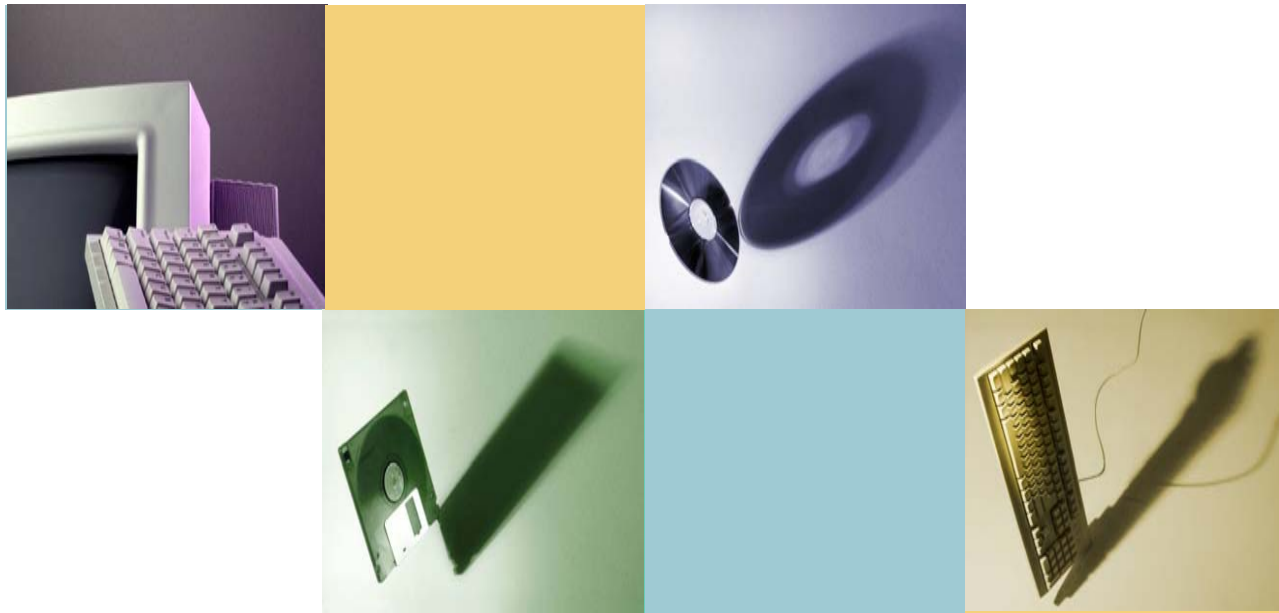


PTHREADS



CSC 469/557 – High Availability and Performance Computing

This powerpoint is based on YoLinux Tutorial: POSIX thread (pthread) libraries,
<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

The POSIX thread (pthread) libraries

- standards based thread API for C/C++.
- spawn a new concurrent process flow.
- the process flow can be scheduled to run on another processor
- Less overhead than "forking"
- Share the same process address space – no initialization for a new system virtual memory space.
- Also gain performance on uniprocessor since one thread may execute while another is waiting for I/O
- Threads are only for a single computer system. MPI processes are across multiple machines or a distributed computing environment.



Basics

- Thread operations
 - thread creation
 - termination,
 - synchronization (joins, blocking), scheduling,
 - data management
 - process interaction.



- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
 - Process instructions
 - Most data
 - open files
 - signals and signal handlers
 - current working directory
 - User and group id
- Each thread has a unique:
 - Thread ID
 - set of registers, stack pointer
 - stack for local variables, return addresses
 - signal mask
 - priority
 - Return value: errno
- pthread functions return "0" if ok.



- Create an independent thread

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

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

- Normally, we wait till the created thread finishes before the main thread continues

```
int pthread_join(pthread_t th, void **thread_return);
```

- suspends the execution of the calling thread until the thread identified by *th* terminates



See example `pt1.c` in `~box/directory`

- **Compile:**

- `gcc -lpthread pt1.c`
or
- `g++ -lpthread pt1.c`

```
[box@oscar box]$ gcc -lpthread pt1.c
```

```
[box@oscar box]$ ./a.out
```

```
Thread 1
```

```
Thread 2
```

```
Thread 1 returns: 0
```

```
Thread 2 returns: 0
```

- [



From example

- Threads terminate by
 - just returning from the function or
 - explicitly calling `pthread_exit`
 - by a call to the function `exit` which will terminate the process including any threads.
- `int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg);`
- `thread` - returns the thread id.
- `attr` - Set to `NULL` if default thread attributes are used. (else define members of the struct `pthread_attr_t` defined in `bits/pthreadtypes.h`) Attributes include:
 - detached state (joinable? Default: `PTHREAD_CREATE_JOINABLE`. Other option: `PTHREAD_CREATE_DETACHED`)
 - scheduling policy (real-time? `PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED, SCHED_OTHER`)
 - scheduling parameter
 - `inheritsched` attribute (Default: `PTHREAD_EXPLICIT_SCHED` Inherit from parent thread: `PTHREAD_INHERIT_SCHED`)
 - scope (Kernel threads: `PTHREAD_SCOPE_SYSTEM` User threads: `PTHREAD_SCOPE_PROCESS` Pick one or the other not both.)
 - guard size
 - stack address (See `unistd.h` and `bits/posix_opt.h` `_POSIX_THREAD_ATTR_STACKADDR`)
 - stack size (default minimum `PTHREAD_STACK_SIZE` set in `pthread.h`),



From example

- Threads terminate by
 - just returning from the function or
 - explicitly calling `pthread_exit`
 - by a call to the function `exit` which will terminate the process including any threads.
- `int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg);`
- `thread` - returns the thread id. (unsigned long int defined in `bits/pthreadtypes.h`)
- `attr` - Set to `NULL` if default thread attributes are used. (else define members of the struct `pthread_attr_t` defined in `bits/pthreadtypes.h`) Attributes include:
 - detached state (joinable? Default: `PTHREAD_CREATE_JOINABLE`. Other option: `PTHREAD_CREATE_DETACHED`)
 - scheduling policy (real-time? `PTHREAD_INHERIT_SCHED`, `PTHREAD_EXPLICIT_SCHED`, `SCHED_OTHER`)
 - scheduling parameter
 - inheritsched attribute (Default: `PTHREAD_EXPLICIT_SCHED` Inherit from parent thread: `PTHREAD_INHERIT_SCHED`)
 - scope (Kernel threads: `PTHREAD_SCOPE_SYSTEM` User threads: `PTHREAD_SCOPE_PROCESS` Pick one or the other not both.)
 - guard size
 - stack address (See `unistd.h` and `bits/posix_opt.h` `_POSIX_THREAD_ATTR_STACKADDR`)
 - stack size (default minimum `PTHREAD_STACK_SIZE` set in `pthread.h`),
- `void * (*start_routine)` - pointer to the function to be threaded. Function has a single argument: pointer to void.
- `*arg` - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.



synchronization

- Pthread provides three synchronization mechanisms:
 1. mutex - Mutual exclusion lock is a blocking access to prevent racing condition. It enforces exclusive access by a thread to a variable or set of variables.
 2. join - Make a thread wait till others are complete (terminated).
 3. condition variables - data type `pthread_cond_t`



MUTEX

- A race condition often occurs when two or more threads competing on the same memory area
- Results of computations depends on the order in which the operations are executed.
- We can use mutex to access a critical section



Example w/o mutex

```
int counter=0;
/* Function C */
void functionC() {
    counter++
}
```



Example with mutex

```
pthread_mutex_t mutex1 =  
    PTHREAD_MUTEX_INITIALIZER;  
int counter=0;  
/* Function C */  
void functionC() {  
    pthread_mutex_lock( &mutex1 );  
    counter++;  
    pthread_mutex_unlock( &mutex1 );  
}
```



What could possibly be the problem?

- w/o mutex – answers are unpredictable
- With mutex- we can protect the critical section and allow only one thread in CS at a time.
- The statement “**count++**” for thread1 in machine language as:

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- The statement “**count++**” for thread2 implemented as:

```
register2 = counter  
register2 = register2 + 1  
counter = register2
```

- Show pt2.c



Join

- A function to wait for the completion of the threads with a join.
- A thread calling routine may launch multiple threads then wait for them to finish to get the results.
- Show pt3.c



pthread_cond_t

- A condition variable is used with the appropriate functions for waiting and later, process continuation.
- allows threads to suspend execution and give up the processor until a given condition is true.
- must always be associated with a mutex to avoid a race condition.



pthread_cond_t (continued)

- Functions used in conjunction with the condition variable:
- Creating/Destroying:
 - [pthread_cond_init](#)
 - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
 - [pthread_cond_destroy](#)
- Waiting on condition:
 - [pthread_cond_wait](#)
 - [pthread_cond_timedwait](#) - place limit on how long it will block.
- Waking thread based on condition:
 - [pthread_cond_signal](#)
 - [pthread_cond_broadcast](#) - wake up all threads blocked by the specified condition variable.



Example for pthread_cond_t

- See pt4.c



Thread Scheduling

- When the option is enabled, each thread may have its own scheduling properties. Scheduling attributes may be specified:
 - during thread creation
 - by dynamically by changing the attributes of a thread already created
 - by defining the effect of a mutex on the thread's scheduling when creating a mutex
 - by dynamically changing the scheduling of a thread during synchronization operations.
 - The threads library provides default values that are sufficient for most cases.



Thread safe

- is threaded routine code that must call functions which are "thread safe".
- no static or global variables which other threads may cause a racing condition
- If static or global variables are used then mutexes must be applied or the functions must be re-written to avoid the use of these variables.

