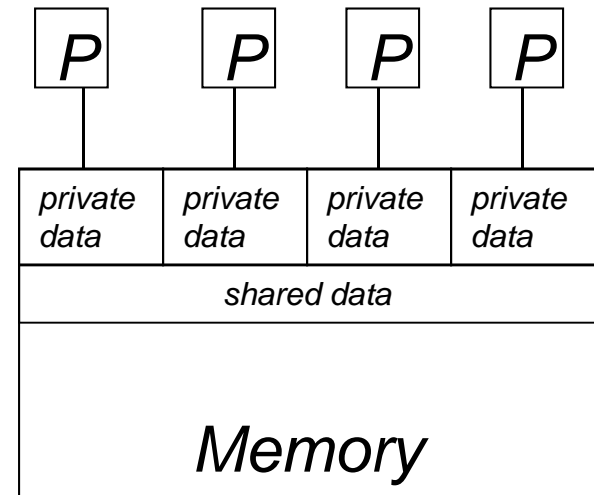


Programming with Threads

Outline

- Threads & Processes
- PThreads

*simple HW model
to get us started*



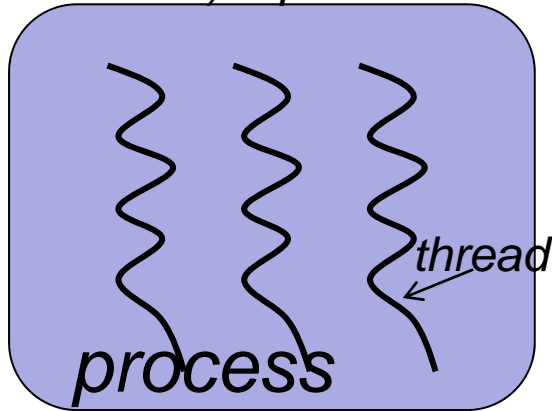
*Critical system issue:
Does the OS know about
threads or not?*

Threads

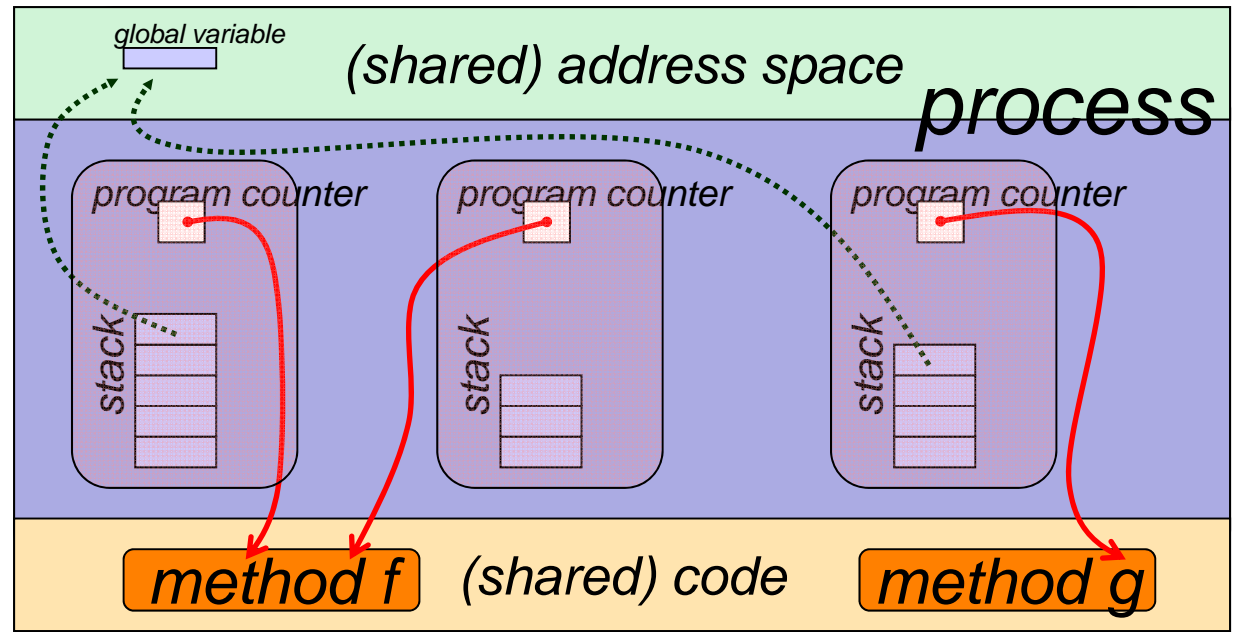
- **Motivation:** Threads came about because of the need to write concurrent applications, that is the need for “tasks” that share memory.
 - Since the advent of the multiprocess OS it has been possible to write processes that share memory, but it is neither the right level of abstraction nor as efficient as it could be
 - *In this course, our view is a bit different. We are looking for independent instruction streams to run on independent cores that are easy to manipulate and have low overhead*
- **Intuition:** Threads are “processes” that share a single address space
- **Vocabulary:** Threads are often called “lightweight processes”
 - N processes have N page tables, N address spaces, N PIDs, ...
 - N threads have 1 page table, 1 address space, 1 PID
- **Essentials:** Things that threads do not share: *program counter & stack*
 - N threads have N program counters
 - N threads have N stacks
- **Observation:** Therefore, multiple threads can be executing different executions of the program “at the same time,” and be following completely different calling sequences
- **OS view:** A thread is an independent stream of instructions that can be scheduled to run by the OS.
 - >> *Not always – see comment on 1st page*

Threads in a Process

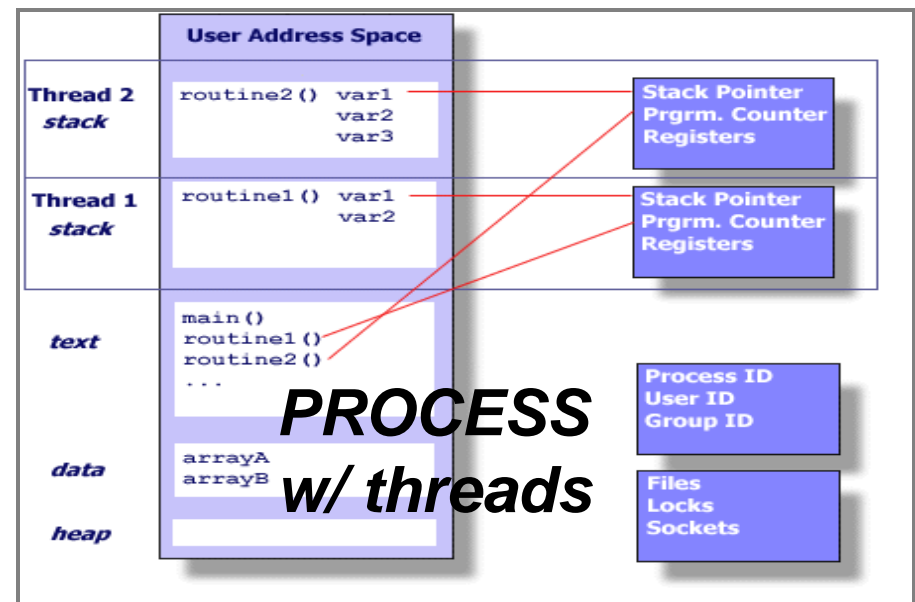
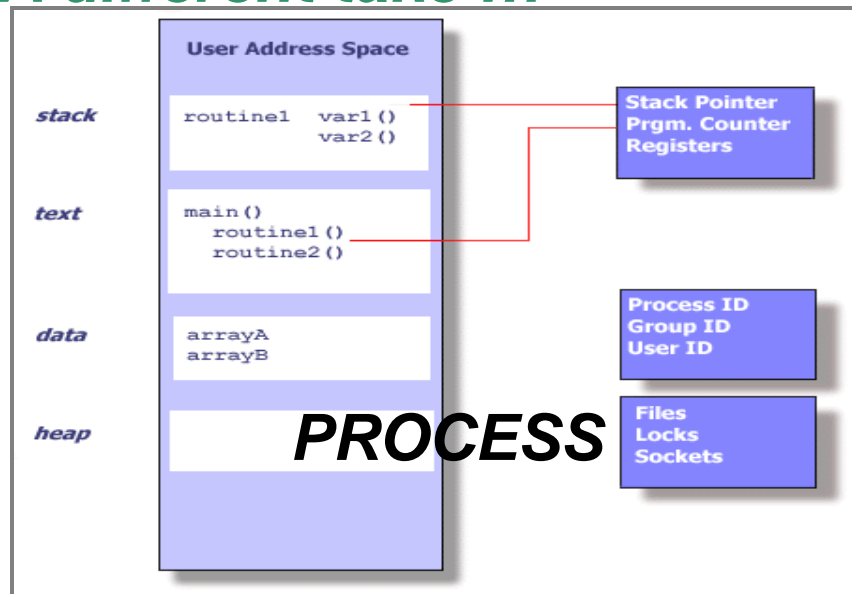
Typical (but mostly useless) representation



A much more useful representation



A different take ...



Threads vs. Processes

- Sharing memory with threads is straightforward
 - They were designed for this
- Threads are much “cheaper” than processes
 - Less time to start a thread
 - Less time to switch between threads
- Threads are more memory-constrained than processes
 - Simply because many threads can live in a process
- Threads do not benefit from memory protection
 - from each other, that is!
 - Can cause nasty bugs when a thread stomps over another thread’s memory (like in the days before virtual memory!)
- We will see that thread synchronization is in principle simple but also fraught with peril

Example multithread programs?

We will be considering mostly pretty straightforward decompositions, such as we saw with SOR. But there are lots of other possibilities →

- Computer games
 - each thread controls the movement of an object.
- Scientific simulations
 - Hurricane movement simulation: each thread simulates the hurricane in a small domain.
 - Molecular dynamics: each thread simulates a subset of particles.
 -
- Simulators
 - Each thread handles an event
- Web server
 - Each thread handles a connection.
-

Threads...

- Exist within processes
- Die if the process dies
- Use process resources
- Duplicate only the essential resources for OS (*or RTS*) to schedule them independently
- Each thread (generally) maintains
 - Stack
 - Registers
 - Scheduling properties (e.g. priority)
 - Set of pending and blocked signals (to allow different react differently to signals)
 - Thread specific data

Variables, shared and local

- Threads keep independent stack pointers
 - *Stacks are not shared among threads*
 - *Automatic variables (local declarations) are therefore only visible to the thread that declares them.*
- Global variables are visible to all threads in a process
- Any data can be made visible to another thread by giving that other thread a pointer to the data.

Advantages of Threads

- Light-weight
 - Lower overhead for thread creation
 - Lower Context Switching Overhead
 - Fewer OS resources

Times in microseconds

<i>Platform</i>	<i>fork</i>			<i>pthread_create</i>		
	<i>real</i>	<i>user</i>	<i>sys</i>	<i>real</i>	<i>user</i>	<i>sys</i>
<i>AMD 2.4 GHz Opteron (8cpus/node)</i>	41.1	60.1	9.0	0.7	0.2	0.4
<i>IBM 1.9 GHz POWER5 p5-575 (8cpus/node)</i>	64.2	30.8	27.7	1.8	0.7	1.1
<i>IBM 1.5 GHz POWER4 (8cpus/node)</i>	104.1	48.6	47.2	2.0	1.0	1.5
<i>INTEL 2.4 GHz Xeon (2 cpus/node)</i>	55.0	1.5	20.8	1.6	0.7	0.9
<i>INTEL 1.4 GHz Itanium2 (4 cpus/node)</i>	54.5	1.1	22.2	2.0	1.3	0.7

Advantages of Threads, cont.

- Shared State
 - Don't need IPC-like mechanism to communicate between threads of same process

Disadvantages of Threads

- Shared State!
 - Global variables are shared between threads. Accidental changes can be fatal.
- Many library functions are not **thread-safe**
 - Library Functions that return pointers to static internal memory. E.g. **gethostbyname()**
- Lack of robustness
 - Crash in one thread will crash the entire process.

Pthreads: POSIX Threads

- Pthreads is a standard set of C library functions for multithreaded programming
 - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
 - Thread management: create, exit, detach, join, . . .
 - Thread cancellation
 - Mutex locks: init, destroy, lock, unlock, . . .
 - Condition variables: init, destroy, wait, timed wait, . . .
 - . . .
- Programs must include the file **pthread.h**
- Programs must be linked with the pthread library (**-pthread**)

pthread_s -- Basic Datatypes

(variables you declare in your pthread code)

pthread_t *var*;

A variable that will be used as an **ID for a thread**

pthread_attr_t *var*;

A variable that defines the thread behavior (i.e. scheduling, priority, stack size, scope, etc.)

pthread_mutex_t *var*;

The definition of a **Mutually Exclusive Lock** in Pthreads

pthread_mutexattr_t *var*;

Defines the mutex behavior. Either PTHREAD_PROCESS_PRIVATE (the default) and PTHREAD_PROCESS_SHARED

pthread_cond_t *var*;

The definition of a Conditional variable in Pthreads

pthread_condattr_t *var*;

Defines the conditional variable behavior. Either PTHREAD_PROCESS_PRIVATE (the default) and PTHREAD_PROCESS_SHARED

Thread creation

Return a non zero value in success

```
→ int pthread_create(pthread_t *thr,  
                     const pthread_attr_t *attr,  
                     void *(*start_routine)(void),  
                     void *arg)
```

pthread_t *thr

Will contain the newly created thread's id. Must be *passed by reference*

const pthread_attr_t *attr

Give the attributes that this thread will have. Use **NULL** as default

→ void *(*start_routine)(void)

The name of the function that the thread will run. Function must have a **void pointer** as its return and parameter values

→ void *arg

The argument for the function that will be the body of the Pthread

Pointers of the type **void** can reference **ANY** type of data, but **MUST FIRST BE CAST** to the data type being read or written.



Useful functions

```
pthread_t pthread_self(void);
```

Return the id of the calling thread. Returns a `pthread_t` type which is usually an integer type variable

OpenMP Counterpart

```
int omp_get_thread_num(void);
```

```
void pthread_exit(void *arg);
```

Used to terminate a Pthread neatly → return value put in *arg*



Simple Thread Creation & Execution Example

Example 1:

```
#include <pthread.h>
#define NUM_THREADS 4

void *work(void *i){
    printf("Hello, world from %i\n", pthread_self());
    pthread_exit(NULL);
}

int main(int argc, char **argv){
    int i;
    pthread_t id[NUM_THREADS];
    for(i = 0; i < NUM_THREADS; ++i){
        if(pthread_create(&id[i], NULL, work, NULL)){
            printf("Error creating the thread\n"); exit(19);
        }
    }
    printf("After creating the thread. My id is: %i\n",
        pthread_self());
    return 0;
}
```

Hello, world from 2
Hello, world from 3
After creating the thread.
My id is: 1
Hello, world from 4

What happened to thread 5???



Passing data to a thread at create time

Passing Single Argument

- Cast its value as a void pointer (a tricky pass by value)
- Cast its address as a void pointer (pass by reference).
 - The value that the address is pointing to should NOT change between thread creation and its use of the variable

Passing Multiple Arguments

- Heterogeneous: Create a structure with all the desired arguments and pass a pointer to an instance of that structure as a void pointer.
- Homogeneous: Create an array and then cast its name (which is a pointer) as a void pointer



Passing parameters on thread create, v1

Example 2a:

v1. All threads get a **pointer** to the same **int**

```
#include <pthread.h>
#define NUM_THREADS 10

void *work(void *i){
    int f = *((int *)i);
    printf("Hello, world from %i with value %i\n",
        pthread_self(), f);
    pthread_exit(NULL);
}

int main(int argc, char **argv){
    int i;
    pthread_t id[NUM_THREADS];
    for(i = 0; i < NUM_THREADS; ++i){
        if(pthread_create(&id[i], NULL, work, (void *)&i)){
            printf("Error creating the thread\n"); exit(19);}
    }
    return 0;
}
```

Hello, world from 2 with value 1
Hello, world from 3 with value 2
Hello, world from 6 with value 5
Hello, world from 5 with value 5
Hello, world from 4 with value 4
Hello, world from 8 with value 9
Hello, world from 9 with value 9
Hello, world from 10 with value 9
Hello, world from 7 with value 6
Hello, world from 11 with value 10

Probably wrong method
for this task!!



Passing parameters on thread create, v2

Example 2b:

v2. All threads get **value** of the same **int**

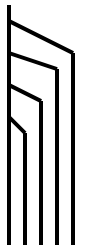
```
#include <pthread.h>
#define NUM_THREADS 10

void *work(void *i){
    int f = (int)(i);
    printf("Hello, world from %i with value %i\n",
        pthread_self(), f);
    pthread_exit(NULL);
}

int main(int argc, char **argv){
    int i;
    pthread_t id[NUM_THREADS];
    for(i = 0; i < NUM_THREADS; ++i){
        if(pthread_create(&id[i], NULL, work, (void *)(i))){
            printf("Error creating the thread\n");
            exit(19);
        }
    }
    return 0;
}
```

Hello, world from 2 with value 0
Hello, world from 3 with value 1
Hello, world from 4 with value 2
Hello, world from 5 with value 3
Hello, world from 6 with value 4
Hello, world from 7 with value 5
Hello, world from 8 with value 6
Hello, world from 10 with value 8
Hello, world from 11 with value 9

Right Method 1
(Kind of ...)



Passing parameters on thread create, v3

Example 2c:

v3. All threads get a *pointer* to a *different* int

```
#include <pthread.h>
#define NUM_THREADS 10

void *work(void *i){
    int f = *((int *)(i));
    printf("Hello, world from %i with value %i\n",
        pthread_self(), f);
    pthread_exit(NULL);}

int main(int argc, char **argv){
    int i;
    int y[NUM_THREADS];
    pthread_t id[NUM_THREADS];
    for(i = 0; i < NUM_THREADS; ++i){
        y[i] = i;
        if(pthread_create(&id[i], NULL, work, (void *)&y[i])){
            printf("Error creating the thread\n");
            exit(19);
        }
    }
    return 0;}
```

Hello, world from 2 with value 0
Hello, world from 4 with value 2
Hello, world from 5 with value 3
Hello, world from 6 with value 4
Hello, world from 7 with value 5
Hello, world from 8 with value 6
Hello, world from 9 with value 7
Hello, world from 3 with value 1
Hello, world from 10 with value 8
Hello, world from 11 with value 9

Right Method 2
(Definitely!)





Passing parameters on thread create, v4

v4. Pass multiple parameters to a thread with a *pointer* to a struct

- Want to create a thread to compute the sum of the elements of an array

```
void *do_work(void *arg);
```

- Needs three arguments
 - the array, its size, where to store the sum
 - we need to bundle them in a structure

```
struct arguments {  
    double *array;  
    int size;  
    double *sum;  
}
```



Passing parameters on thread create, v4, cont.

v4. Pass multiple parameters to a thread

```
int main(int argc, char *argv) {
    double array[100];
    double sum;
    pthread_t worker_thread;
    struct arguments *arg;

    arg = (struct arguments *)calloc(1,
                                     sizeof(struct arguments));

    arg->array = array;
    arg->size=100;
    arg->sum = &sum;

    if (pthread_create(&worker_thread, NULL,
                     do_work, (void *)arg)) {
        fprintf(stderr, "Error while creating thread\n");
        exit(1);
    }
    ...
}
```



Passing parameters on thread create, v4, cont.

v4. Pass multiple parameters to a thread

```
void *do_work(void *arg) {  
    struct arguments *argument;  
    int i, size;  
    double *array;  
    double *sum;  
  
    argument = (struct arguments*)arg;  
  
    size = argument->size;  
    array = argument->array;  
    sum = argument->sum;  
  
    *sum = 0;  
    for (i=0;i<size;i++)  
        *sum += array[i];  
  
    return NULL;  
}
```



Comments about the example

- The “main thread” continues its normal execution after creating the “child thread”
- **IMPORTANT:** If the main thread terminates, then all threads are killed!
 - We will see that there is a `join()` function
- Of course, memory is shared by the parent and the child (the array, the location of the sum)
 - nothing prevents the parent from doing something to it while the child is still executing
 - which may lead to a wrong computation
 - we will see that Pthreads provide locking mechanisms
- The bundling and unbundling of arguments is a bit tedious



Memory Management of Args *(this example)*

- The parent thread allocates memory for the arguments
- **Warning #1:** you don't want to free that memory before the child thread has a chance to read it
 - That would be a race condition
 - Better to let the child do the freeing, or free after join
- **Warning #2:** if you create multiple threads you want to be careful there is no sharing of arguments, or that the sharing is safe
 - For instance, if you reuse the same data structure for all threads and modify its fields before each call to `pthread_create()`, some threads may not be able to read the arguments destined to them
 - Safest way: have a separate arg structure for each thread

Passing parameters on thread create, v5

v5. Pass a different set of multiple parameters to each thread

```
// Pass an entire initialized structure.
// This is a good way to pass arbitrary. Package it
// all so that the data can get pointed at. To
// support multiple threads create an array of such
// pointers.

struct thread_data{
    int thread_id;
    int sum;
    char *message;
};

void *PrintHello(void *threadarg) {
    long taskid, sum;
    struct thread_data *my_data;
    char *message;

    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    message = my_data->message;

    printf(" Hello World! It's me, thread #%ld sum =
        %ld message = %s\n", taskid, sum, message);

    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    struct thread_data thread_data_array[NUM_THREADS];
    int rc;
    long t;
    char *Messages[NUM_THREADS] = {"First Message",
                                    "Second Message",
                                    "Third Message",
                                    "Fourth Message",
                                    "Fifth Message"};

    for (t = 0; t < NUM_THREADS; t++) {
        thread_data_array[t].thread_id = t;
        thread_data_array[t].sum = t+28;
        thread_data_array[t].message = Messages[t];
        printf("In main:  creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello,
                           (void*) &thread_data_array[t]);
        if (rc) {
            printf("ERROR; return code from pthread_create()
                is %d\n", rc);
            exit(-1);
        }
    }
} /* end main */
```

Joining Loose Ends: pthread_join

```
int pthread_join(pthread_t id, void **tr);
```

Make sure that the thread that has this *id* returns. Otherwise *waits* for it.

pthread_t id

The id of a created thread

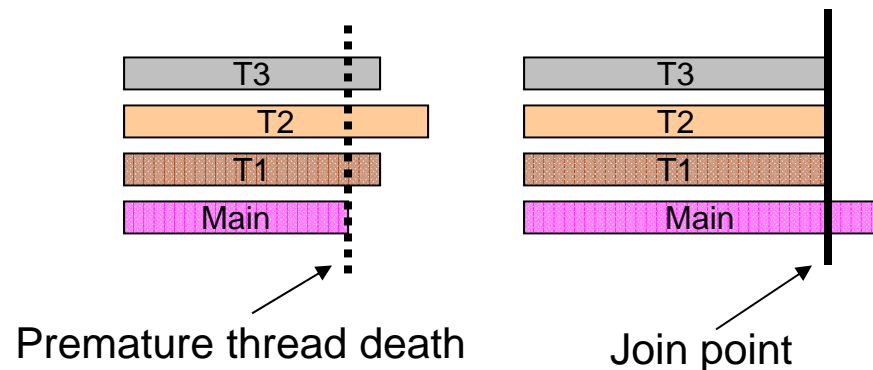
void **tr

A pointer to the result of the thread

OpenMP Counterpart

#pragma omp barrier

Returns a non zero value in success



Why use it? *If the main thread dies, then all other threads will die with it. Even if they have not completed their work. “join” tells main when it is OK to exit.*

pthread_join example

Example 3:

```
#include <pthread.h>
#define NUM_THREADS 4

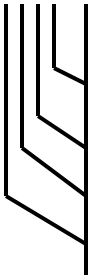
void *work(void *i){
    printf("Hello, world from %i\n", pthread_self());
    pthread_exit(NULL);
}


int main(int argc, char **argv){
    int i;
    pthread_t id[NUM_THREADS];

    for(i = 0; i < NUM_THREADS; ++i){
        if(pthread_create(&id[i], NULL, work, NULL)){
            exit(19);
        }
    }

    printf("After creating the thread. My id is: %i\n",
        pthread_self());
    for(i = 0; i < NUM_THREADS; ++i){
        if(pthread_join(id[i], NULL)){
            exit(19);
        }
    }
    printf("After joining all\n");
    return 0;
}
```

Hello, world from 2
Hello, world from 3
Hello, world from 4
After creating the thread.
My id is: 1
Hello, world from 5
After joining all





pthread_join() Warning

- This is a common “bug” that first-time pthread programmers encounter
 - Without the call to pthread_join() the previous program may end immediately, with the main thread reaching the end of main() thread and exiting, thus killing all other threads perhaps even before they have had a chance to execute
- When creating multiple threads be careful to store the handle of each thread in a separate variable
 - Typically one has an array of thread handles
- That way you’ll be able to call pthread_join() for each thread
- Also, note that the following code is sequential!

```
for (i=0; i < num_threads; i++) {  
    pthread_create(&(threads[i]),...)  
    pthread_join(threads[i],...)  
}
```

Another pthread_join example

- This code doesn't seem correct. If you set up the join after creating the thread you appear to have a race condition. But you can't join on a thread you haven't created because you don't have an ID. Do you have to Sync?
- Actually what happens is that the run time system keeps track of the "dead" threads for just such occasions. The logic is OK -- join is to prevent the exiting parent from killing the child thread, not the other way around.
- If you want to avoid setting up such a RTS structure, then you can give your thread the DETACHED attribute.

```
void *work(void *i) {
    printf(" Hello World!  It's me, thread #i!\n", pthread_self());
    pthread_exit(NULL);
}

/*****
int main(int argc, char *argv[]) {
    int arg,i,j,k,m;          /* Local variables. */
    pthread_t id[NUM_THREADS];

    for (i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&id[i], NULL, work, NULL)) {
            exit(19);
        }
    }

    printf("\n After creating a bunch of threads.  My own id is:
           %i\n", pthread_self());

    for (i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(id[i],NULL)){ exit(19);
        }
    }
}
*/ end main */
```

Detached Thread

- One option when creating a thread is whether it is joinable or detached
 - **Joinable**: another thread can call join on it
 - By default a thread is joinable
 - **Detached**: no thread can call join on it
- Let's look at the function that allows to set the “detached state”

- Set the detach state attribute

```
int pthread_attr_setdetachstate(  
    pthread_attr_t *attr,  
    int detachstate);
```

- returns 0 to indicate success, else error code
- attr: input parameter, thread attribute
- detachstate: can be either
 - **PTHREAD_CREATE_DETACHED**
 - **PTHREAD_CREATE_JOINABLE** (default)

- Detached threads have all resources freed when they terminate
- Joinable threads have state info about the thread kept even after they finish
 - To allow for a thread to join a finished thread
 - So-called “no rush to join”
- So, if you know that you will not need to join a thread, create it in a detached state so that you save resources
- This is lean-and-mean C, as opposed to hand-holding Java, and every little saving is important



Creating a Detached Thread

```
#include <pthread.h>
#define NUM_THREAD 25

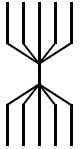
void *thread_routine (void *arg) {
    printf("Thread %d, my TID is %u\n",
        (int)arg, pthread_self());
    pthread_exit(0);
}

int main() {
    pthread_attr_t attr;
    pthread_t tids[NUM_THREADS];
    int x;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_create(&(tids[x]),
        &attr,
        thread_routine,
        (void *)x);

    . . . // should take a while otherwise
    . . . // the child may not have time to run
}
```

Synchronization Types



pthread

Mutex

Mutual Exclusion Lock. Only the thread that has the lock can access the protected region

Semaphores

A counter of resources that are available. Zero means no resources are left. Binary (Mutex) and Counting

Java

Monitors

Act as a guard of some resource. Consists of a mutex with some kind of notification scheme

pthread

Conditional Variables

Synchronization occurs when a condition is met. Always used in conjunction with a mutex. Inter-thread (process) communication.

pthread

Reader / Writer Locks

Permits only reads on a data or writes on data in a group. In other words, lock out the writers when the readers are on the shared data and vice versa.

Synchronization (1): Mutex

Pthread

```
1 int pthread_mutex_init(pthread_mutex_t *m,  
  const pthread_mutexattr_t *ma);  
2 int pthread_mutex_lock(pthread_mutex_t *m);  
3 int pthread_mutex_unlock(pthread_mutex_t *m);  
4 int pthread_mutex_destroy(pthread_mutex_t  
  *m);
```

OpenMP

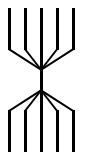
```
int omp_lock_init(omp_lock_t *lkc);  
int omp_lock_set(omp_lock_t *lkc);  
int omp_lock_unset(omp_lock_t *lkc);  
int omp_lock_destroy(omp_lock_t  
  *lkc);
```

1 → Initialization

2 → Setting

3 → Unsetting

4 → Destroying



pthread_mutexattr_t * ma can be left NULL for the default values



Mutual Exclusion and Pthreads

- Pthreads provide a simple mutual exclusion lock

- **Lock creation**

```
int pthread_mutex_init(  
    pthread_mutex_t *mutex,  
    const pthread_mutexattr_t *attr);
```

- returns 0 on success, an error code otherwise
- **mutex**: output parameter, lock
- **attr**: input, lock attributes
 - NULL: default
 - There are functions to set the attribute (look at the man pages if you're interested)

- **Locking a lock**

- If the lock is already locked, then the calling thread is blocked
- If the lock is not locked, then the calling thread “acquires” it

```
int pthread_mutex_lock(  
    pthread_mutex_t *mutex);
```

- returns 0 on success, an error code otherwise
- **mutex**: input parameter, lock



Pthread: Locking, cont.

■ Just checking

- Returns instead of locking

```
int pthread_mutex_trylock(  
    pthread_mutex_t *mutex);
```

- returns 0 on success, EBUSY if the lock is locked, an error code otherwise
- **mutex**: input parameter, lock

■ Releasing a lock

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

- returns 0 on success, an error code otherwise
- **mutex**: input parameter, lock

■ Releasing memory for a mutex

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


Example: Shared variable w/o mutex

Example 5

The Initial Balance: 100000.00
The Final Balance: 100010.00

```
#include <pthread.h>
#define NUM_THREADS 2
#define CYCLE 100
double b;

void *deposit(void *i){ double m = *(double *)(i); int j;
    for(j = 0; j < CYCLE; ++j){ b += m; sleep(1); }}

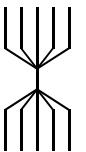
void *withdraw(void *i){double m = *(double *)(i); int j;
    for(j = 0; j < CYCLE; ++j){ b -= m; sleep(1); }}

int main(int argc, char **argv){
    int i; double bi; double q = 10.0;
    pthread_t id[NUM_THREADS];
    b = 100000; bi = b;

    pthread_create(&id[0], NULL, deposit , (void *)(&q));
    pthread_create(&id[1], NULL, withdraw, (void *)(&q));

    for(i = 0; i < NUM_THREADS; ++i){pthread_join(id[i], NULL);}

    printf("The Initial Balance: %.2lf\nThe Final Balance: %.2lf\n", bi, b);
    return 0;}
```



Example: Shared variable w/ mutex

Example 5, cont.

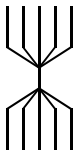
The Initial Balance: 100000.00
The Final Balance: 100000.00

```
...
pthread_mutex_t mt;
void *deposit(void *i){
    double m = *(double *)i;
    int j;

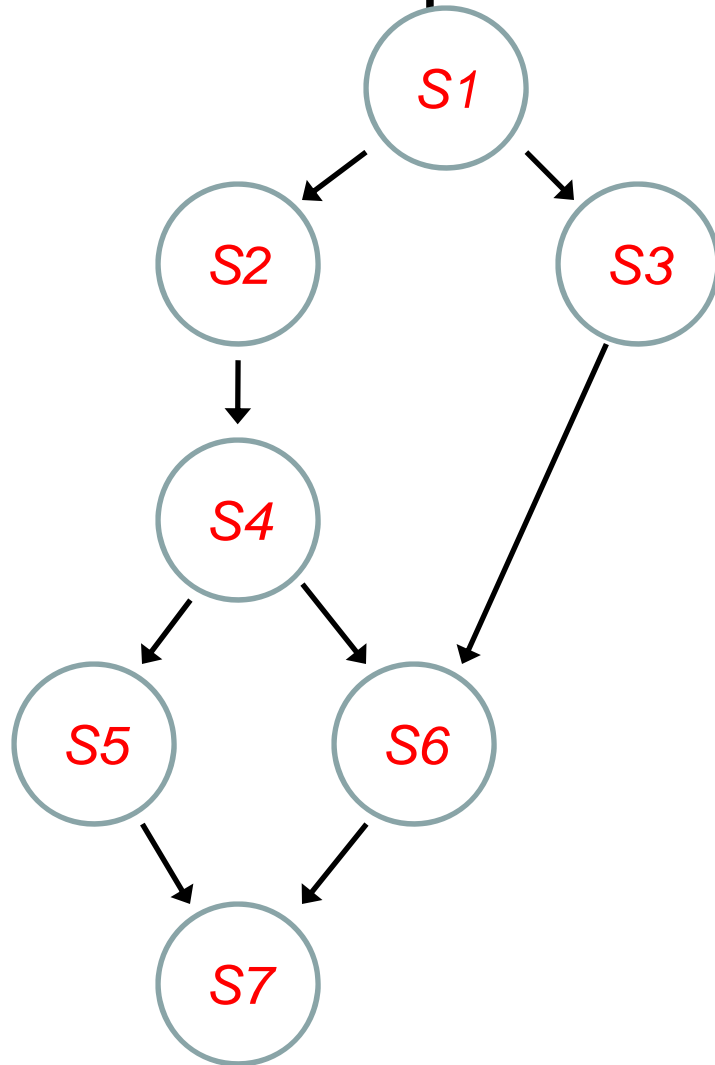
    for(j = 0; j < CYCLE; ++j){
        pthread_mutex_lock(&mt);
        b += m;
        pthread_mutex_unlock(&mt);
        sleep(1);
    }
}

void *withdraw(void *i){
    double m = *(double *)i;
    int j;
    for(j = 0; j < CYCLE; ++j){
        pthread_mutex_lock(&mt);
        b -= m;
        pthread_mutex_unlock(&mt);
        sleep(1);
    }
}

int main(int argc, char **argv){
    ...
    pthread_mutex_init(&mt, NULL);
    ...
    pthread_mutex_destroy(&mt);
    return 0; }
```



Another pthread_mutex example



```
int main(int argc, char *argv[]) {
    int arg,i,j,k,m,rc;
    pthread_t threads[NUM_THREADS+1];
    struct thread_data thread_data_array[NUM_THREADS+1];
    long t;
    char *Messages[NUM_THREADS+1] = {"Zeroth Message","First Message",
        "Second Message", "Third Message", "Fourth Message",
        "Fifth Message", "Sixth Message", "Seventh Message"};

    for (i = 0; i <= NUM_THREADS; i++) {
        if (pthread_mutex_init(&mutexA[i], NULL)) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

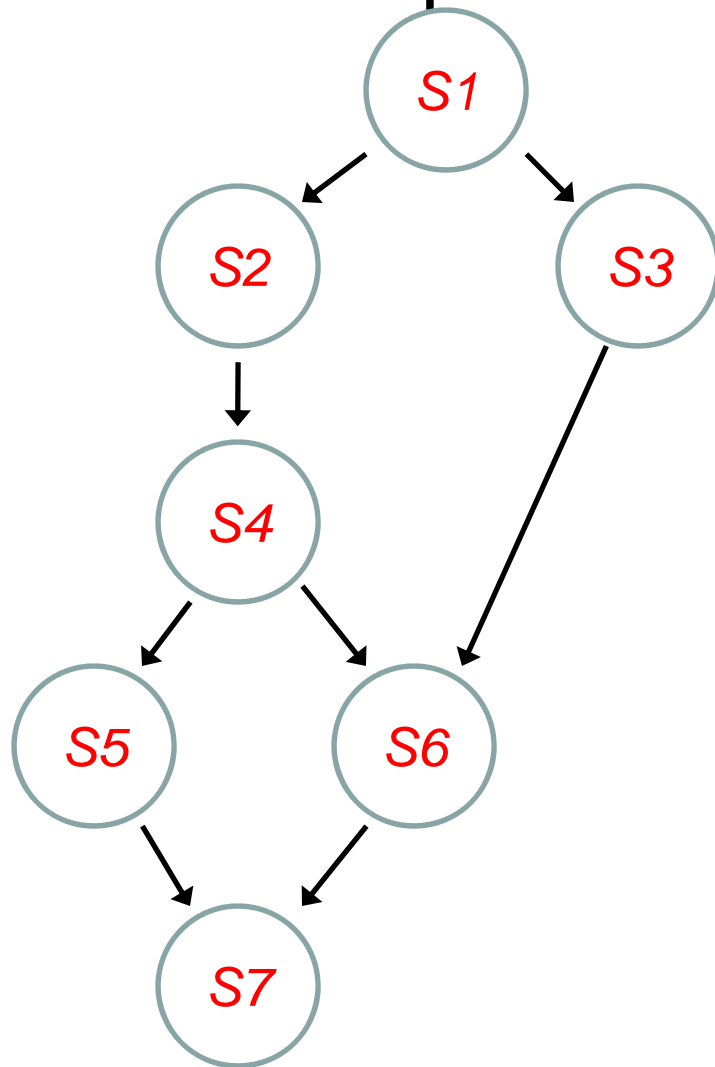
    for (i = 0; i <= NUM_THREADS; i++) {
        if (pthread_mutex_lock(&mutexA[i])) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    for (t = 2; t <= NUM_THREADS; t++) {
        thread_data_array[t].thread_id = t;
        thread_data_array[t].sum = t+28;
        thread_data_array[t].message = Messages[t];
        // printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello,
            (void*) &thread_data_array[t]);

        if (rc) {
            printf("ERROR; code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    if (pthread_mutex_unlock(&mutexA[1])) printf("\nERROR unlock\n");
    pthread_mutex_lock(&mutexA[7]);
}/* end main */
```

Another pthread_mutex example



```
void *PrintHello(void *threadarg) {
    long taskid, sum;
    struct thread_data *my_data;
    char *message;

    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    message = my_data->message;

    printf(" Hello World!  It's me, thread #%ld! sum = %ld message = %s",
           taskid, sum, message);
    switch (taskid) {
    case 2:
        printf("\n It's me, thread #%ld! I'm waiting 2 ... \n", taskid);
        pthread_mutex_lock(&mutexA[1]);
        pthread_mutex_unlock(&mutexA[1]); break;
    case 3:
        printf("\n It's me, thread #%ld! I'm waiting 3 ... \n", taskid);
        pthread_mutex_lock(&mutexA[1]);
        pthread_mutex_unlock(&mutexA[1]); break;
    case 4:
        printf("\n It's me, thread #%ld! I'm waiting 4 ... \n", taskid);
        pthread_mutex_lock(&mutexA[2]);
        pthread_mutex_unlock(&mutexA[2]); break;
    case 5:
        printf("\n It's me, thread #%ld! I'm waiting 5 ... \n", taskid);
        pthread_mutex_lock(&mutexA[4]);
        pthread_mutex_unlock(&mutexA[4]); break;
    case 6:
        printf("\n It's me, thread #%ld! I'm waiting 6 ... \n", taskid);
        pthread_mutex_lock(&mutexA[3]);
        pthread_mutex_unlock(&mutexA[3]);
        pthread_mutex_lock(&mutexA[4]);
        pthread_mutex_unlock(&mutexA[4]); break;
    case 7:
        printf("\n It's me, thread #%ld! I'm waiting 7 ... \n", taskid);
        pthread_mutex_lock(&mutexA[5]);
        pthread_mutex_unlock(&mutexA[5]);
        pthread_mutex_lock(&mutexA[6]);
        pthread_mutex_unlock(&mutexA[6]); break;
    default:
        printf("\n It's me, thread #%ld! I'm waiting ... \n", taskid); break;
    }
    printf("\n It's me, thread #%ld! I'm unlocking ... \n", taskid);
    if (pthread_mutex_unlock(&mutexA[taskid])) printf("\nERROR on unlock\n");
    printf("\n It's me, thread #%ld! I'm done ... \n", taskid);
    pthread_exit(NULL);
}
```

pthread_barrier example

```
// Barrier variable
pthread_barrier_t barr;

void *PrintHello(void *threadarg) {
    long taskid, sum, i;
    struct thread_data *my_data;
    char *message;

    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    message = my_data->message;

    // print some thread-specific stuff, then wait for
    // everybody else
    for (i = 0; i < 3; i++)
        printf("Thread %ld printing iteration %ld of 3\n",
               taskid, i+1);

    //wait at barrier
    int rc = pthread_barrier_wait(&barr);
    if (rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD) {
        printf("Could not wait on barrier\n");
        exit(-1);
    }

    //now print some more thread-specific stuff
    for (i = 0; i < 2; i++)
        printf("Thread %ld printing after barrier %ld
               of 2\n", taskid, i+1);

    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
    int arg,i,j,k,m;           /* Local variables. */
    pthread_t threads[NUM_THREADS];
    struct thread_data thread_data_array[NUM_THREADS];
    int rc;
    long t;
    char *Messages[NUM_THREADS] = {"First Message",
    "Second Message", "Third Message", "Fourth Message",
    "Fifth Message"};

    // Barrier initialization
    if(pthread_barrier_init(&barr, NULL, NUM_THREADS)) {
        printf("Could not create a barrier\n");
        return -1;  }

    for (t = 0; t < NUM_THREADS; t++) {
        thread_data_array[t].thread_id = t;
        thread_data_array[t].sum = t+28;
        thread_data_array[t].message = Messages[t];
        printf("In main:  creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello,
                           (void*) &thread_data_array[t]);
        if (rc) {
            printf("ERROR; return code from pthread_create()
                   is %d\n", rc);  exit(-1);  }  }

    for (t = 0; t < NUM_THREADS; t++) {
        if (pthread_join(threads[t],NULL)){
            printf("\n ERROR on join\n");
            exit(19);
        }
    }
    printf("\n After joining");
    return(0);
}/* end main */
```

Synchronization (2): Conditional Variables

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

Initialize a conditional variable *cond* with the attributes given by *attr*. The *attr* can be left NULL so it will use the default variable

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

De-allocated any resources associated with the conditional variable *cond*

```
int pthread_condattr_init (pthread_condattr_t *attr)
```

Initialize the conditional attribute variable *attr* with the default value.

```
int pthread_condattr_destroy (pthread_condattr_t *attr)
```

De-allocate the conditional attribute variable *attr*. It is safe to de-allocate the variable just after the conditional variable has been initialized

```
int pthread_cond_wait(pthread_cond_t *condition, pthread_mutex_t *m)
```

Make the calling thread wait for a signal in the conditional variable. Must be called when the associated mutex is locked and it will unlock it while the thread blocks. It will also unlock the mutex if the signal has been received.

```
int pthread_cond_signal(pthread_cond_t *condition)
```

Signal a thread that has been blocked waiting for condition to become true. It must be called **after** its associated mutex has been **locked** and the mutex must be **unlocked** after the signal has been **issued**

```
int pthread_cond_broadcast (pthread_cond_t *condition)
```

Similar to `pthread_cond_signal` but it signals all the waiting threads for this conditional variable

Condition Variable: example

- Say I want to have multiple threads wait until a counter reaches a maximum value and be awakened when it happens

```
pthread_mutex_lock(&lock);  
while (count < MAX_COUNT) {  
    pthread_cond_wait(&cond, &lock);  
}  
pthread_mutex_unlock(&lock)
```

- Locking the lock so that we can read the value of count without the possibility of a race condition
- Calling `pthread_cond_wait()` in a while loop to avoid “spurious wakes ups”
- When going to sleep the `pthread_cond_wait()` function **implicitly releases the lock**
- When waking up the `pthread_cond_wait()` function **implicitly acquires the lock**
- The lock is unlocked after exiting from the loop

Cond Example

Statically initialize the mutexes and the conditional variables

```
#include <pthread.h>
#define COUNT_DONE 10
#define COUNT_HALT1 3
#define COUNT_HALT2 6
```

```
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER;
int count = 0;
```

First function: If the count variable is between COUNT_HALT1 and COUNT_HALT2, wait for a signal from f2. Otherwise increment count

```
void *f1(){
    for(;;){
        pthread_mutex_lock( &condition_mutex );
        while( count >= COUNT_HALT1 &&
               count <= COUNT_HALT2 ){
            pthread_cond_wait( &condition_cond, &condition_mutex );
        }
        pthread_mutex_unlock( &condition_mutex );
        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value %i (f1): %d\n",pthread_self(), count);
        pthread_mutex_unlock( &count_mutex );
        if(count >= COUNT_DONE) pthread_exit(NULL);
    } }
}
```

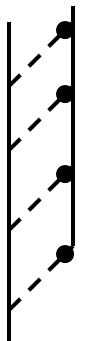
Second function: Send a signal to f1 if count is less than COUNT_HALT1 or greater than COUNT_HALT2. Later, increment the count

```
void *f2(){
    for(;;){
        pthread_mutex_lock( &condition_mutex );
        if( count < COUNT_HALT1 ||
           count > COUNT_HALT2 ){
            pthread_cond_signal( &condition_cond );
        }
        pthread_mutex_unlock( &condition_mutex );
        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value %i (f2): %d\n",pthread_self(), count);
        pthread_mutex_unlock( &count_mutex );
        if(count >= COUNT_DONE) pthread_exit(NULL);
    } }
}
```

Counter value 2 (f1): 1
Counter value 2 (f1): 2
Counter value 2 (f1): 3
Counter value 3 (f2): 4
Counter value 3 (f2): 5
Counter value 3 (f2): 6
Counter value 3 (f2): 7
Counter value 3 (f2): 8
Counter value 3 (f2): 9
Counter value 3 (f2): 10
Counter value 2 (f1): 11

```
int main(int argc, char **argv)
{
    pthread_t thread1, thread2;
    pthread_create( &thread1, NULL, &f1, NULL);
    pthread_create( &thread2, NULL, &f2, NULL);
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    return(0);
}
```

F1 is effectively halted between COUNT_HALT1 and COUNT_HALT2. Otherwise random



Attributes Revisited: Useful Functions (1)

```
int pthread_attr_init(pthread_attr_t *attr);
```

Initialize an attribute with the default values for the attribute object

- **Default Schedule:** SCHED_OTHER (?)
- **Default Scope:** PTHREAD_SCOPE_SYSTEM (?)
- **Default Join State:** PTHREAD_CREATE_JOINABLE (?)

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

De-allocate any memory and state that the attribute object occupied. It is safe to delete the attribute object after the thread has been created

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int JOIN_STATE);
```

Set the attached parameter on the attribute object with the *JOIN_STATE* variable

- **PTHREAD_CREATE_JOINABLE:** It can be joined at a join point. State must be saved after function ends
- **PTHREAD_CREATE_DETACHED:** It cannot be joined at a join point. State and resources are de-allocated immediately



Attributes Revisited: Useful Functions (2)

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)
```

Set the scheduling policy of the thread:

- **SCHED_OTHER** → Regular scheduling
- **SCHED_RR** → Round-robin (**SU**)
- **SCHED_FIFO** → First-in First-out (**SU**)

```
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *pr)
```

Contains the schedule priority of the thread

Default: 0

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit)
```

Tell if the scheduling parameters will be inherit from the parent or the ones in the attribute object will be used

PTHREAD_EXPLICIT_SCHED → Scheduling parameters from the attribute object will be used.

PTHREAD_INHERIT_SCHED → inherit the attributes from its parent.

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope)
```

Contention parameter

- **PTHREAD_SCOPE_SYSTEM**
- **PTHREAD_SCOPE_PROCESS**



Miscellaneous Useful Functions

```
int pthread_attr_getstackaddr (const pthread_attr_t *attr, void **stackaddr)
```

Return the stack address that this P-thread will be using

```
int pthread_attr_getstacksize (const pthread_attr_t *attr, size_t *stacksize)
```

Return the stack size that this P-thread will be using

```
int pthread_detach (pthread_t thr, void **value_ptr)
```

Make the thread that is identified by *thr* not joinable

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

Make sure that the *init_routine* is executed by a single thread and only once. The *once_control* is a synchronization mechanism that can be defined as:

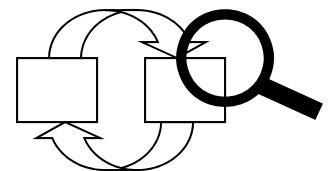
```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

OpenMP Counterpart

```
#pragma omp single
```

```
void pthread_yield ()
```

Relinquish the use of the processor



```

#include <pthread.h>
#define NUM_THREADS 4

struct args{int a; float b; char c;};
void *work(void *i){
    struct args *a = (struct args *)i;
    printf("(%3i, %.3f, %3c) --> %i\n", a->a, a->b, a->c, pthread_self());
    pthread_exit(NULL);
}

int main(int argc, char **argv){
    int i;
    struct args a[NUM_THREADS];
    pthread_t id[NUM_THREADS];
    pthread_attr_t ma;
    pthread_attr_init(&ma);
    pthread_attr_setdetachstate(&ma, PTHREAD_CREATE_JOINABLE);
    for(i = 0; i < NUM_THREADS; ++i){
        a[i].a = i; a[i].b = 1.0 / (i+1); a[i].c = 'a' + (char)(i);
        pthread_create(&id[i], &ma, work, (void *)(&a[i]));
    }
    pthread_attr_destroy(&ma);
    for(i = 0; i < NUM_THREADS; ++i){pthread_join(id[i], NULL);}
    return 0;}

```

Example 4:

```

( 0, 1.000, a) --> 2
( 3, 0.250, d) --> 5
( 2, 0.333, c) --> 4
( 1, 0.500, b) --> 3

```





pthread_cond_timedwait()

■ Waiting on a condition variable with a timeout

```
int pthread_cond_timedwait(  
    pthread_cond_t *cond,  
    pthread_mutex_t *mutex,  
    const struct timespec *delay);
```

- returns 0 on success, an error code otherwise
- **cond**: input parameter, condition
- **mutex**: input parameter, associated mutex
- **delay**: input parameter, timeout (same fields as the one used for gettimeofday)