Threads

Stefan D. Bruda

CS 464/564, Fall 2023

Threads vs. processes

- We have seen how to use concurrent processes, with one thread of execution each
- Concurrency can be also implemented using one process with multiple threads of execution
  - Multiple processes with multiple threads of execution each are of course possible as well
- Threads (sometimes called "light processes") behave similar to processes, in the sense that they execute concurrently
  - However, threads share most of their memory space with each other, including the process' descriptor table
- In Linux you can create something similar with threads (but considerably more robust) using `clone(2)`
  - However, `clone(2)` is not portable (not even to other Unices), so the POSIX standard is usually preferred
  - In Linux the POSIX threads are implemented as a relatively thin layer over the `clone(2)` and related API

Posix threads

- Linux threads follow the POSIX standard 1003.1, which is observed by many other Unix systems
- Features:
  - Threads can be created at any time using the system call `pthread_create`
  - Threads execute concurrently, and are preemptible (one thread cannot block the CPU)
    - A thread can give up the CPU voluntarily by using the system call `sched_yield` (also available for processes)
  - Each thread has its own stack (local variables), but all threads in a process share the rest of the address space (global variables, descriptor table, heap, ...)
  - The threads API include functions for coordination and synchronization (including mechanisms to implement critical regions in memory, i.e., without file locks)
- A program that uses threads must include `<pthread.h>` and must be linked with the library `pthread`, i.e.,

```
g++ -lpthread -o tserv tserv.o tcp-utils.o
g++ -pthread -o tserv tserv.o tcp-utils.o
```


Threads vs. processes

Advantages of threads:

- Efficiency: context switching between threads is generally (though not always) faster than between processes
- The existence of shared memory: threads can communicate between each other using the shared memory, as opposed to processes
  - The implementation of critical regions does not need to use locks on files
  - Monitoring is also easy to implement

Disadvantages of threads:

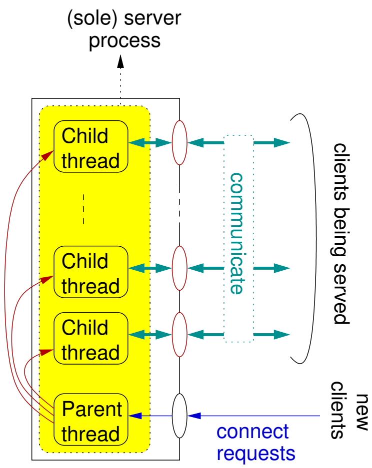
- The existence of shared memory: two threads may interfere with each other when both try to access shared objects (e.g., the same global variable) = interference
- Lack of robustness: if a thread performs an illegal operation (e.g., a segmentation violation) the whole process is terminated
- System calls may not be thread safe
  - Annoyingly, thread safety is not always documented
  - If in doubt, put the respective system call in a critical region (discussed later)

Caveats

- Do not abuse critical regions
  - You have a very good chance to unboundedly decrease response time
  - In particular, a `read`/`recv` in a critical region can easily deadlock a server (so don't ever do it!)
  - Critical regions and signal handlers do not mix well
- File and socket descriptors are shared
  - Once a thread opens a file/socket, it is opened for all the threads
  - Most importantly, once a thread closes a descriptor, no other thread can access that descriptor successfully
- If a thread calls `exit` then the whole process terminates
  - A thread terminates itself when its top-level function returns, or explicitly by calling `pthread_exit`

Concurrent servers, revisited

1. create, bind and place in passive mode the master socket
2. repeat forever:
   1. accept the next connection request from the socket and create a new slave socket *s* for the connection.
   2. pthread_create; in the new thread:
      1. do not close master socket
      2. read a request from the client
      3. serve the request and reply
      4. if finished with the client, close *s* and terminate; otherwise, repeat from 2
   3. do not close slave socket



Coordination and Synchronization

- When working with processes, you generally need to wory about exclusive access only when accessing the file system or pipes
- When using threads memory space is also shared, so we also need to worry about memory access
- The following mechanisms for coordination and synchronization are available:

- Mutex: Used to provide exclusive access to a shared piece of data
  - More generally, you can use a mutex to implement a critical regions

Operation	System call	File lock equivlent
Initialization	`pthread_mutex_init`	opening the lock file
Enter critical region	`pthread_mutex_lock`	`enter_critical`
Release c. r.	`pthread_mutex_unlock`	`exit_critical`
Test for availability	`pthread_mutex_trylock`	

Coordination and Synchronization (cont'd)

- (Counting) semaphore: Like a mutex, but for *n* copies of the resource

Instead of:	Use:
`pthread_mutex_init`	`sem_init`
`pthread_mutex_lock`	`sem_wait`
`pthread_mutex_unlock`	`sem_post`
`pthread_mutex_trylock`	`sem_trywait`
	`sem_getvalue`

  - Include `<semaphore.h>` to work with semaphores

- Condition variable = mutex + condition
  - A number of threads need to access a critical region (mutex)
  - Once the critical region is acquired, a certain condition has to be met before going any further
  - While it waits for the condition, a thread gives up the mutex so that other threads may proceed
  - Not using condition variables when appropriate will result in either busy-waiting loops or poor responsiveness

Condition variable (example)

- Initialization:

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Wait for x to become larger than y:

```
pthread_mutex_lock(&mut);
while (x <= y) {  pthread_cond_wait(&cond, &mut);  }
                 /* mut is released while waiting */
/* mut is reacquired */
/* do stuff with x and y */
pthread_mutex_unlock(&mut);
```

- When x becomes larger than y, the corresponding condition should be signalled:

```
pthread_mutex_lock(&mut);
/* code that changes x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```


Coding examples: Mutex

```
#include <pthread.h>


// lock1, lock2 MUST be global
pthread_mutex_t lock1;
pthread_mutex_t lock2;

pthread_mutex_init(&lock1,NULL);
pthread_mutex_init(&lock2,NULL);


// Do something involving two
// critical regions, i.e. use
//   pthread_mutex_lock(&lock1)
//   pthread_mutex_unlock(&lock1)
//   pthread_mutex_lock(&lock2)
//   pthread_mutex_unlock(&lock2)

// clean up:
// nothing to do
// (could call
//    pthread_mutex_destroy
//  except that it does nothing)
```

```
char lock1name[256], lock2name[256];
snprintf(lock1name,255,...);
snprintf(lock2name,255,...);
// lock1, lock2 can be local
int lock1 = open(lock1name,...);
int lock2 = open(lock2name,...);

if (lock1 == -1 || lock2 == -1) {
  perror("Cannot create locks");
  return 1;                          }

// Do something involving two
// critical regions, i.e. use
//   enter_critical(lock1)
//   exit_critical(lock1)
//   enter_critical(lock2)
//   exit_critical(lock2)

// clean up
close(lock1);
close(lock2);
unlink(lock1name);
unlink(lock2name);
```


Coding examples: Mutex and threads

```
pthread_mutex_t lock1, lock2;

void* do_lock (int n) {
  pthread_mutex_lock(&lock1);
  cout << "Thread " << n << " enters critical.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock1);
  cout << "Thread " << n << " exits critical.\n";
  return NULL;
}
int main () {
  pthread_mutex_init(&lock1,NULL);
  pthread_mutex_init(&lock2,NULL);

  pthread_t tt;
  pthread_attr_t ta;
  pthread_attr_init(&ta);
  pthread_attr_setdetachstate(&ta,PTHREAD_CREATE_DETACHED);

  pthread_create(&tt, &ta, (void* (*) (void*))do_lock, (void*)1);
  pthread_create(&tt, &ta, (void* (*) (void*))do_lock, (void*)2);
  pthread_create(&tt, &ta, (void* (*) (void*))do_lock, (void*)3);
  sched_yield(); sleep(60);
}
```


Mutex and deadlocks

```
void* do_lock_21 (int n) {
  pthread_mutex_lock(&lock2);
  cout<<"Th. "<<n<<" enters 1.\n";
  sched_yield(); sleep(1);
  pthread_mutex_lock(&lock1);
  cout<<"Th. "<<n<<" enters 2.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock2);
  cout<<"Th. "<<n<<" exits 2.\n";
  pthread_mutex_unlock(&lock1);
  cout<<"Th. "<<n<<" exits 1.\n";
  return NULL;
}
```

```
void* do_lock_12 (int n) {
  pthread_mutex_lock(&lock1);
  cout<<"Th. "<<n<<" enters 1.\n";
  sched_yield(); sleep(1);
  pthread_mutex_lock(&lock2);
  cout<<"Th. "<<n<<" enters 2.\n";
  sched_yield(); sleep(3);
  pthread_mutex_unlock(&lock2);
  cout<<"Th. "<<n<<" exits 2.\n";
  pthread_mutex_unlock(&lock1);
  cout<<"Th. "<<n<<" exits 1.\n";
  return NULL;
}
```

```
int main () {
  [ ... initialize mutexes, thread data ... ]

  pthread_create(&tt, &ta,
      (void* (*) (void*))do_lock_12, (void*)1);
  pthread_create(&tt, &ta,
      (void* (*) (void*))do_lock_21, (void*)2);
  sched_yield(); sleep(60);
}
```

Output:

```
Th. 1 enters 1.
Th. 2 enters 1.
```

. . . nothing happens in the next minute!

Terminating a thread

- A thread can terminate itself by returning from its main function of by calling `pthread_exit`
- A thread can cancel (i.e., terminate) other threads by sending a cancellation request using `pthread_cancel`
  - Sole argument: the thread being cancelled (`pthread_t`)
  - Depending on its settings, the target thread can ignore the request, honor it immediately, or defer it until it reaches a cancellation point
    - The following POSIX threads functions are cancellation points: `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_testcancel`, `sem_wait`, `sigwait`
    - All other POSIX threads functions are guaranteed not to be cancellation points
    - `pthread_testcancel` does nothing except testing for pending cancellation and executing it if applicable
  - When the cancellation is honored the thread being cancelled behaves as if it calls `pthread_exit(PTHREAD_CANCELED)`

Cancellation points

- In addition to the cancellation points enumerated above, a number of system calls (basically, all system calls that may block) are cancellation points
  - And so are the library functions that use these system calls
- Older implementations may not conform to this even if hey call themselves POSIX compliant
- Workaround:
  - Cancellation requests are transmitted to the target thread through signals
  - The signal will interrupt all blocking system calls, causing them to return immediately with the `EINTR` error
  - Using `pthread_cancel` immediately after a system call is thus safe and acheives the desired effect
  - It is unclear what is the behaviour of newer implementations (feel free to experiment)

Cancellation state

- `pthread_setcancelstate` changes the cancellation state for the calling thread
  - That is, whether cancellation requests are ignored or not (possible state values: `PTHREAD_CANCEL_DISABLE`, `PTHREAD_CANCEL_ENABLE`)
  - The old cancellation state is stored and can thus be restored (unless the second argument is 0)
  - Prototype: `pthread_setcancelstate(int state, int *oldstate);`
- `pthread_setcanceltype` changes the type of responses to cancellation requests
  - Possible behaviour: asynchronous (immediate) or deferred cancellation (`PTHREAD_CANCEL_ASYNCHRONOUS`, `PTHREAD_CANCEL_DEFERRED`)
  - The old cancellation type is stored and can thus be restored (unless the second argument is 0)
  - Prototype: `int pthread_setcanceltype(int type, int *oldtype);`
- A thread is created by default with cancellation enabled and deferred

Joining and detaching

- A thread can wait for the completion of other threads:

```
                                  void* ret;
pthread_create(&tt, ...);   ⇝    pthread_join(tt, &ret);
```

  - `pthread_join` suspends execution of the calling thread until the thread given as argument terminates
  - the return value of the thread (`PTHREAD_CANCELED` if cancelled) is stored in the second argument unless the second argument is 0
  - At most one thread can wait for the termination of any given thread
- A thread can be waited upon ("joined") only if it is attached
- However, if a thread is attached it does not release any of its resources unless a `pthread_join` is called on it
  - Similar with zombie processes
  - If you do not want/need to deal with "zombie threads" then you can set them to be detached; otherwise you must call `pthread_join` on them

Monitoring a server

- Gathering statistics on server usage is easy in a multithreaded environment, because of the global variables that are accessible from all the threads:
  - We build a structure with statistical data of interest
  - We create a monitor thread that will from time to time process the statistical data and store the result (write it in a log file, etc.)
  - The other threads update this structure according to what they did
  - Since the structure is used by all the running threads, we have to put all the accesses to it in critical regions