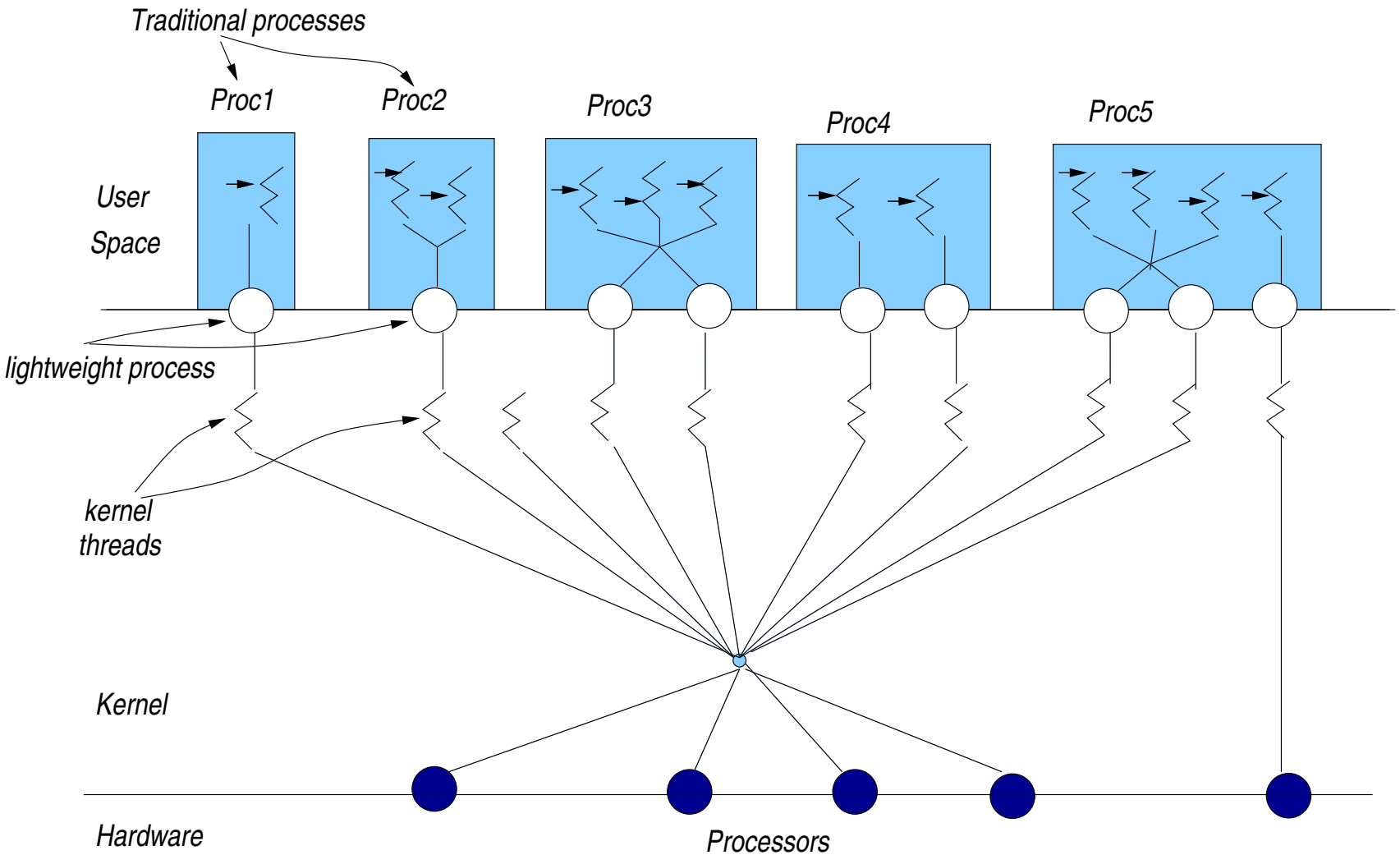
Threads

April 2017

Threads

- Threads are an alternative to multi-tasking.
- Try to overcome penalties when it comes to context switching and synchronization among different "flows" (or sequences) of execution.
- Offer a more efficient way to develop applications.

Thread (Solaris) Model



Thread Highlights

- One or more threads may be executed in the context of a process.
- The entity that is being scheduled is the thread – **not** the process itself.
- In the presence of a single processor, threads are executed concurrently.
- If there are more than one processors, threads can be assigned to different kernel thread (and so different CPUs) and run in parallel.
- Any thread may create a new thread.

Thread Highlights (continued)

- All threads of a single process share the same address space (address space, file descriptors etc.) BUT they have their own PC, stack and set of registers.
- Evidently, the kernel may manage *faster* the switch from one thread to another than the respective change from one process to another.
- The header `#include <pthread.h>` is required by all programs that use threads.
- Programs have to be compiled with the `pthread` library.
  `gcc <filename>.c -lpthread`

Thread Highlights (continued)

- The functions of the `pthread` library do not set the value of the variable `errno` and so, we cannot use the function `perror()` for the printing of a diagnostic message.
- If there is an error in one of the thread functions, `strerror()` is used for the printing of the diagnostic code (which is the "function return" for the thread).
- Function `char *strerror(int errnum)`
  - returns a pointer to a string that describes the error code passed in the argument `errnum`.
  - requires: `#include <string.h>`

Threads vs. Processes

	Threads	*Processes*
Address Space	Common. Any change made by one thread is visible to all (ie, `malloc()/free()`)	Different for each process Afer a `fork()` we have different address spaces
File Descriptors	Common. Any two threads can use the same descriptor One `close()` on this descriptor is sufficient	Two processes use copies of the file descriptors

What happens to threads when...

	What happens..
`fork`	Only the thread that invoked `fork` is duplicated.
`exit`	All threads die together (`pthread_exit` for the termination of a single thread).
`exec`	All threads disappear (the shared/common address space is replaced)
`signals`	This is somewhat more complex - Section 13.5 –*Robbins*[2] book

POSIX Thread Management

POSIX function	description
`pthread_create`	create a thread
`pthread_self`	find out own thread ID
`pthread_equal`	test 2 thread IDs for equality
`pthread_exit`	exit thread without existing process
`pthread_detach`	set thread to release resources
`pthread_join`	wait for a thread
`pthread_cancel`	terminate another thread
`pthread_kill`	send a signal to a thread

Creation of Threads

- The function that helps generate a thread is:

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *), void *arg);
```

- creates a new thread with attributes specified by `attr` within a process.
- if `attr` is NULL, default attributes are used.
- Upon successful completion, `pthread_create()` shall store the ID of the created thread in the location referenced by *thread*.
- Through the `attr` we can change features of the thread but oftentimes we let the default value work, giving a `NULL`.
- If successful, the function returns `0`; otherwise, an error number shall be returned to indicate the error.

Terminating a Thread

- ```
  void pthread_exit(void *retval);
  ```

- terminates the calling thread and makes the value `retvalue` available to any successful join with the terminating thread.

- After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. So, references to local variables of the exiting thread should not be used for the `retvalue` parameter value.
  ```

pthread_join - waiting for thread termination

- ```
  int pthread_join(pthread_t thread, void **retval);
  ```

- suspends execution of the calling thread until the target `thread` terminates (unless the target `thread` *has already* terminated).
- When a `pthread_join()` returns successfully, the target `thread` has been terminated.
- On successful completion, the function returns 0.
- If retval is not NULL, then `pthread_join()` copies the exit status of the target thread into the location pointed to by `*retval`. If `thread` was canceled, then `PTHREAD_CANCELED` is placed in `*retval`.
  ```

Identifying - Detaching Threads

$\Longrightarrow$ Get the calling thread-ID:

- `pthread_t pthread_self(void);`
- returns the thread-ID of the calling thread.

$\Longrightarrow$ Detaching a thread:

- `int pthread_detach(pthread_t thread);`
- indicates that the storage for the `thread` can be reclaimed only when the thread terminates.
- If `thread` has not terminated, `pthread_detach()` shall not cause it to terminate.
- If the call succeeds, `pthread_detach()` shall return 0; otherwise, an error number shall be returned.
- Issuing a `pthread_join` on a detached thread fails.

Creating and using threads

```c
#include <stdio.h>
#include <string.h>    /* For strerror */
#include <stdlib.h>    /* For exit     */
#include <pthread.h>   /* For threads  */
#define perror2(s,e) fprintf(stderr, "%s: %s\n", s, strerror(e))

void *thread_f(void *argp){ /* Thread function */
   printf("I am the newly created thread %ld\n", pthread_self());
   pthread_exit((void *) 47); // Not recommended way of "exit"ing
   }                          // avoid using automatic variables
                              // use malloc-ed structs to return status
main(){
  pthread_t thr;
  int err, status;
  if (err = pthread_create(&thr, NULL, thread_f, NULL)) { /* New thread */
      perror2("pthread_create", err);
      exit(1);
      }
  printf("I am original thread %ld and I created thread %ld\n",
           pthread_self(), thr);
  if (err = pthread_join(thr, (void **) &status)) { /* Wait for thread */
      perror2("pthread_join", err); /* termination */
      exit(1);
      }
  printf("Thread %ld exited with code %d\n", thr, status);
  printf("Thread %ld just before exiting (Original)\n", pthread_self());
  pthread_exit(NULL);
  }
```

Outcome

```
ad@haiku:~/src$
ad@haiku:~/src$ ./create_a_thread
I am original thread 140400641664832 and I created thread 140400633423616
I am the newly created thread 140400633423616
Thread 140400633423616 exited with code 47
Thread 140400641664832 just before exiting (Original)
ad@haiku:~/src$
ad@haiku:~/src$
```

Using pthread_detach

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

#define perror2(s,e) fprintf(stderr,"%s: %s\n",s,strerror(e))

void *thread_f(void *argp){ /* Thread function */
   int err;
   if (err = pthread_detach(pthread_self())) {/* Detach thread */
       perror2("pthread_detach", err);
       exit(1);
       }
   printf("I am thread %ld and I was called with argument %d\n",
              pthread_self(), *(int *) argp);
   pthread_exit(NULL);
}
```

Using pthread_detach

```
main(){
   pthread_t thr;
   int err, arg = 29;

   if (err = pthread_create(&thr,NULL,thread_f,(void *) &arg)){/* New thread */
       perror2("pthread_create", err);
       exit(1);
       }
    printf("I am original thread %d and I created thread %d\n",
              pthread_self(), thr);
    pthread_exit(NULL);
}
```

→ Outcome:

```
ad@haiku:~/Set007/src$ ./p16-detached_thread
I am thread 140411743098624 and I was called with argument 29
I will wait for some time in detached now.. 10 secs
I am original thread 140411751352064 and I created thread 140411743098624
I am the original and I am done!
About to leave detached thread now.. 140411743098624
ad@haiku:~/Set007/src$
```

Create *n* threads that wait for random secs and then terminate

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_SLEEP 10 /* Maximum sleeping time in seconds */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))

void *sleeping(void *arg) {
   long sl = (long) arg;
   printf("thread %ld sleeping %ld seconds ...\n", pthread_self(), sl);
   sleep(sl); /* Sleep a number of seconds */
   printf("thread %ld waking up\n", pthread_self());
   pthread_exit(NULL);
}

main(int argc, char *argv[]){
   int  n, i, err;
   long sl;
   pthread_t *tids;
   if (argc > 1) n = atoi(argv[1]); /* Make integer */
   else exit(0);
   if (n > 50) { /* Avoid too many threads */
      printf("Number of threads should be up to 50\n");
      exit(0);
      }

   if ((tids = malloc(n * sizeof(pthread_t))) == NULL) {
      perror("malloc");
      exit(1);
      }
```

n threads waiting for randm secs

```
    srandom((unsigned int) time(NULL)); /* Initialize generator */
    for (i=0 ; i<n ; i++) {
    sl = random() % MAX_SLEEP + 1; /* Sleeping time 1..MAX_SLEEP */
    if (err = pthread_create(tids+i, NULL, sleeping, (void *) sl)) {
       /* Create a thread */
       perror2("pthread_create", err);
       exit(1);
       }
    }

    for (i=0 ; i<n ; i++)
    if (err = pthread_join(*(tids+i), NULL)) {
       /* Wait for thread termination */
       perror2("pthread_join", err);
       exit(1);
       }
    printf("all %d threads have terminated\n", n);
}
```

Outcome

```
ad@haiku:~/src$
ad@haiku:~/src$ ./random_sleeps 3
thread 140648973833984 sleeping 3 seconds ...
thread 140648965441280 sleeping 4 seconds ...
thread 140648982226688 sleeping 4 seconds ...
thread 140648973833984 waking up
thread 140648965441280 waking up
thread 140648982226688 waking up
all 3 threads have terminated
ad@haiku:~/src$ ./random_sleeps 6
thread 140434098566912 sleeping 5 seconds ...
thread 140434115352320 sleeping 8 seconds ...
thread 140434081781504 sleeping 2 seconds ...
thread 140434090174208 sleeping 8 seconds ...
thread 140434106959616 sleeping 5 seconds ...
thread 140434073388800 sleeping 4 seconds ...
thread 140434081781504 waking up
thread 140434073388800 waking up
thread 140434098566912 waking up
thread 140434106959616 waking up
thread 140434115352320 waking up
thread 140434090174208 waking up
all 6 threads have terminated
ad@haiku:~/src$
ad@haiku:~/src$
```

Going from sigle- to multi-threaded programs

```c
#include <stdio.h>
#define NUM 5

void print_mesg(char *);

int     main(){
    print_mesg("hello");
    print_mesg("world\n");
}

void print_mesg(char *m){
int i;
for (i=0; i<NUM; i++){
    printf("%s", m);
    fflush(stdout);
    sleep(1);
    }
}
```

```
ad@haiku:~/src$ ./print_single
hellohellohellohellohelloworld
world
world
world
world
ad@haiku:~/src$
```

First Effort in Multi-threading

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

#define NUM 5

main()
{   pthread_t t1, t2;

    void *print_mesg(void *);

    pthread_create(&t1, NULL, print_mesg, (void *)"hello ");
    pthread_create(&t2, NULL, print_mesg, (void *)"world\n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}

void *print_mesg(void *m)
{   char *cp = (char *)m;
    int i;

    for (i=0;i<NUM; i++){
        printf("%s", cp);
        fflush(stdout);
        sleep(2);
        }
    return NULL;
}
```

Outcome

```
ad@haiku:~/src$ ./multi_hello
hello world
hello world
hello world
hello world
hello world
ad@haiku:~/src$
```

What is “unexpected” here?

```c
#include <stdio.h>
#include <pthread.h>
#define  NUM  5
int      counter=0;

main(){
    pthread_t t1;
    void *print_count(void *);
    int i;

    pthread_create(&t1, NULL, print_count, NULL);
    for(i=0; i<NUM; i++){
        counter++;
        sleep(1);
    }
    pthread_join(t1,NULL);
}

void *print_count(void *m)
{
    /* counter is a shared variable */
    int i;
    for (i=0;i<NUM;i++){
        printf("count = %d\n",counter);
        sleep(1);
        /*changing this 1 -->> 0 has an effect */
    }
    return NULL;
}
```

The "unexpected" outcome:

```
ad@ad-desktop:~/src$ ./incprint
count = 1
count = 2
count = 3
count = 4
count = 5
ad@ad-desktop:~/src$
```

⊙ Changing *sleep(1)* $\Longrightarrow$ *sleep(0)*:

```
ad@ad-desktop:~/Set007/src$ vi incprint.c
ad@ad-desktop:~/src$ gcc incprint.c -o incprint -lpthread
ad@ad-desktop:~/src$ ./incprint
count = 1
count = 1
count = 1
count = 1
count = 1
ad@ad-desktop:~/Set007/src$
```

$\Rightarrow$ Reading may be inconsistent to what is written to `counter`:
Race Condition?

More problems!

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

int total_words;

int main(int ac, char *av[]){
    pthread_t t1, t2;
    void *count_words(void *);

    if (ac != 3 ) {
        printf("usage: %s file1 file2 \n", av[0]);
        exit(1);
    }
    total_words=0;
    pthread_create(&t1, NULL, count_words, (void *)av[1]);
    pthread_create(&t2, NULL, count_words, (void *)av[2]);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Main thread with ID: %ld reports %5d total words\n",
                pthread_self(), total_words);
}
```

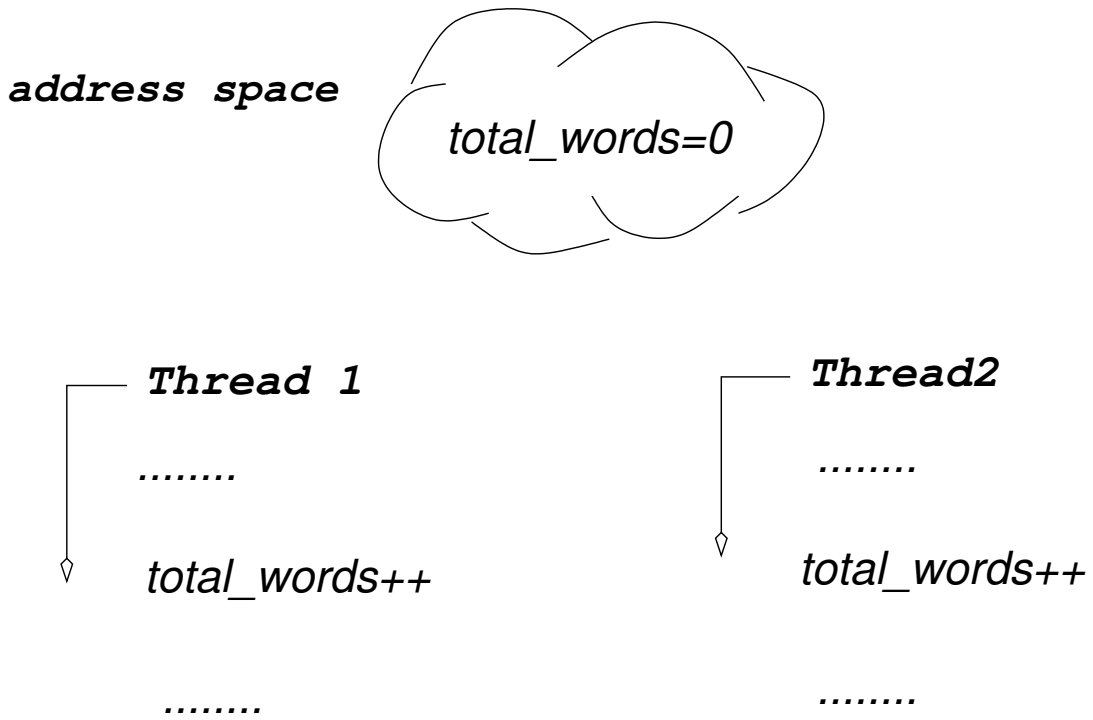
More problems!

```
void *count_words(void *f)
{
    char *filename = (char *)f;
    FILE *fp;    int c, prevc = '\0';
    printf("In thread with ID: %ld counting words.. \n",pthread_self());

    if ( (fp=fopen(filename,"r")) != NULL ){
        while ( ( c = getc(fp) )!= EOF ){
            if ( !isalnum(c) && isalnum(prevc) )
                total_words++;
            prevc = c;
            }
    fclose(fp);
    } else perror(filename);
    return NULL;
}
```

Outcome:

```
ad@haiku:~/src$ !wc
wc -w fileA  fileB
 11136 fileA
  9421 fileB
 20557 total
ad@haiku:~/src$ ./twordcount1 fileB fileA
In thread with ID: 140614526764800 counting words..
In thread with ID: 140614518372096 counting words..
Main thread with ID: 140614535006016 reports 20557 total words
ad@haiku:~/src$ ./twordcount1 fileB fileA
In thread with ID: 140342756800256 counting words..
In thread with ID: 140342748407552 counting words..
Main thread with ID: 140342765041472 reports 20397 total words
ad@haiku:~/src$ ./twordcount1 fileB fileA
In thread with ID: 140211362490112 counting words..
In thread with ID: 140211354097408 counting words..
Main thread with ID: 140211370731328 reports 20414 total words
ad@haiku:~/src$ ./twordcount1 fileB fileA
In thread with ID: 140157487077120 counting words..
In thread with ID: 140157478684416 counting words..
Main thread with ID: 140157495318336 reports 20557 total words
ad@haiku:~/src$ ./twordcount1 fileB fileA
In thread with ID: 139747725231872 counting words..
In thread with ID: 139747716839168 counting words..
Main thread with ID: 139747733473088 reports 20551 total words
ad@haiku:~/src$
```

Race Condition



- *total_words* might NOT have a consistent value after executing the above two (concurrent) assignments.

Binary POSIX Mutexes

- When threads share common structures (resources), the POSIX library offers a simplified version of semaphores termed `binary semaphores` or `mutexes`.
- A binary semaphore can find itself in only two states: *locked* or *unlocked*.
- ```
  int  pthread_mutex_init(pthread_mutex_t  *mutex,
                const  pthread_mutexattr_t *mutexattr)
  ```
  initializes the mutex-object pointed to by `mutex` according to the mutex attributes specified in `mutexattr`.
- A `mutex` may be initialized only once by setting its value by the macro `PTHREAD_MUTEX_INITIALIZER`
  ```
  static pthread_mutex_t mymutex =
      PTHREAD_MUTEX_INITIALIZER;
  ```
- `pthread_mutex_init` always returns 0
  ```

Locking mutexes

- Locking a mutex is carried out by:

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

- If the `mutex` is currently unlocked, it becomes locked and owned by the calling thread, and `pthread_mutex_lock` returns immediately.
- If successful, `pthread_mutex_lock` returns 0.
- If the mutex is already locked by another thread, `pthread_mutex_lock` *blocks* (or "suspends" for the user) the calling thread until the mutex is unlocked.

Unlocking and Destroying mutexes

Unlocking a mutex

- ```
  int pthread_mutex_unlock(pthread_mutex_t *mutex)
  ```
- If the `mutex` has been locked and owned by the calling thread, the `mutex` gets unlocked.
- Upon successful call, it returns `0`.

Destroying a Mutex

- ```
  int pthread_mutex_destroy(pthread_mutex_t *mutex)
  ```
- Destroys the `mutex`, freeing resources it might hold.
- In the LinuxThreads implementation, the call does nothing except checking that `mutex` is unlocked.
- Upon successful call, it returns `0`.

Trying to obtain an lock

Trying to get a lock:

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

- behaves identically to `pthread_mutex_lock`, except that it does not block the calling thread if the `mutex` is already locked by another thread.
- Instead, `pthread_mutex_trylock` returns immediately with the error code `EBUSY`.
- If `pthread_mutex_trylock` returns the code `EINVAL`, the mutex was not initialized properly.
  ```

Addressing the problem

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>
int              total_words;
pthread_mutex_t  counter_lock = PTHREAD_MUTEX_INITIALIZER;

int main(int ac, char *av[])
{   pthread_t t1, t2;
    void *count_words(void *);
    if ( ac != 3 ) {
        printf("usage: %s file1 file2 \n", av[0]);
        exit(1); }
    total_words=0;
    pthread_create(&t1, NULL, count_words, (void *)av[1]);
    pthread_create(&t2, NULL, count_words, (void *)av[2]);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Main thread wirth ID %ld reporting %5d total words\n",
                pthread_self(),total_words);
}
```

Addressing the problem

```
void *count_words(void *f)
{
    char *filename = (char *)f;
    FILE *fp; int c, prevc = '\0';

    if ( (fp=fopen(filename,"r")) != NULL ){
        while ( ( c = getc(fp) )!= EOF ){
            if ( !isalnum(c) && isalnum(prevc) ){
                pthread_mutex_lock(&counter_lock);
                total_words++;
                pthread_mutex_unlock(&counter_lock);
                }
            prevc = c;
            }
        fclose(fp);
    } else perror(filename);
    return NULL;
}
```

Outcome (correct!)

```
ad@gympie:~/src$
ad@gympie:~/src$ wc fileA fileB
   232  11136  64728 fileA
   986   9421  54559 fileB
  1218  20557 119287 total
ad@gympie:~/src$
ad@gympie:~/src$ ./twordcount2  fileA  fileB
Main thread wirth ID 140277266184000 reporting 20557 total words
ad@gympie:~/src$
ad@gympie:~/src$ ./twordcount2  fileA  fileB
Main thread wirth ID 140158499264320 reporting 20557 total words
ad@gympie:~/src$
```

Another Way to Accomplish the Same Correct Operation

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>
#define EXIT_FAILURE 1
void *count_words(void *);
struct arg_set{
    char *fname;
    int  count;
    };

int main(int ac, char *av[]) {
       pthread_t t1, t2;
    struct arg_set args1, args2;
    if ( ac != 3 ) {
          printf("usage: %s file1 file2 \n", av[0]); exit (EXIT_FAILURE); }

    args1.fname = av[1]; args1.count = 0;
    pthread_create(&t1, NULL, count_words, (void *) &args1);
    args2.fname = av[2]; args2.count = 0;
    pthread_create(&t2, NULL, count_words, (void *) &args2);

    pthread_join(t1, NULL); pthread_join(t2, NULL);

    printf("In file  %-10s there are %5d words\n", av[1], args1.count);
    printf("In file  %-10s there are %5d words\n", av[2], args2.count);
    printf("Main thread %ld reporting %5d total words\n",
                pthread_self(), args1.count+args2.count);
}
```

Another Way to Accomplish the Same Correct Operation

```
void *count_words(void *a) {
     struct arg_set *args = a;
     FILE *fp; int c, prevc = '\0';
     printf("Working within Thread with ID %ld and counting\n",pthread_self());

     if ( (fp=fopen(args->fname,"r")) != NULL ){
        while ( ( c = getc(fp) )!= EOF ){
            if ( !isalnum(c) && isalnum(prevc) ){
                args->count++;
                }
            prevc = c;
            }
        fclose(fp);
      } else perror(args->fname);
      return NULL;
}
```

$\Rightarrow$ No `mutex` is used in the above function!

Ourcome:

```
ad@gympie:~/src$
ad@gympie:~/src$ wc fileA fileB
   232  11136  64728 fileA
   986   9421  54559 fileB
  1218  20557 119287 total
ad@gympie:~/src$
ad@gympie:~/src$ ./twordcount3 fileA fileB
Working within Thread with ID 139641419077376 and counting
Working within Thread with ID 139641427470080 and counting
In file  fileA      there are 11136 words
In file  fileB      there are  9421 words
Main thread 139641435739968 reporting 20557 total words
ad@gympie:~/src$
ad@gympie:~/src$ ./twordcount3 fileA fileB
Working within Thread with ID 140256880609024 and counting
Working within Thread with ID 140256889001728 and counting
In file  fileA      there are 11136 words
In file  fileB      there are  9421 words
Main thread 140256897271616 reporting 20557 total words
ad@gympie:~/src$
ad@gympie:~/src$
```

Things to Remember:

- `pthread_mutex_trylock` returns `EBUSY` if the mutex is already locked by another thread.
- Every mutex has to be initialized only once.
- `pthread_mutex_unlock` should be called only by the thread holding the mutex.
- NEVER have `pthread_mutex_lock` called by the thread that has already locked the mutex. A deadlock will occur.
- Should you have `EINVAL` while trying to obtain a lock on a mutex, then the respective initialization has not occurred properly.
- NEVER call `pthread_mutex_destroy` on a locked mutex (`EBUSY`)

Using pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_destroy

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h> /* For threads */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
pthread_mutex_t mtx;          /* Mutex for synchronization */
char buf[25];                 /* Message to communicate */
void *thread_f(void *);       /* Forward declaration */

main() {
   pthread_t thr;
   int err;
   printf("Main Thread %ld running \n",pthread_self());
   pthread_mutex_init(&mtx, NULL);

   if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
      perror2("pthread_mutex_lock", err); exit(1); }
   printf("Thread %ld: Locked the mutex\n", pthread_self());

   if (err = pthread_create(&thr, NULL, thread_f, NULL)) { /* New thread */
      perror2("pthread_create", err); exit(1); }
   printf("Thread %ld: Created thread %ld\n", pthread_self(), thr);

   strcpy(buf, "This is a test message");
   printf("Thread %ld: Wrote message \"%s\" for thread %ld\n",
          pthread_self(), buf, thr);
```

Using pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_destroy

```
    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
       perror2("pthread_mutex_unlock", err); exit(1);
       }
    printf("Thread %ld: Unlocked the mutex\n", pthread_self());

    if (err = pthread_join(thr, NULL)) { /* Wait for thread */
        perror2("pthread_join", err); exit(1); } /* termination */

    printf("Exiting Threads %ld and %ld \n", pthread_self(), thr);

    if (err = pthread_mutex_destroy(&mtx)) { /* Destroy mutex */
       perror2("pthread_mutex_destroy", err); exit(1); }
    pthread_exit(NULL);
}
```

Using pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_destroy

```
void *thread_f(void *argp){ /* Thread function */
    int err;
    printf("Thread %ld: Just started\n", pthread_self());
    printf("Thread %ld: Trying to lock the mutex\n", pthread_self());

    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
       perror2("pthread_mutex_lock", err); exit(1); }

    printf("Thread %ld: Locked the mutex\n", pthread_self());
    printf("Thread %ld: Read message \"%s\"\n", pthread_self(), buf);

    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
       perror2("pthread_mutex_unlock", err); exit(1); }
    printf("Thread %ld: Unlocked the mutex\n", pthread_self());

    pthread_exit(NULL);
}
```

Running the multi-threaded program

```
ad@gympie:~/src$
ad@gympie:~/src$ ./sync_by_mutex
Main Thread 139654610011968 running
Thread 139654610011968: Locked the mutex
Thread 139654610011968: Created thread 139654601742080
Thread 139654610011968: Wrote message "This is only a test message!" for thread
     139654601742080
Thread 139654610011968: Unlocked the mutex
Thread 139654601742080: Just started
Thread 139654601742080: Trying to lock the mutex
Thread 139654601742080: Locked the mutex
Thread 139654601742080: Read message "This is only a test message!"
Thread 139654601742080: Unlocked the mutex
Exiting Threads 139654610011968 and 139654601742080
ad@gympie:~/src$
ad@gympie:~/src$
```

Sum of Squares of n integers using m threads

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
#define LIMITUP 100

pthread_mutex_t mtx;              /* Mutex for synchronization */
int n, nthr, mtxfl;               /* Variables visible by thread function */
double sqsum;                     /* Sum of squares */
void *square_f(void *);           /* Forward declaration */

main(int argc, char *argv[]){
    int err;
    long i;
    pthread_t *tids;
    if (argc > 3) {
        n = atoi(argv[1]);          /* Last integer to be squared */
        nthr = atoi(argv[2]);       /* Number of threads */
        mtxfl = atoi(argv[3]); }    /* with lock (1)? or without lock (0) */
    else exit(0);

    if (nthr > LIMITUP) { /* Avoid too many threads */
        printf("Number of threads should be up to 100\n"); exit(0); }
    if ((tids = malloc(nthr * sizeof(pthread_t))) == NULL) {
        perror("malloc"); exit(1); }

    sqsum = (double) 0.0; /* Initialize sum */
    pthread_mutex_init(&mtx, NULL); /* Initialize mutex */
```

Sum of Squares of n integers using m threads

```
    for (i=0 ; i<nthr ; i++) {
       if (err = pthread_create(tids+i, NULL, square_f, (void *) i)) {
       /* Create a thread */
       perror2("pthread_create", err); exit(1); } }

    for (i=0 ; i<nthr ; i++)
       if (err = pthread_join(*(tids+i), NULL)) {
       /* Wait for thread termination */
           perror2("pthread_join", err); exit(1); }

    if (err = pthread_mutex_destroy(&mtx)) { /* Destroy mutex */
           perror2("pthread_mutex_destroy", err); exit(1); }

    if (!mtxfl) printf("Without mutex\n");
    else printf("With mutex\n");

    printf("%2d threads: sum of squares up to %d is %12.9e\n",nthr,n,sqsum);

    sqsum = (double) 0.0; /* Compute sum with a single thread */
    for (i=0 ; i<n ; i++)
        sqsum += (double) (i+1) * (double) (i+1);
    printf("Single thread: sum of squares up to %d is %12.9e\n", n, sqsum);

    printf("Formula based: sum of squares up to %d is %12.9e\n",
             n, ((double) n)*(((double) n)+1)*(2*((double) n)+1)/6);
    pthread_exit(NULL);
}
```

Sum of Squares of n integers using m threads

```
void *square_f(void *argp){ /* Thread function */
    int err;
    long i, thri;
    thri = (long) argp;

    for (i=thri ; i<n ; i+=nthr) {
        if (mtxfl) /* Is mutex used? */
            if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
                perror2("pthread_mutex_lock", err); exit(1); }

        sqsum += (double) (i+1) * (double) (i+1);

        if (mtxfl) /* Is mutex used? */
            if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
                perror2("pthread_mutex_unlock", err); exit(1); } }
    pthread_exit(NULL);
}
```

Running the program

```
ad@gympie:~/src$
ad@gympie:~/src$ ./sum_of_squares 12345678 99 0
Without mutex
99 threads: sum of squares up to 12345678 is 5.892573212e+19
Single thread: sum of squares up to 12345678 is 6.272253963e+20
Formula based: sum of squares up to 12345678 is 6.272253963e+20
ad@gympie:~/src$
ad@gympie:~/src$ ./sum_of_squares 12345678 99 1
With mutex
99 threads: sum of squares up to 12345678 is 6.272253963e+20
Single thread: sum of squares up to 12345678 is 6.272253963e+20
Formula based: sum of squares up to 12345678 is 6.272253963e+20
ad@gympie:~/src$
```

⊙ Observe the discrepancy in the result when no `mutex` is used.

Condition Variables

- A condition (or "condition variable") is a synchronization means that allows POSIX threads to suspend execution and relinquish the processors until some predicate on the shared data is satisfied.
- The basic operations on conditions are:
  - signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread in execution
  - The waiting lasts until another thread signals (or notifies) the condition.
- A condition variable must always be associated with a mutex to avoid a race condition:
  - This race may occur when a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on the condition variable.

Initializing a Condition Variable (dynamically)

- ```
  int pthread_cond_init(pthread_cond_t *cond,
                        pthread_condattr_t *cond_attr)
  ```

- initializes the condition variable `cond`, using the condition attributes specified in `cond_attr`, or default attributes of `cond_attr` is simply NULL.
- The call always returns 0 upon completion.
- The LinuxThreads implementation supports no attributes for conditions (`cond_attr` is ignored).
- Variables of type `pthread_cond_t` can also be initialized statically, using the constant `PTHREAD_COND_INITIALIZER`.
  ```

Waiting on a condition

- ```
  int pthread_cond_wait(pthread_cond_t *cond,
                        pthread_mutex_t *mutex);
  ```
- atomically unlocks the `mutex` and waits for the condition variable `cond` to be signaled.
- The call always returns 0.
- The thread execution is suspended and does not consume any CPU time until the condition variable is signaled (with the help of a `pthread_cond_signal`).
- Before returning to the calling thread, `pthread_cond_wait` re-acquires `mutex`.
- The signaling thread must acquire the `mutex` before the `pthread_cond_signal` call and unlock it immediately after the call.
  ```

Signaling variables

⇒ Signaling a variable:

- `int pthread_cond_signal(pthread_cond_t *cond)}`
- restarts one of the threads that are waiting on the condition variable `cond`.
- If no threads are waiting on `cond`, nothing happens.
- If several threads are waiting on `cond`, exactly one is restarted.
- The call always returns 0.

Broadcasting to variables

⇒ Broadcasting to a condition variable:

- `int pthread_cond_broadcast(pthread_cond_t *cond)`
- restarts all the threads that are waiting on the condition variable `cond`.
- Nothing happens if no threads are waiting on `cond`.
- The call always returns 0.

Destroying condition variables

- `int pthread_cond_destroy(pthread_cond_t *cond)`
- destroys a condition variable `cond`, freeing the resources it might hold.
- No threads must be waiting on the condition variable on entrance to `pthread_cond_destroy`.
- In the LinuxThreads, the call does nothing except checking that the condition has no waiting threads.
- Upon successful return the call returns 0.
- In case some threads are waiting on `cond`, `pthread_cond_destroy` returns `EBUSY`.

While working with conditions keep in mind:

- For every condition, use a single distinctly-associated with the condition, `mutex`
- Get the `mutex`, before checking of any condition.
- Always use the same `mutex` when changing variables of a condition.
- Keep a `mutex` for the shortest possible time.
- Do not forget to release locks at the end with `pthread_mutex_unlock`.

Using system calls on condition variables

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))

pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t cvar;            /* Condition variable */
char buf[25];                   /* Message to communicate */
void *thread_f(void *);         /* Forward declaration */

main(){
    pthread_t thr; int err;
    pthread_cond_init(&cvar, NULL); /* Initialize condition variable */

    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1); }
    printf("Thread %ld: Locked the mutex\n", pthread_self());

    if (err = pthread_create(&thr, NULL, thread_f, NULL)) { /* New thread */
        perror2("pthread_create", err); exit(1); }
    printf("Thread %ld: Created thread %ld\n", pthread_self(), thr);

    printf("Thread %ld: Waiting for signal\n", pthread_self());

    pthread_cond_wait(&cvar, &mtx); /* Wait for signal */
    printf("Thread %ld: Woke up\n", pthread_self());
    printf("Thread %ld: Read message \"%s\"\n", pthread_self(), buf);
```

Using system calls on condition variables

```
    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
        perror2("pthread_mutex_unlock", err); exit(1); }
    printf("Thread %ld: Unlocked the mutex\n", pthread_self());

    if (err = pthread_join(thr, NULL)) { /* Wait for thread */
        perror2("pthread_join", err); exit(1); } /* termination */
    printf("Thread %ld: Thread %ld exited\n", pthread_self(), thr);

    if (err = pthread_cond_destroy(&cvar)) {
        /* Destroy condition variable */
        perror2("pthread_cond_destroy", err); exit(1); }
    pthread_exit(NULL);
}
```

Using system calls on condition variables

```
void *thread_f(void *argp){ /* Thread function */
    int err;

    printf("Thread %ld: Just started\n", pthread_self());
    printf("Thread %ld: Trying to lock the mutex\n", pthread_self());

    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1); }
    printf("Thread %ld: Locked the mutex\n", pthread_self());

    strcpy(buf, "This is a test message");

    printf("Thread %ld: Wrote message \"%s\"\n", pthread_self(), buf);
    pthread_cond_signal(&cvar); /* Awake other thread */
    printf("Thread %ld: Sent signal\n", pthread_self());

    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
        perror2("pthread_mutex_unlock", err); exit(1); }
    printf("Thread %ld: Unlocked the mutex\n", pthread_self());

    pthread_exit(NULL);
}
```

Using system calls on condition variables

```
ad@gympie:~/src$
ad@gympie:~/src$ ./mutex_condvar
Thread 140117895444288: Locked the mutex
Thread 140117895444288: Created thread 140117887174400
Thread 140117895444288: Waiting for signal
Thread 140117887174400: Just started
Thread 140117887174400: Trying to lock the mutex
Thread 140117887174400: Locked the mutex
Thread 140117887174400: Wrote message "This is a test message"
Thread 140117887174400: Sent signal
Thread 140117887174400: Unlocked the mutex
Thread 140117895444288: Woke up
Thread 140117895444288: Read message "This is a test message"
Thread 140117895444288: Unlocked the mutex
Thread 140117895444288: Thread 140117887174400 exited
ad@gympie:~/src$
ad@gympie:~/src$
```

Another example:

⊙ Three threads increase the value of a global variable while a forth one suspends its operation until a *maximum* value is reached.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))

#define COUNT_PER_THREAD 8        /* Count increments by each thread */
#define THRESHOLD 21              /* Count value to wake up thread */
int count = 0;                     /* The counter */
int thread_ids[4] = {0, 1, 2, 3}; /* My thread ids */

pthread_mutex_t mtx;               /* mutex */
pthread_cond_t cv;                 /* the condition variable */

void *incr(void *argp){
    int i, j, *id = argp;
    int err;
    for (i=0 ; i<COUNT_PER_THREAD ; i++) {
        if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
            perror2("pthread_mutex_lock", err); exit(1); }
        count++; /* Increment counter */
        if (count == THRESHOLD) { /* Check for threshold */
            pthread_cond_signal(&cv); /* Signal suspended thread */
            printf("incr: thread %d, count = %d, threshold reached\n",*id,count)
                ;
            }
```

Code (Cont'ed)

```
        printf("incr: thread %d, count = %d\n", *id, count);
        if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
            perror2("pthread_mutex_unlock", err); exit(1); }
        for (j=0 ; j < 1000000000 ; j++);
     } /* For threads to alternate */
    /* on mutex lock */
    pthread_exit(NULL);
}

void *susp(void *argp){
    int err, *id = argp;
    printf("susp: thread %d started\n", *id);
    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1);
        }
    if (count < THRESHOLD) { /* If threshold not reached */
        pthread_cond_wait(&cv, &mtx); /* suspend */
        printf("susp: thread %d, signal received\n", *id);
        }
    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
        perror2("pthread_mutex_unlock", err); exit(1);
        }
    pthread_exit(NULL);
}
```

Code (Cont'ed)

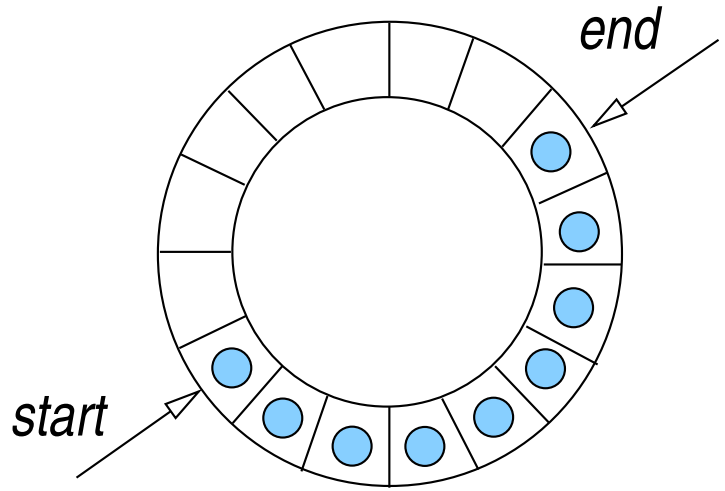
```
main() {
    int i, err;
    pthread_t threads[4];

    pthread_mutex_init(&mtx, NULL); /* Initialize mutex */
    pthread_cond_init(&cv, NULL); /* and condition variable */
    for (i=0 ; i<3 ; i++)
        if (err = pthread_create(&threads[i], NULL, incr,(void *) &thread_ids[i
            ])) { perror2("pthread_create", err); exit(1); /* Create threads 0,
             1, 2 */
           }

    if (err = pthread_create(&threads[3], NULL, susp, (void *) &thread_ids[3]))
        { perror2("pthread_create", err); exit(1); } /* Create thread 3 */

    for (i=0 ; i<4 ; i++)
        if (err = pthread_join(threads[i], NULL)) {
            perror2("pthread_join", err); exit(1);
            };
    /* Wait for threads termination */
    printf("main: all threads terminated\n");
    /* Destroy mutex and condition variable */
    if (err = pthread_mutex_destroy(&mtx)) {
        perror2("pthread_mutex_destroy", err); exit(1); }
    if (err = pthread_cond_destroy(&cv)) {
        perror2("pthread_cond_destroy", err); exit(1); }
    pthread_exit(NULL);
}
```

Outcome:

```
ad@gympie:~/src$ ./counter
incr: thread 1, count = 1
susp: thread 3 started
susp: thread 3 about to suspend
incr: thread 0, count = 2
incr: thread 2, count = 3
incr: thread 1, count = 4
incr: thread 2, count = 5
incr: thread 0, count = 6
incr: thread 1, count = 7
incr: thread 2, count = 8
incr: thread 0, count = 9
incr: thread 1, count = 10
incr: thread 2, count = 11
incr: thread 1, count = 12
incr: thread 0, count = 13
incr: thread 1, count = 14
incr: thread 2, count = 15
incr: thread 0, count = 16
incr: thread 1, count = 17
incr: thread 2, count = 18
incr: thread 0, count = 19
incr: thread 1, count = 20
incr: thread 2, count = 21, threshold reached
incr: thread 2, count = 21
susp: thread 3, signal received
incr: thread 0, count = 22
incr: thread 2, count = 23
incr: thread 0, count = 24
main: all threads terminated
ad@gympie:~/src$
```

The Producer–Consumer Synchronization Problem

- There is one producer and one consumer.
- The producer may produce upto a *maximum* number of goods.
- An item cannot be consumed if the producer has not successfully completed its placement on the buffer.
- If no items exist on the buffer, the consumer has to wait.
- if the buffer is full, the producer has to wait.



A solution for the “bounded buffer” problem

```
#include <stdio.h>                  // from www.mario-konrad.ch
#include <pthread.h>
#include <unistd.h>

#define POOL_SIZE 6

typedef struct {
        int data[POOL_SIZE];
        int start;
        int end;
        int count;
} pool_t;

int num_of_items = 15;

pthread_mutex_t mtx;
pthread_cond_t cond_nonempty;
pthread_cond_t cond_nonfull;
pool_t pool;

void initialize(pool_t * pool) {
        pool->start = 0;
        pool->end = -1;
        pool->count = 0;
}
```

```
void place(pool_t * pool, int data) {
        pthread_mutex_lock(&mtx);
        while (pool->count >= POOL_SIZE) {
                printf(">> Found Buffer Full \n");
                pthread_cond_wait(&cond_nonfull, &mtx);
                }
        pool->end = (pool->end + 1) % POOL_SIZE;
        pool->data[pool->end] = data;
        pool->count++;
        pthread_mutex_unlock(&mtx);
}


int obtain(pool_t * pool) {
        int data = 0;
        pthread_mutex_lock(&mtx);
        while (pool->count <= 0) {
                printf(">> Found Buffer Empty \n");
                pthread_cond_wait(&cond_nonempty, &mtx);
                }
        data = pool->data[pool->start];
        pool->start = (pool->start + 1) % POOL_SIZE;
        pool->count--;
        pthread_mutex_unlock(&mtx);
        return data;
}
```

```
void * producer(void * ptr)
{
        while (num_of_items > 0) {
                place(&pool, num_of_items);
                printf("producer: %d\n", num_of_items);
                num_of_items--;
                pthread_cond_signal(&cond_nonempty);
                usleep(0);
                }
        pthread_exit(0);
}

void * consumer(void * ptr)
{
        while (num_of_items > 0 || pool.count > 0) {
                printf("consumer: %d\n", obtain(&pool));
                pthread_cond_signal(&cond_nonfull);
                usleep(500000);
                }
        pthread_exit(0);
}
```

```
int main(int argc, char ** argv)
{
        pthread_t cons, prod;

        initialize(&pool);
        pthread_mutex_init(&mtx, 0);
        pthread_cond_init(&cond_nonempty, 0);
        pthread_cond_init(&cond_nonfull, 0);
        pthread_create(&cons, 0, consumer, 0);
        pthread_create(&prod, 0, producer, 0);
        pthread_join(prod, 0);
        pthread_join(cons, 0);
        pthread_cond_destroy(&cond_nonempty);
        pthread_cond_destroy(&cond_nonfull);
        pthread_mutex_destroy(&mtx);
        return 0;
}
```

⇒ Outcome:

```
ad@ad-desktop:~/Set007/src$ ./prod-cons
>> Found Buffer Empty
producer: 15
consumer: 15
producer: 14
producer: 13
producer: 12
producer: 11
producer: 10
producer: 9
>> Found Buffer Full
```

```
consumer: 14
producer: 8
>> Found Buffer Full
consumer: 13
producer: 7
>> Found Buffer Full
consumer: 12
producer: 6
>> Found Buffer Full
consumer: 11
producer: 5
>> Found Buffer Full
consumer: 10
producer: 4
>> Found Buffer Full
consumer: 9
producer: 3
>> Found Buffer Full
consumer: 8
producer: 2
>> Found Buffer Full
consumer: 7
producer: 1
consumer: 6
consumer: 5
consumer: 4
consumer: 3
consumer: 2
consumer: 1
ad@ad-desktop:~/Set007/src$
```

Thread Safety

• Problem: a thread may call library functions that are not thread-safe creating spurious outcomes.

- A function is "thread-safe," if multiple threads can simultaneously execute invocations of the same function without *side-effects* (or intereferences of any type!).
- POSIX specifies that all functions (including all those from the Standard C Library) except those (next slide) are implemented in a thread-safe manner.
- Directive: the calls of the table (next slide) *should* thread-safe implentations denoted with the postfix *_r*.

System calls not required to be thread-safe

asctime	*basename*	*catgets*	*crypt*	*ctime*
dbm_clearerr	*dbm_close*	*dbm_delete*	*dbm_error*	*dbm_fetch*
dbm_firstkey	*dbm_nextkey*	*dbm_open*	*dbm_store*	*dirname*
dlerror	*drand48*	*ecvt*	*encrypt*	*endgrent*
endpwent	*endutxent*	*fcvt*	*ftw*	*gcvt*
getc_unlocked	*getchar_unlocked*	*getdate*	*getenv*	*getgrent*
getgrgid	*getgrname*	*gethostbyaddr*	*gethostbyname*	*getlogin*
getnetbyaddr	*getnetbyname*	*getnetent*	*getopt*	*getprotobyname*
getprotobynumber	*getprotoend*	*getpwent*	*getopwnam*	*getpwuid*
getservbyname	*getservbyport*	*getservent*	*getutxent*	*getutxid*
getutxline	*gmtime*	*hcreate*	*hdestroy*	*hsearch*
inet_ntoa	*l64a*	*lgamma*	*lgammaf*	*lgammal*
localeconv	*localtime*	*lrand48*	*mrand48*	*nftw*
nl_langinfo	*ptsname*	*putc_unlocked*	*putchar_unlocked*	*putenv*
pututxline	*rand*	*readdir*	*setenv*	*setgrent*
setkey	*setpwent*	*setuxent*	*strerror*	*strtok*
ttyname	*unsetenv*	*wcstombs*	*wctomb*	

◇ An easy ("dirty") way to safely use the above calls with threads is to invoke them in conjunction with *mutexes* (i.e., in mutually exclusive fashion).