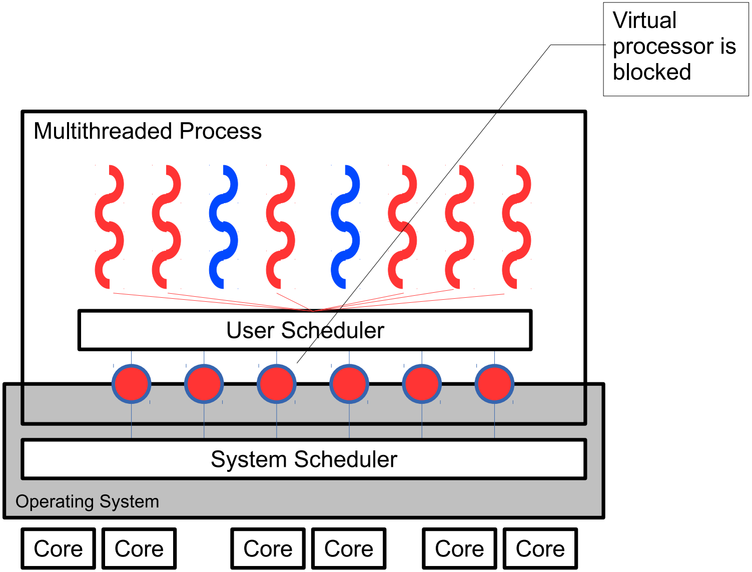
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



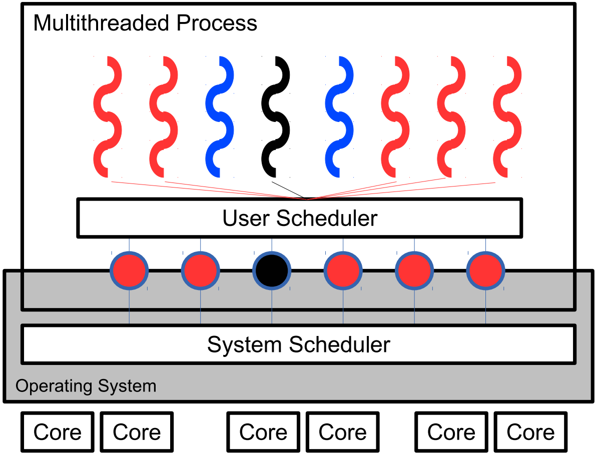
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



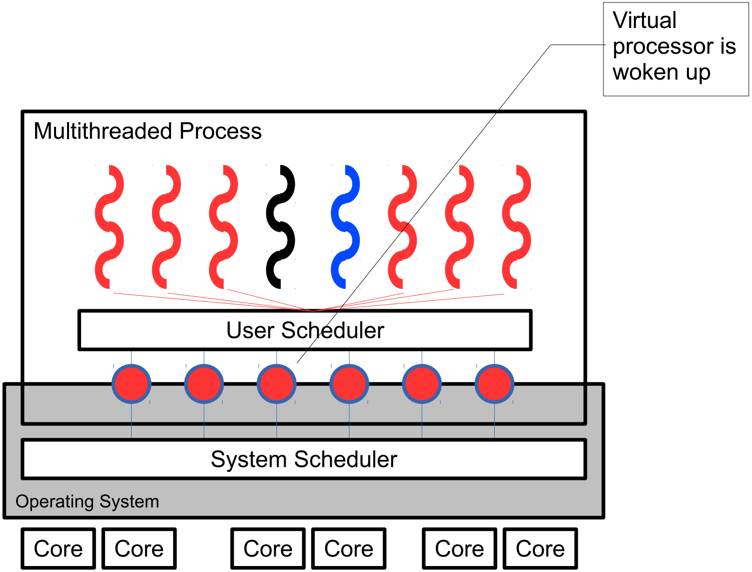
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



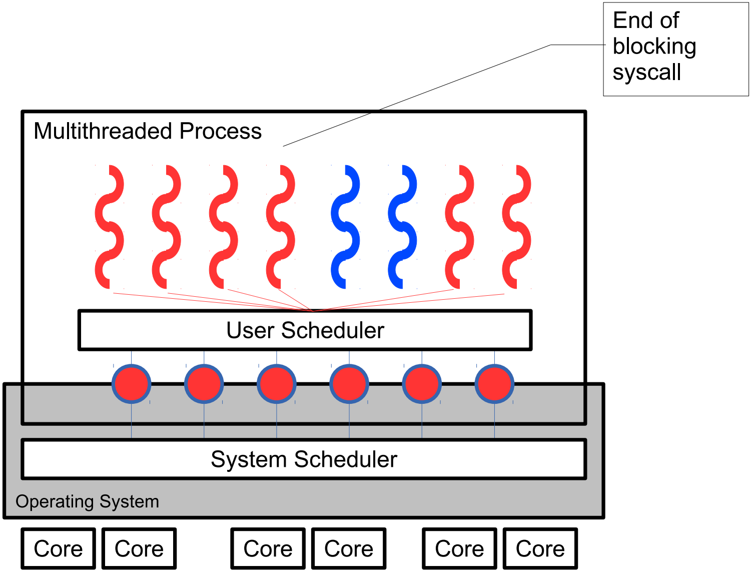
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



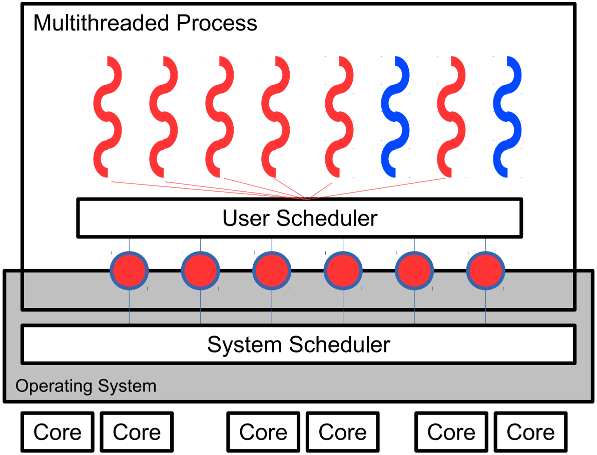
Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course



Hybrid Thread Libraries

Slides inspired by M. Pérache's multithreading course

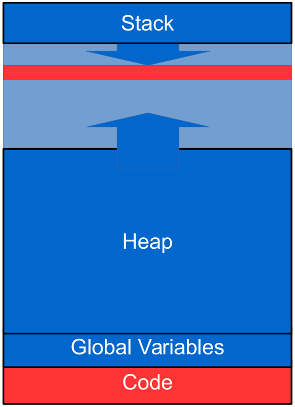


Characteristics of Hybrid Threads

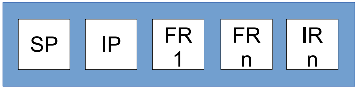
- M kernel threads and N user threads: hybrid threads are also called $M \times N$ threads (or sometimes $M : N$ threads)
- Well suited to SMP and CMP architectures
- Most Complex to implement
- Two schedulers:
  - Kernel Space Scheduler
  - User Space Scheduler
- Efficient
- Handles system calls "well enough" (better than user threads, less than kernel threads)
- Examples of $M \times N$ thread libraries: Solaris' default thread library (until Solaris v10), MPC, most efficient implementations of OpenMP's runtime system.

Process Layout in Memory

An Example Implementation in the Linux OS



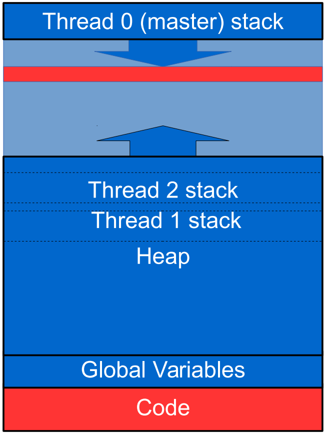
Memory



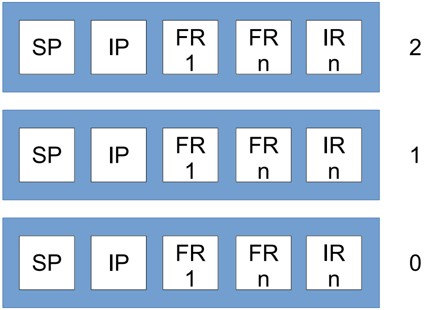
Structure

Thread Layout in Memory

An Example Implementation in the Linux OS



Memory



Structure

A Thread's Characteristics

An Example Implementation in the Linux OS

- All threads share the same address space
- A thread's stack never grows (except for Thread 0)
- A thread's stack is located in the heap (except for Thread 0)
- Global variables are shared by all threads
- Threads communicate directly through memory

PThreads Basics

A Short Introduction to POSIX Threads

- Based on the IEEE POSIX 1003.1 standard
- Any POSIX-compliant system (*i.e.*, UNIX and Linux at the very least) implement the PThread standard:
  - Linux implements PThreads using kernel threads
  - Solaris used to implement PThreads as an $M \times N$ library, but now it is implemented as a kernel thread library
  - OpenBSD used to have a user-level PThread library, but now uses kernel-level one
  - There are a few third-party libraries to provide a source compatibility with PThreads on MS-Windows systems
- Are PThreads lightweight processes?
  - Well, a lightweight process, in essence, is a kernel thread. So if your PThread library is implemented as kernel threads, then yes.
  - In general, the answer is “it depends”

What We Will See in this Tutorial

- How to create and destroy threads
- How to make threads synchronize with each other

PThreads: Basic Types

`pthread_t`	A PThread descriptor and ID
`pthread_mutex_t`	A lock for PThreads
`pthread_cond_t`	A conditional variable. It is necessarily associated with a mutex
`pthread_attr_t`	Descriptor for a PThread's properties (*e.g.*, scheduling hints)
`pthread_mutexattr_t`	Descriptor for mutex' properties (*e.g.*, private to the process or shared between processes; recursive or not; *etc.*)
`pthread_condattr_t`	Descriptor for a condition variable (*e.g.*, private to the process, or shared between processes)

PThreads: Basic Functions
Creation and Destruction

Creation

```
int pthread_create( pthread_t* thd_id, pthread_attr_t* attr,
                    void* (*code)(void*), void* data )
```

Creates a new PThread, using its descriptor reference, the required attributes (or `NULL` for default attributes), a function pointer, and an argument pointer. The function returns 0 if it succeeded, and −1 otherwise. The descriptor is filled and becomes "active" if the call succeeded.

Destruction

```
int pthread_join( pthread_t tid, void** retval )
```

Waits for the PThread with ID `tid` to return, and stores its return value `retval`. If `retval` is `NULL`, the return value is discarded. `pthread_join` returns 0 on success, and −1 otherwise.

Note: Calling `exit(3)` from *any* thread will terminate the whole process, and thus all threads will also terminate!

Usual PThread Calls from Within a Thread

`void pthread_exit( void* retval )`

Exits from the thread calling the function. If `retval` is not `NULL`, it contains the return value of the thread to `pthread_join` (see below).

`pthread_t pthread_self( void )`

Retrieves a thread's own ID.

Note: `pthread_t`, while often implemented as an integer, does not have to be!

A First PThread Example

Hello, World! . . . Headers and worker function

```
#include <stdio.h>   // for snprintf(), fprintf(), printf(), puts()
#include <stdlib.h>  // for exit()
#include <errno.h>   // for errno (duh!)
#include <pthread.h> // for pthread_*
#define MAX_NUM_WORKERS 4UL

typedef struct worker_id_s { unsigned long id } worker_id_t;
void* worker(void* arg)
{
    // Remember, pthread_t objects are descriptors, not just IDs!
    worker_id_t* self = (worker_id_t*) arg;  // Retrieving my ID

    char hello[100]; // To print the message
    int err = snprintf(hello, sizeof(hello),
                       "[%lu]\t␣Hello,␣World!\n", self->id);
    if (err < 0) { perror("snprintf"); exit(errno); }

    puts(hello);
    return arg; // so that the "master" thread
                // knows which thread has returned
}
```

A First PThread Example
Hello, World! . . . main

```
#define ERR_MSG(prefix,...) \
    fprintf(stderr,prefix "␣%lu␣out␣of␣%lu␣threads",__VA_ARGS__)

int main(void) {
  pthread_t    workers    [ MAX_NUM_WORKERS ];
  worker_id_t worker_ids [ MAX_NUM_WORKERS ];
  puts("[main]\tCreating␣workers...\n");
  for (unsigned long i = 0; i < MAX_NUM_WORKERS; ++i) {
    worker_ids[i].id = i;
    if (0 != pthread_create(&workers[i], NULL, worker, &worker_ids[i]))
      { ERR_MSG("Could␣not␣create␣thread", i, MAX_NUM_WORKERS);
        exit(errno); }
  }
  puts("[main]\tJoining␣the␣workers...\n");
  for (unsigned long i = 0; i < MAX_NUM_WORKERS; ++i) {
    worker_id_t* wid = (worker_id_t*) retval;
    if (0 != pthread_join(workers[i], (void**) &retval))
      ERR_MSG("Could␣not␣join␣thread", i, MAX_NUM_WORKERS);
     else
      printf("[main]\tWorker␣N.%lu␣has␣returned!\n", wid->id);
  }
  return 0;}
```

A First PThread Example

Hello, World! . . . Output

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c hello.c
gcc -o hello hello.o -lpthread
```

. . . Don't forget to link with the PTHREAD library!

. . . And the output:

Output of `./hello`

```
[main]  Creating workers...
[0]  Hello, World!
[main]  Joining the workers...
[2]  Hello, World!
[main]  Worker N.0 has returned!
[1]  Hello, World!
[3]  Hello, World!
[main]  Worker N.1 has returned!
[main]  Worker N.2 has returned!
[main]  Worker N.3 has returned!
```

Incrementing a Global Counter
Naïve Code

```
#ifndef BAD_GLOBAL_SUM_H
#define BAD_GLOBAL_SUM_H

#include <stdio.h>
#include <stdlib.h>
#include "utils.h"

typedef struct bad_global_sum_s {
    unsigned long *value;
} bad_global_sum_t;

#endif // BAD_GLOBAL_SUM_H
```

Figure: bad_global_sum.h

Incrementing a Global Counter
Naïve Code (2)

```
#include "bad_global_sum.h"
#define MAX_NUM_WORKERS 20UL
typedef unsigned long ulong_t;

void* bad_sum(void* frame) {
    bad_global_sum_t* pgs = (bad_global_sum_t*) frame;
    ++*pgs->value;
    return NULL;
}

int main(void) {
    pthread_t         threads [ MAX_NUM_WORKERS ];
    bad_global_sum_t frames   [ MAX_NUM_WORKERS ];
    ulong_t counter = 0;

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i) {
        frames[i].value = &counter;
        spthread_create(&threads[i],NULL,bad_sum,&frames[i]);
    }

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)
        spthread_join(threads[i],NULL);

    printf("%lu␣threads␣were␣running.␣Sum␣final␣value:␣%lu\n", MAX_NUM_WORKERS, counter);

    return 0;
}
```

Incrementing a Global Counter
Naïve Code (3)

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c bad_sum_pthreads.c
gcc -o badsum bad_sum_pthreads.o -lpthread
```

…Don't forget to link with the PThread library!

Incrementing a Global Counter
Naïve Code (3)

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c bad_sum_pthreads.c
gcc -o badsum bad_sum_pthreads.o -lpthread
```

. . . Don't forget to link with the PTHREAD library!

Output of `./badsum`

```
szuckerm@evans201g:bad$ ./badsum
20 threads were running. Sum final value: 20
```

Hey, it's working!

Incrementing a Global Counter
Naïve Code (3)

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c bad_sum_pthreads.c
gcc -o badsum bad_sum_pthreads.o -lpthread
```

. . . Don't forget to link with the PThread library!

Output of `./badsum`

```
szuckerm@evans201g:bad$ ./badsum
20 threads were running. Sum final value: 20
```

Hey, it's working!

Multiple executions of `./badsum`

```
szuckerm@evans201g:bad$ (for i in `seq 100`;do ./badsum ;done)|uniq
20 threads were running. Sum final value: 20
20 threads were running. Sum final value: 19
20 threads were running. Sum final value: 20
20 threads were running. Sum final value: 19
20 threads were running. Sum final value: 20
```

Incrementing a Global Counter
Naïve Code (3)

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c bad_sum_pthreads.c
gcc -o badsum bad_sum_pthreads.o -lpthread
```

. . . Don't forget to link with the PThread library!

Output of `./badsum`

```
szuckerm@evans201g:bad$ ./badsum
20 threads were running. Sum final value: 20
```

Hey, it's working!

Multiple executions of `./badsum`

```
szuckerm@evans201g:bad$ (for i in ‘seq 100‘;do ./badsum ;done)|uniq
20 threads were running. Sum final value: 20
20 threads were running. Sum final value: 19
20 threads were running. Sum final value: 20
20 threads were running. Sum final value: 19
20 threads were running. Sum final value: 20
```

Waiiiiit a minute. . .

Incrementing a Global Counter

Fixing the Implementation

Mutexes

A MUTual EXclusive object (or mutex) is a synchronization object which is either owned by a single thread, or by no-one. It is the basic block to create critical sections.

```c
#ifndef GLOBAL_SUM_H
#define GLOBAL_SUM_H

#include <stdio.h>
#include <stdlib.h>
#include "utils.h"

typedef struct global_sum_s {
    unsigned long   *value;
    pthread_mutex_t *lock;
} global_sum_t;

#endif // GLOBAL_SUM_H
```

Incrementing a Global Counter
Fixing the Implementation (2)

```
#include "global_sum.h"
#define MAX_NUM_WORKERS 20UL
typedef unsigned long ulong_t;

void* sum(void* frame) {
    global_sum_t* gs = (global_sum_t*) frame;
    spthread_mutex_lock ( gs->lock );  /* Critical section starts here */
    ++*gs->value;
    spthread_mutex_unlock ( gs->lock ); /* Critical section ends here */
    return NULL;
}

int main(void) {
    pthread_t        threads [ MAX_NUM_WORKERS ];
    global_sum_t     frames  [ MAX_NUM_WORKERS ];
    ulong_t          counter = 0;
    pthread_mutex_t m        = PTHREAD_MUTEX_INITIALIZER;

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i) {
        frames[i] = (global_sum_t){ .value = &counter, .lock = &m };
        spthread_create(&threads[i],NULL,sum,&frames[i]);
    }

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)
        spthread_join(threads[i],NULL);

    printf("%lu␣threads␣were␣running.␣Sum␣final␣value:␣%lu\n", MAX_NUM_WORKERS, counter);
    return 0;
}
```

Figure: sum_pthreads.c

Incrementing a Global Counter
Fixing the Implementation (3)

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c sum_pthreads.c
gcc -o sum sum_pthreads.o -lpthread
```

. . . Don't forget to link with the PTHREAD library!

Multiple executions of ./sum

```
szuckerm@evans201g:good$ (for i in ‘seq 100‘;do ./sum ;done)|uniq
20 threads were running. Sum final value: 20
```

Incrementing a Global Counter
Fixing the Implementation (3)

Compilation Process

```
gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c sum_pthreads.c
gcc -o sum sum_pthreads.o -lpthread
```

…Don't forget to link with the PThread library!

Multiple executions of ./sum

```
szuckerm@evans201g:good$ (for i in ‘seq 100‘;do ./sum ;done)|uniq
20 threads were running. Sum final value: 20
```

Fixed!

Reacting on Specific Events I
Condition Variables

Condition variables

Condition variables are used when threads are waiting on a specific event. When the event occurs, the code where it the event was realized *signals* a condition variable, either to wake up one of the threads waiting on the event, or all of them.

Examples of Events to Be Worth Signaling

- Availability of a resource, *e.g.*:
  - A file descriptor for a network connection,
  - A file descriptor for accessing (reading or writing) a regular file,
  - Any device handle, really
- A specific input provided by the user (string provided by the user, *etc.*)
- *etc.*

Reacting on Specific Events II
Condition Variables

High-Level Explanation: Waiting on a Condition

1. A condition variable is always associated with a mutex
2. To wait on an event, a thread must first acquire the mutex, then
3. Call `int pthread_cond_wait( pthread_cond_t* cond, pthread_mutex_t* mutex )`
4. If the call succeeds, then the thread *releases* the mutex
5. When the condition variable is signaled, *if* the thread which was "asleep" is re-awakened, the system first returns ownership of the mutex back to it

Reacting on Specific Events III
Condition Variables

High-Level Explanation: Signaling an Event Has Occurred

There are two function calls to perform this function:

- `int pthread_cond_signal( pthread_cond_t* cond )`
  - To signal a single thread that the event has occurred. Note: there is no guarantee as to *which* thread will wake
- `int pthread_cond_broadcast( pthread_cond_t* cond )`
  - To signal all threads that the event has occurred.

Reacting on Specific Events
Condition Variables

```
#ifndef BARRIER_H
#define BARRIER_H

#define SET_BARRIER_MSG(...) \
    snprintf(buffer, sizeof(buffer), __VA_ARGS__)
#define NOT_LAST_TO_REACH \
    "[%lu]\tI'm␣NOT␣the␣last␣one␣to␣reach␣the␣barrier!"
#define LAST_TO_REACH      \
    "[%lu]\tI␣am␣the␣last␣to␣reach␣the␣barrier!␣Waking␣up␣the␣others."

typedef struct barrier_s {
    pthread_mutex_t *lock;
    pthread_cond_t  *cond;
    ulong_t         *count;
} barrier_t;

typedef struct context_s {
    barrier_t* barrier;
    ulong_t    id;
} context_t;

#endif // BARRIER_H
```

Figure: barrier.h

Reacting on Specific Events
Condition Variables (2)

```c
#include "barrier.h"
void* worker(void* frame);

int main(void) {
    pthread_t        threads  [ MAX_NUM_WORKERS ];
    context_t        contexts [ MAX_NUM_WORKERS ];
    pthread_mutex_t m       = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t   cond   = PTHREAD_COND_INITIALIZER;
    ulong_t          count  = MAX_NUM_WORKERS;
    barrier_t barrier = {.lock = &m, .cond = &cond, .count = &count};

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)  {
        contexts[i] = (context_t){ .barrier = &barrier, .id = i };
        spthread_create(&threads[i],NULL,worker,&contexts[i]);
    }

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)
        spthread_join(threads[i],NULL);

    return 0;
}
```

Reacting on Specific Events
Condition Variables (3)

```
#include "barrier.h"

void* worker(void* frame) {
    char        buffer[81];
    context_t* c = (context_t*) frame;
    printf("[%lu]\tReaching␣the␣barrier...\n",c->id);
    spthread_mutex_lock ( c->barrier->lock );
    --*c->barrier->count;
    if (*c->barrier->count > 0) {
        SET_BARRIER_MSG(NOT_LAST_TO_REACH, c->id);
        spthread_cond_wait ( c->barrier->cond, c->barrier->lock );
    } else {
        SET_BARRIER_MSG(LAST_TO_REACH, c->id);
    }
    puts(buffer);

    spthread_mutex_unlock ( c->barrier->lock );
    pthread_cond_broadcast( c->barrier->cond );
    printf("[%lu]\tAfter␣the␣barrier\n", c->id);

    return NULL;
}
```

Figure: barrier.c

Reacting on Specific Events

Condition Variables (4)

```
szuckerm@evans201g:condvar$ gcc -Wall -Wextra -pedantic -Werror -O3 -std=c99 -c barrier.c
szuckerm@evans201g:condvar$ gcc -o barrier barrier.o -lpthread
szuckerm@evans201g:condvar$ ./barrier
[0] Reaching the barrier...
[2] Reaching the barrier...
[1] Reaching the barrier...
[3] Reaching the barrier...
[4] Reaching the barrier...
[5] Reaching the barrier...
[7] Reaching the barrier...
[6] Reaching the barrier...
[6] I am the last to reach the barrier! Waking up the others.
[6] After the barrier
[0] I'm NOT the last one to reach the barrier!
[0] After the barrier
[1] I'm NOT the last one to reach the barrier!
[1] After the barrier
[2] I'm NOT the last one to reach the barrier!
[2] After the barrier
[3] I'm NOT the last one to reach the barrier!
[3] After the barrier
[4] I'm NOT the last one to reach the barrier!
[4] After the barrier
[5] I'm NOT the last one to reach the barrier!
[5] After the barrier
[7] I'm NOT the last one to reach the barrier!
[7] After the barrier
```

Creating Barriers More Easily

- "Hey, barriers are nice! I wish I could have a more practical construct, though."

Creating Barriers More Easily

- "Hey, barriers are nice! I wish I could have a more practical construct, though."
- . . . Well actually, did I tell you about PThread barriers?

pthread_barrier_t and its associated functions

- `int pthread_barrier_init( pthread_barrier_t restrict* barrier, const pthread_barrierattr_t *restrict attr, unsigned count )`
- `int pthread_barrier_destroy( pthread_barrier_t restrict* barrier )`
- `int pthread_barrier_wait( pthread_barrier_t restrict* barrier )`

Updated Barrier Program

Using PThread Barriers

```
#ifndef BARRIER_H
#define BARRIER_H
#include "utils.h"
#define MAX_NUM_WORKERS 8UL
typedef unsigned long ulong_t;
typedef struct context_s {
    pthread_barrier_t* barrier;
    ulong_t    id;
} context_t;

#endif // BARRIER_H
```

Figure: pth_barrier.h

```
#include "barrier.h"

void* worker(void* frame) {
    context_t* c = (context_t*) frame;
    printf("[%lu]\tReaching␣the␣barrier...\n",c->id);
    spthread_barrier_wait( c->barrier );
    printf("[%lu]\tAfter␣the␣barrier\n", c->id);
    return NULL;
}
```

Figure: pth_barrier.c (1)

Updated Barrier Program

Using PThread Barriers (2)

```
#include "barrier.h"

int main(void) {
    pthread_t          threads   [ MAX_NUM_WORKERS ];
    context_t          contexts  [ MAX_NUM_WORKERS ];
    ulong_t            count = MAX_NUM_WORKERS;
    pthread_barrier_t  barrier;

    spthread_barrier_init(&barrier,NULL,count);

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)  {
        contexts[i] = (context_t){ .barrier = &barrier, .id = i };
        spthread_create(&threads[i],NULL,worker,&contexts[i]);
    }

    for (ulong_t i = 0; i < MAX_NUM_WORKERS; ++i)
        spthread_join(threads[i],NULL);

    spthread_barrier_destroy(&barrier);

    return 0;
}
```

Figure: pth_barrier.c (2)

Where to Learn More

Learning More About Multi-Threading and PThreads

UNIVERSITÉ de Cergy-Pontoise

Books (from most theoretical to most practical)

- Tanenbaum 2007
- Herlihy and Shavit 2008
- Bovet and Cesati 2002
- Stevens and Rago 2013

Internet Resources

- "POSIX Threads Programmings" at `https://computing.llnl.gov/tutorials/pthreads/`
- "Multithreaded Programming (POSIX pthreads Tutorial)" at `http://randu.org/tutorials/threads/`

Food for Thoughts

- Sutter 2005 (available at `http://www.gotw.ca/publications/concurrency-ddj.htm`)
- Lee 2006 (available at `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf`)
- Boehm 2005 (available at `www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf`)

Bibliography

References I

References

Bovet, Daniel and Marco Cesati (2002). *Understanding the Linux Kernel, Second Edition*. Ed. by Andy Oram. 2nd ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc. ISBN: 0596002130.

Boehm, Hans-J. (2005). "Threads Cannot Be Implemented As a Library". In: *SIGPLAN Not.* 40.6, pp. 261–268. ISSN: 0362-1340. DOI: 10.1145/1064978.1065042. URL: http://doi.acm.org/10.1145/1064978.1065042.

Sutter, Herb (2005). "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3.

References II

Lee, Edward A. (2006). "The Problem with Threads". In: *Computer* 39.5, pp. 33–42. ISSN: 0018-9162. DOI: 10.1109/MC.2006.180. URL: http://dx.doi.org/10.1109/MC.2006.180.

Tanenbaum, Andrew S. (2007). *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press. ISBN: 9780136006633.

Herlihy, Maurice and Nir Shavit (2008). *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0123705916, 9780123705914.

Blaess, Christophe (2011). *Programmation système en c sous linux : signaux, processus, threads, ipc et sockets*. 3rd. Eyrolles. ISBN: 9782212085549.

Stevens, Richard W. and Steven A. Rago (2013). *Advanced Programming in the UNIX Environment, 3rd Edition*. Indianapolis, IN, USA: Addison-Wesley Professional. ISBN: 0321637739, 9780321637734.