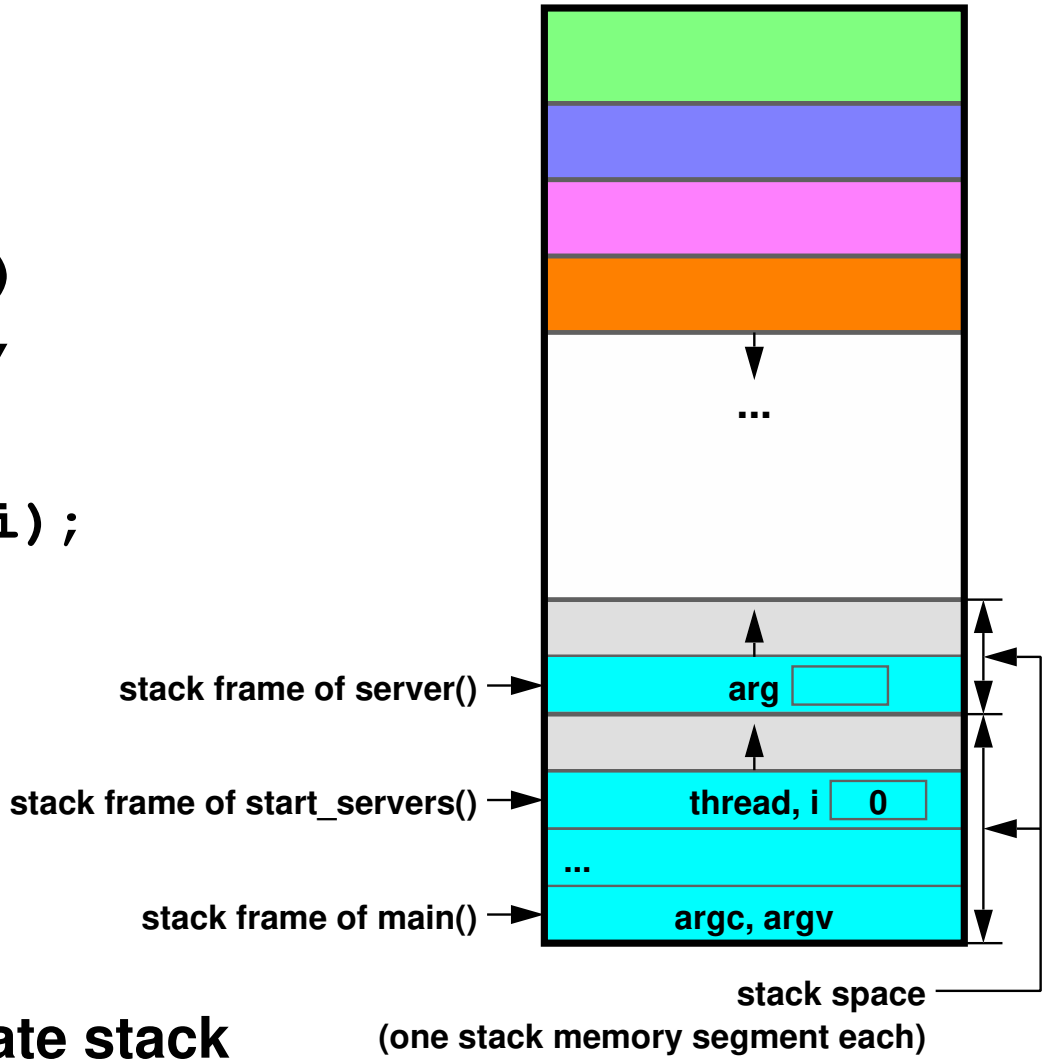
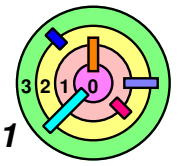
Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i = 0; i < 100; i++)
    pthread_create(&thread,
                   0,
                   server,
                   (void*)i);
}

void *server(void *arg) {
  int k=(int)arg;
  // perform service
  return(0);
}
```

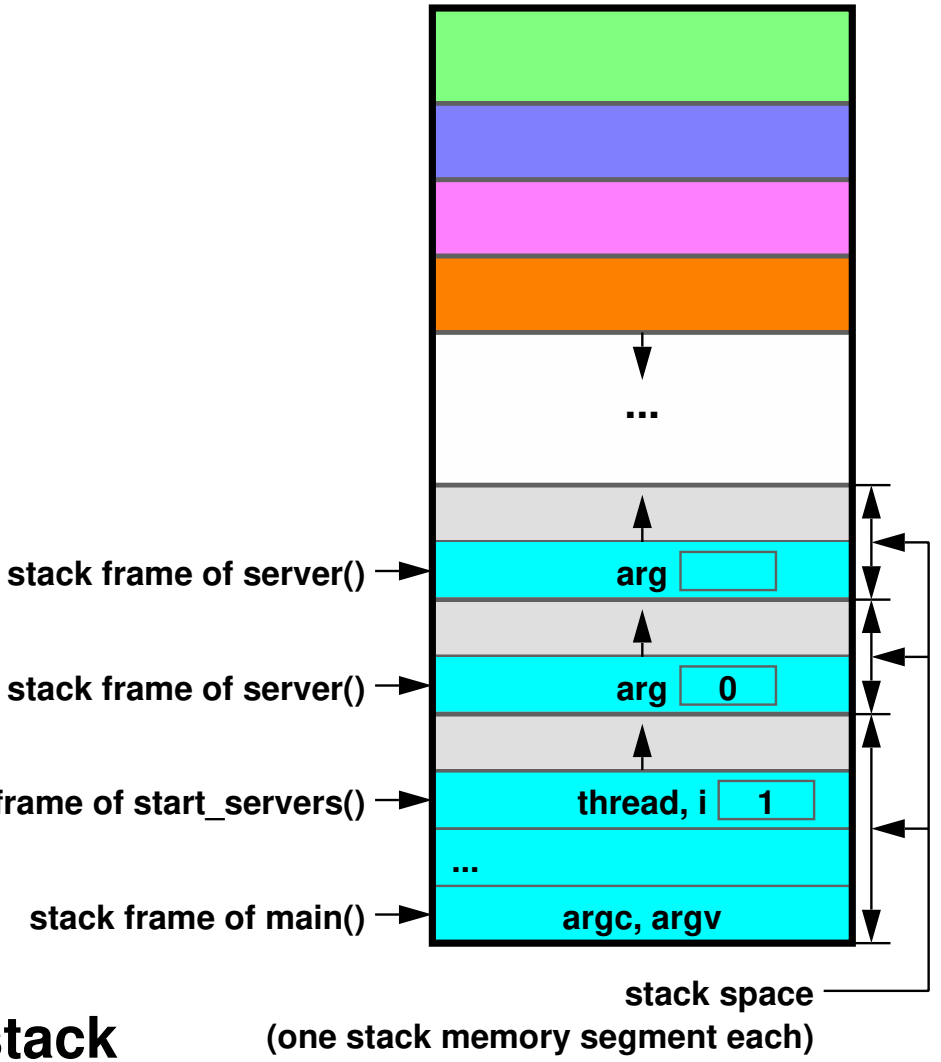


- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - one arg in each of these stack frames
    - a stack space is in its own stack memory segment

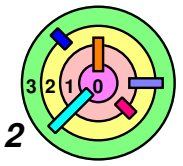


Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i = 0; i < 100; i++)
    pthread_create(&thread,
                   0,
                   server,
                   (void*)i);
}

void *server(void *arg) {
  int k=(int)arg;
  // perform service
  return(0);
}
```

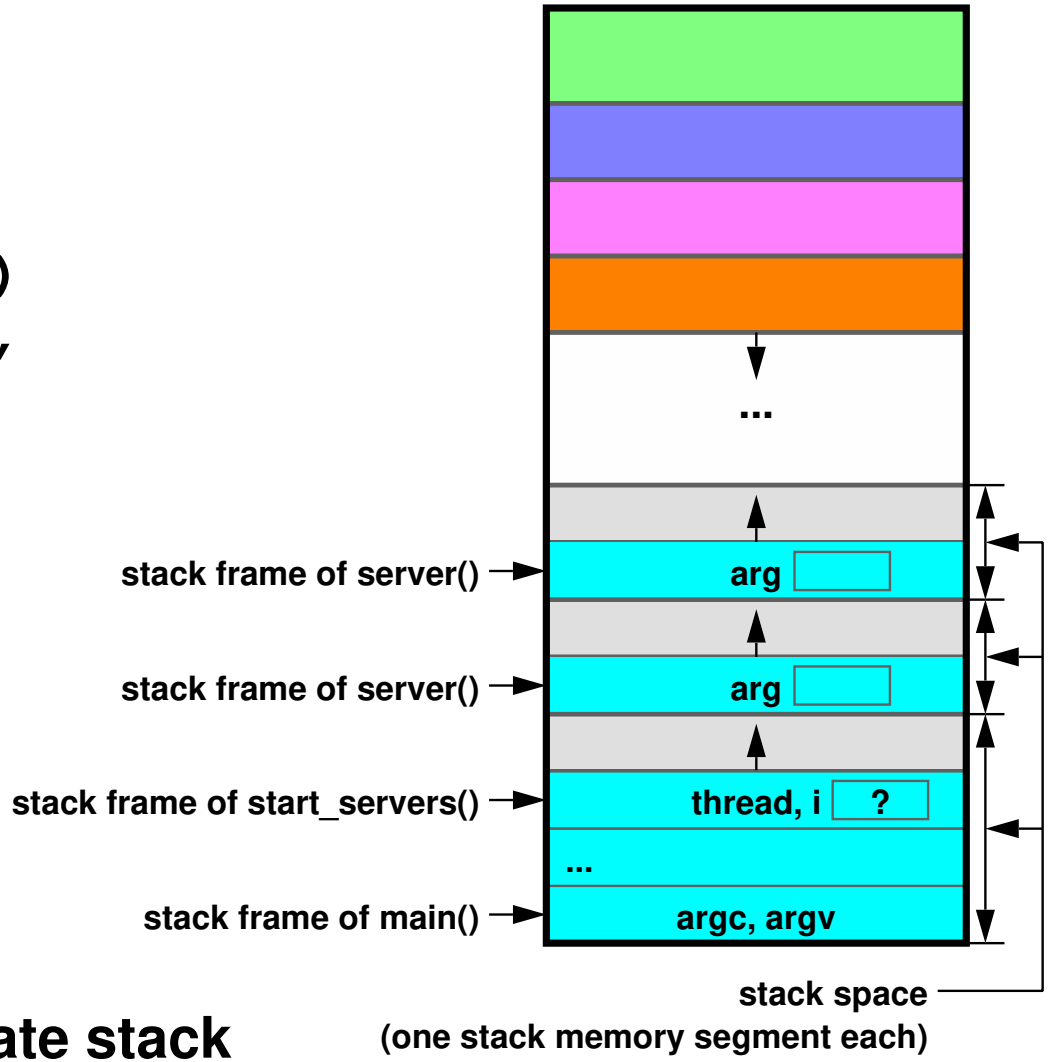


- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - one arg in each of these stack frames
    - a stack space is in its own stack memory segment

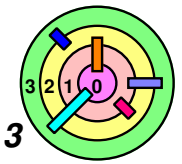


Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i = 0; i < 100; i++)
    pthread_create(&thread,
                   0,
                   server,
                   &i);
}

void *server(void *arg) {
  int *iptr=(int*)arg;
  // perform service
  return(0);
}
```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - one arg in each of these stack frames
    - a stack space is in its own stack memory segment



Creating a POSIX Thread

- These are the same:
  - keep thread handle in the stack

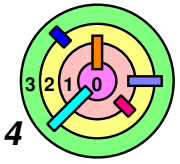
```
pthread_t thread;
pthread_create(&thread, ...);
```

  - keep thread handle in the heap

```
pthread_t *thread_ptr =
    (pthread_t*)malloc(sizeof(pthread_t));
pthread_create(thread_ptr, ...);
```

    - need to make sure that eventually you will call the following to not leak memory

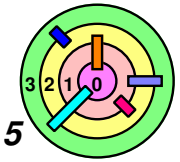
```
free(thread_ptr);
```



Creating a Win32 Thread

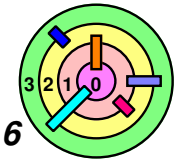
```
start_servers( ) {
   HANDLE thread;
   DWORD id;
   int i;
   for (i = 0; i < 100; i++)
     thread = CreateThread(
          0,        // security attributes
          0,        // default # of stack pages allocated
          server,   // first procedure
          arg,      // argument
          0,        // default attributes
          0,        // creation flags
          &id);     // thread ID
}

DWORD WINAPI server(void *arg) {
   // perform service
   return(0);
}
```

- We won't talk about Win32 much



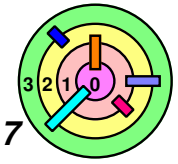
Complications

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  pthread_create(&in_thread,
                 0,
                 incoming,
                 r_in, l_out); // Cannot do this ...
  pthread_create(&out_thread,
                 0,
                 outgoing,
                 l_in, r_out); // Cannot do this ...
  /* How do we wait till they are done? */
}
```



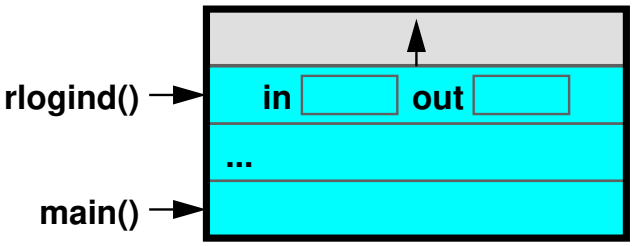
Multiple Arguments

```
typedef struct {
  int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
                 0,
                 incoming,
                 &in);
  ...
  /* How do we wait till they are done? */
}

void *incoming(void *arg) {
  two_ints_t *p=(two_ints_t*)arg;
  ... p->first ...
  return NULL;
}
```



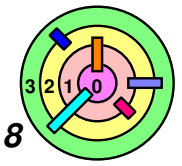
Multiple Arguments

```
typedef struct {
  int first, second;
} two_ints_t;
```



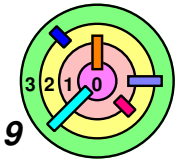

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
                 0,
                 incoming,
                 &in);
  ...
  /* How do we wait till they are done? */
}
```


```
void *incoming(void *arg) {
  two_ints_t *p=(two_ints_t*)arg;
  ... p->first ...
  return NULL;
}
```



Multiple Arguments

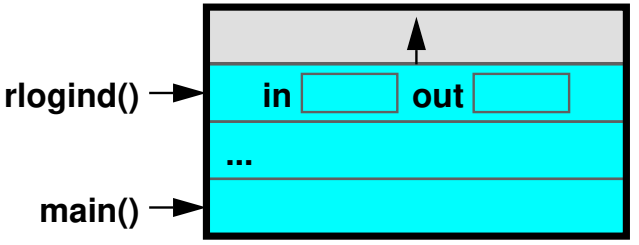
- Need to be careful how to pass argument to new thread when you call `pthread_create()`
  - passing address of a *local* variable (like the previous example) only works if we are certain the this storage doesn't go out of scope until the thread is done with it
  - passing address of a *static* or a *global* variable only works if we are certain that only one thread at a time is using the storage
  - passing address of a *dynamically* allocated storage only works if we can free the storage when, and only when, the thread is finished with it
    - this would not be a problem if the language supports garbage collection
- Memory corruption happens when memory is re-used unexpectedly
  - ask yourself, "How can I be sure?"
    - if the answer is, "I hope it works", then you need a different solution



When Is The Child Thread Done?

- In our example, what would happen if we return from `rlogind()` right after the child threads are created?

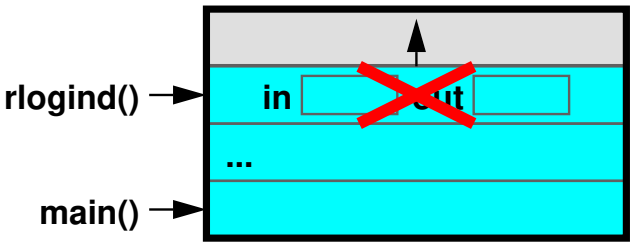
```
typedef struct {
  int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
                 0,
                 incoming,
                 &in);
}
void *incoming(void *arg) {
  two_ints_t *p=(two_ints_t*)arg;
  ... p->first ...
  return NULL;
}
```



When Is The Child Thread Done?

- In our example, what would happen if we return from `rlogind()` right after the child threads are created?

```
typedef struct {
  int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
                 0,
                 incoming,
                 &in);
}
void *incoming(void *arg) {
  two_ints_t *p=(two_ints_t*)arg;
  ... p->first ...
  return NULL;
}
```



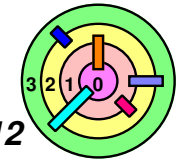
When Is The Child Thread Done?

- To wait for a child thread to die, use pthread_join()

```
int pthread_join(thread_t thread,
                 (void **)ret_value);
```

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};

  pthread_create(&in_thread, 0, incoming, &in);
  pthread_create(&out_thread, 0, outgoing, &out);

  /* if not interested in thread return values */
  pthread_join(in_thread, 0);
  pthread_join(out_thread, 0);
}
```

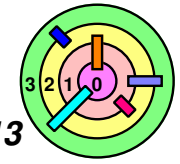
  - `(void**)` is the address of a variable of type `(void*)`
  - `pthread_join()` is a *blocking call*
    - can only return if the specified thread has terminated



Thread Termination

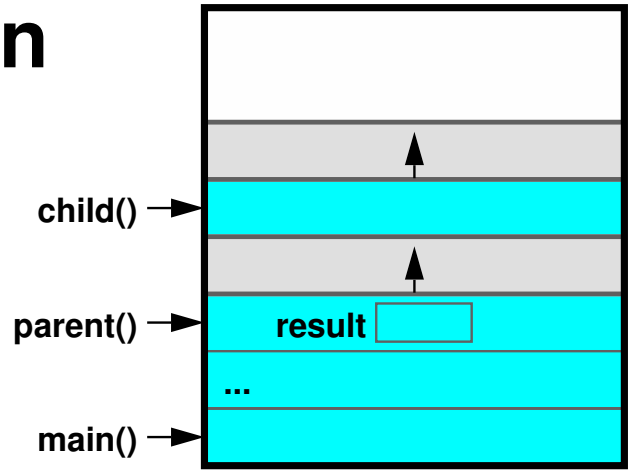
- Thread return values
  - which threads receive these values
  - how do they do it?
    - clearly, receiving thread must wait until the producer thread produced it, i.e., producer thread has terminated
    - so we must have a way for one thread to wait for another thread to terminate
  - must have a way to say which thread you are waiting for
    - need a unique identifier
    - tricky if it can be reused

```
int pthread_join(thread_t thread,
                 (void **)ret_value);
```



Thread Termination

- How does a thread *self-terminate*?
  1) return from its "first procedure"
     - return a value of type `(void*)`
  2) call `pthread_exit(ret_value)`
     - `ret_value` is of type `(void*)`

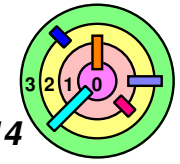



```
parent() {
  pthread_t thread;
  void *result=(void*)0;
  pthread_create(&thread,
       0, child, 0);
  pthread_join(thread,
       (void**)&result);
  switch ((int)result) {
  case 1: ...
  case 2: ...
  }
  ...
}
```

```
void *child(void *arg) {
  ...
  if (terminate_now) {
    pthread_exit((void*)1);
  }
  return((void*)2);
}
```

Thread Control Block

TID
Exit/Return Code
...

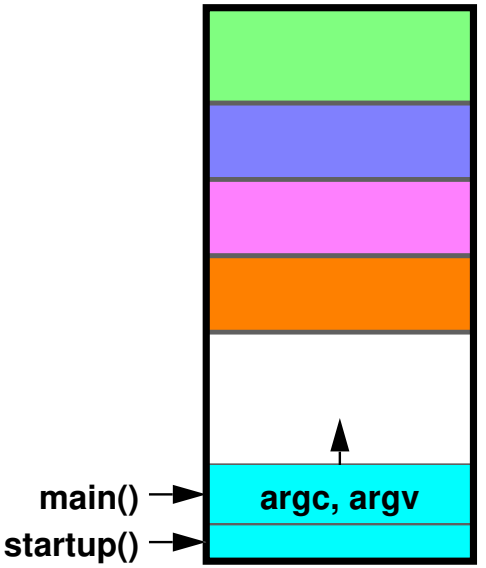


Thread Termination

- Difference between `pthread_exit()` and `exit()`
  - `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what will this code do?

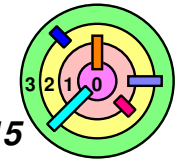
```
int main(int argc, char *argv[]) {
  // create all the threads
  return(0);
}
```

    - when `main()` returns, `exit()` will be called
      - as a result, none of the created child threads may get a chance to run



- `main()` is called by a *"startup routine"*:

```
exit(main(argc,argv))
```



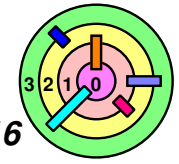
Thread Termination

- Difference between `pthread_exit()` and `exit()`
  - `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what about this code?

```
int main(int argc, char *argv[]) {
  // create all the threads
  pthread_exit(0); // exit the main thread
  return(0);
}
```

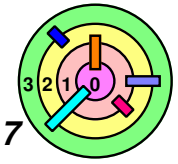
    - here, `pthread_exit()` will terminate the main thread, so `exit()` is never called
      - as it turns out, this special case is taken care of in the pthread library implemetation

- You should use `pthread_join()` unless you are absolutely sure what you are doing



Thread Termination

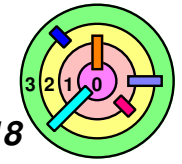
- Calling `pthread_exit()` is the only way a thread can *self-terminate* (without affecting other threads)
- Any thread can join with any other thread
  - there's *no parent/child relationships* among threads
    - unlike process termination and `wait()`
- What happens if a thread terminates and no other thread wants to join with this thread?
  - it also goes into a *zombie* state
    - all the thread related information is freed up, except for the thread ID, return code, and stack space
      - a running thread cannot delete its own stack!
- What if two threads want to join with the same thread?
  - after the first thread joins, the thread ID and return code are freed up and the thread ID may get reused
  - so don't do this!



Detached Threads

- What if you have a thread that you don't want any thread to join with it?
  - you can detach the thread after you have created it

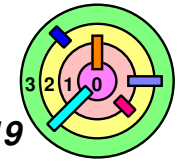
```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i = 0; i < 100; i++) {
    pthread_create(&thread, 0, server, 0);
    pthread_detach(thread);
  }
  ...
}

server( ) {
  ...
}
```



Types

```
pthread_create(&tid,
               0,
               (void *(*)(void *))func,
               (void *)1);

int func = 4; // func definition 1

void func(int i) { // func definition 2
   ...
}

void *func(void *arg) { // func definition 3
   int i = (int)arg;
   ...
   return(0);
}
```

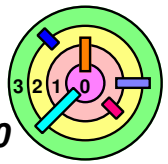
- a function is just an address (of something in the text/code segment)



Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
/* establish some attributes */
...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);
...
```

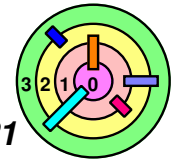
- thread attribute only needs to be valid when a thread is created
  - therefore, it can be destroyed as soon as the thread is created



Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);
...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);
```

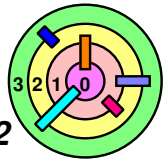
- the above code set the stack size to 20MB
- the default stack size is not small
  - default stack size is probably around 1MB in Solaris and 8MB in some Linux implementations
  - if you need to create a lot of threads, you may want to have smaller stack size
  - if you have very deep recursion in your code, you may want a bigger stack size



Example

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

#define M  3
#define N  4
#define P  5

int A[M][N];
int B[N][P];
int C[M][P];

void *matmult(void *arg) {
  int row = (int)arg, col;
  int i, t;

  for (col=0; col < P; col++) {
    t = 0;
    for (i=0; i<N; i++)
      t += A[row][i] * B[i][col];
    C[row][col] = t;
  }
  return(0);
}
```

```
main( ) {
  int i;
  pthread_t thr[M];
  int error;

  /* initialize the matrices ... */
  ...
  // create the worker threads
  for (i=0; i<M; i++) {
    if (error = pthread_create(
        &thr[i],
        0,
        matmult,
        (void *)i)) {
      fprintf(stderr,
          "pthread_create: %s",
          strerror(error));
      exit(1);
    }
  }
  // wait for workers to finish
  for (i=0; i<M; i++)
    pthread_join(thr[i], 0)
  /* print the results ... */
}
```



Compiling It

```
% gcc -o mat mat.c -pthread
```

- technically speaking, "`-pthread`" is for *linking* with the pthread library
- *compiling* is to compile "`mat.c`" into "`mat.o`"
  - "`mat.o`" is deleted after the "`mat`" executable file is created
- another syntax is (for Unix systems in general):

```
% gcc -o mat mat.c -lpthread
```