Concurrent Threads

Dr. Bystrov

School of Engineering

Newcastle University

Aims

- Threads (underlined topics examined)
  - Concept
  - Modelling
  - Examples
- Shared data
  - Mutex
  - Condition variables
  - Semaphores

Threads

Read YoLinux Tutorial: POSIX thread (pthread) libraries

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

- The POSIX threads standard API for C/C++.
- Spawning a concurrent process flow.
- Most effective on multiprocessor systems;
  - gaining speed through parallel processing.
- Less overhead than "forking" or spawning a new process
  - the system does not initialise a new system virtual memory space and environment
  - threads within a process share the same address space.

Thread Basics

- Thread operations include
  - thread creation,
  - termination,
  - synchronisation (joins,blocking),
  - scheduling,
  - data management and
  - process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- pthread functions return "0" if OK.

Threads share

Threads in the same process share:

- address space;
- process instructions;
- most data;
- open files (descriptors);
- signals and signal handlers;
- current working directory;
- user and group id.

Exiting the process ends all its threads!

Threads don’t share

Each thread has a unique:

- thread ID;
- set of registers, stack pointer;
- stack for local variables, return addresses;
- signal mask;
- priority;
- return value: `errno`.

Thread forking

```
#include <pthread.h>
int pthread_create (
    pthread_t * thread,
    pthread_attr_t * attr,
    void * (*start_routine)(void *),
    void * arg);
```

On success, the identifier of the newly created thread is stored in the location pointed by the thread argument, and a 0 is returned. On error, a non-zero error code is returned.

Attributes (attr):

- detachable or joinable;
- scheduler control.

Thread joining

```
#include <pthread.h>
int pthread_join(
    pthread_t thread, void **retval);
```

`pthread_join` suspends the execution of the calling thread until the thread identified by `th` terminates, either by calling `pthread_exit` or by being cancelled.

The joined thread `th` must be in the joinable state: it must not have been detached.

`pthread_join` must be called once for each joinable thread created to avoid memory leaks.

At most one thread can wait for the termination of a given thread.

Fork-join example

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

void thread_func (void *ptr)
{
  printf ("%s \n", (char *) ptr);
}

int main ()
{
  pthread_t t1, t2;
  char *msg1 = "Thread 1", *msg2 = "Thread 2";

  if (pthread_create (&t1, NULL, (void *) thread_func, (void *) msg1))
    exit(1);
  if (pthread_create (&t2, NULL, (void *) thread_func, (void *) msg2))
    exit(1);

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

  printf ("Threads finished\n");
}
```

Thread synchronisation

The threads library provides three synchronisation mechanisms (<u>we focus on the first two</u>):

join – Make a thread wait until others are completed (illustrated earlier).

mutex – Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

condition variables – data type `pthread_cond_t`

Mutexes

- Mutexes are used to prevent data inconsistencies due to race conditions.
- A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.
- Mutexes are used for serialising shared resources.
- Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.
- One can apply a mutex to protect a segment of memory ("critical region") from other threads.

Mutex example

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

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void thread_func (void *ptr)
{
  int i;
  for (i=0;i<100000;i++)
    {
      pthread_mutex_lock (&mutex1);
      counter++;
      pthread_mutex_unlock (&mutex1);
    }
  printf ("Thread %s, counter = %d \n", ptr, counter);
}

int main ()
{
  pthread_t t1, t2;
  char *msg1 = "Thread 1";
  char *msg2 = "Thread 2";

  pthread_create (&t1, NULL, (void *) &thread_func,  (void *) msg1);
  pthread_create (&t2, NULL, (void *) &thread_func,  (void *) msg2);

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

Experiments on fairness

Run two threads with the following thread function. Distinguish between them by passing them different parameters.

```
void
thread_func (void *ptr)
{
   int i;
   for (i = 0; i < 10; i++)
     {
       pthread_mutex_lock (&mutex1);
       printf ("%d", *(int*)ptr);
       pthread_mutex_unlock (&mutex1);
/*     system("sleep 0.1"); */
     }
}
```

Output: 11111111112222222222
Uncomment the delay.
Output: 12121212121212121212

Condition variables

- A condition variable is used with the appropriate functions for waiting and later, process continuation.

- A condition variable must always be associated with a mutex.
  - A deadlock when one thread is preparing to wait and another thread signals the condition. The thread will be perpetually waiting for a signal that is never sent.
  - Any mutex can be used, there is no explicit link between the mutex and the condition variable.

Functions used with condition variables

- Creating / Destroying:
  - `pthread_cond_init`
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
  - `pthread_cond_destroy`
- Waiting on condition:
  - `pthread_cond_wait`
  - `pthread_cond_timedwait` – bloking time limit
- Waking thread based on condition:
  - `pthread_cond_signal`
  - `pthread_cond_broadcast` – wake up all threads blocked by the condition variable.

Condition wait example

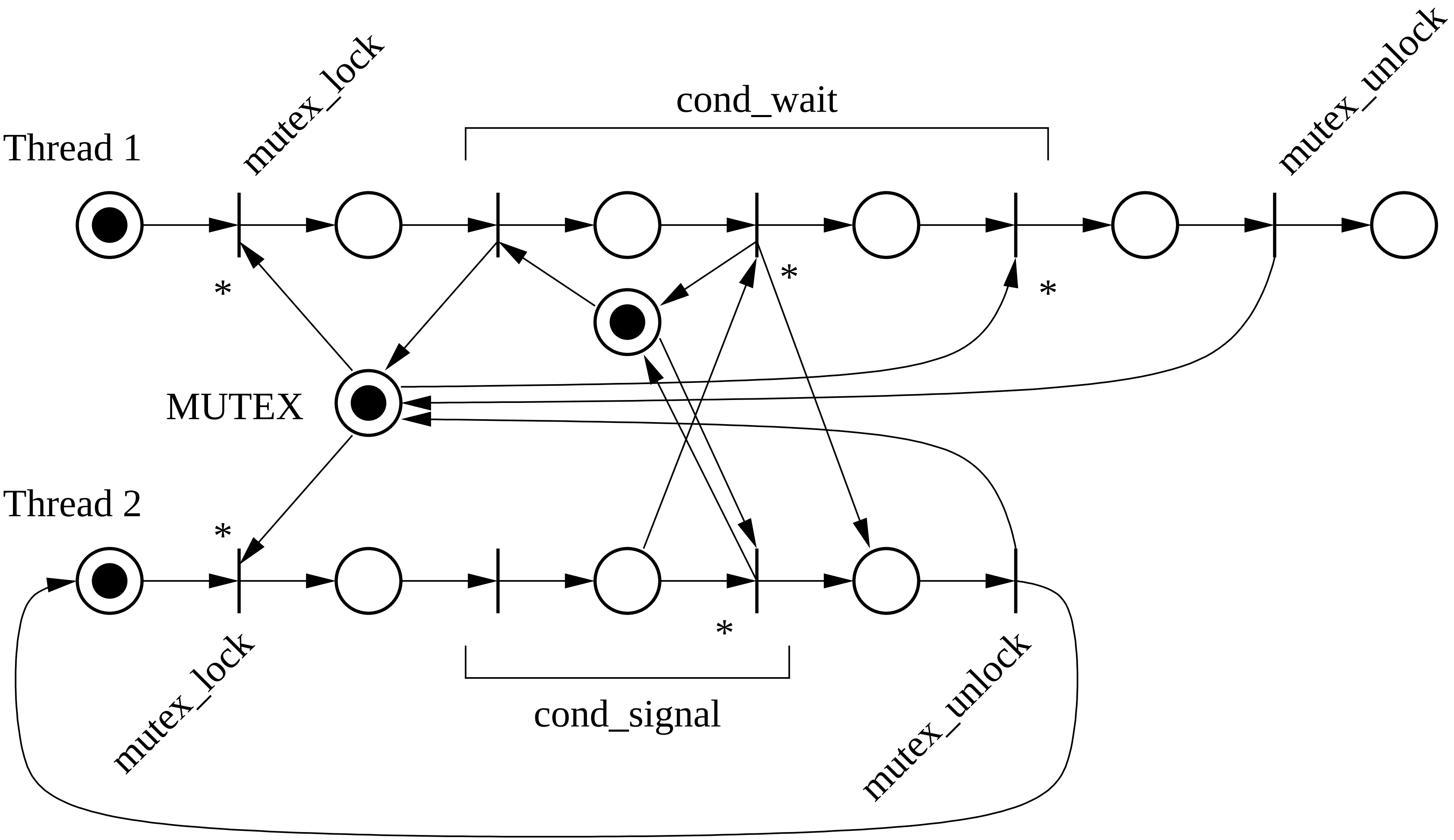
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER;

void *t_func1 ()
{
  system ("sleep 1");
  pthread_mutex_lock (&condition_mutex);
  printf ("thread 1: waiting started\n");
  pthread_cond_wait (&condition_cond, &condition_mutex);
  printf ("thread 1: waiting finished\n");
  pthread_mutex_unlock (&condition_mutex);
}

void *t_func2 ()
{
  system ("sleep 2"); // 0 - deadlock, 2 - no deadlock
  printf ("thread 2: locking mutex\n");
  pthread_mutex_lock (&condition_mutex);
  printf ("thread 2: signal\n");
  pthread_cond_signal (&condition_cond);
  pthread_mutex_unlock (&condition_mutex);
  printf ("thread 2: mutex unlocked\n");
}
```

Condition wait model



Semaphores

- Built upon mutexes and condition variables.
- An integer value is associated with a semaphore.

Semaphore operations:

- `semaphore_init` – create a semaphore with value 1
- `semaphore_up` – increments the semaphore
- `semaphore_down` – blocks if its value ≤ 0, decrements the semaphore value
- `semaphore_decrement` – decrements without blocking
- `semaphore_destroy` – destroys a semaphore

Semaphore example

```
char buffer;
Semaphore writers_turn, readers_turn;

void writer_function(void)
{
  while(1)
  {
    semaphore_down( &writers_turn );
    buffer = produce_new_item();
    semaphore_up( &readers_turn );
  }
}

void reader_function(void)
{
  while(1)
  {
    semaphore_down( &readers_turn );
    consume_item();
    semaphore_up( &writers_turn );
  }
}
```

Conclusions

- Fork-join, choice-merge, arbitration when accessing common resource
- Two types of dependency between threads
  - imposed by start-termination control
  - imposed by data communication
- Modelling concurrent computations with Petri nets
- Interesting effects – counting errors, fairness, deadlock