CSCE 3600
Principles of Systems Programming

Threads

University of North Texas

Threads

Thread Motivation

- Processes are expensive to create and maintain
  - Context switch overhead between processes
  - Process synchronization cumbersome
  - Communication between processes must be done through an external structure
    - Files
    - Pipes
    - Sockets
    - Shared memory

We will discuss these soon!

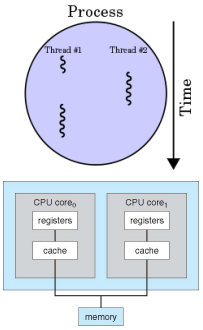
- What to do?
  - Threads

3

Processes and Threads

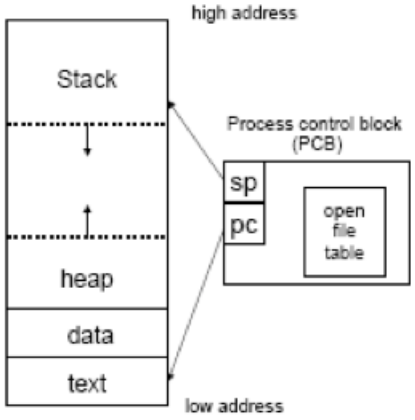
- Process considered to be *heavyweight*
  - Ownership of memory, files, other resources
- Threads
  - A thread can be seen as lightweight process
  - A unit of execution associated with a particular process, using many of the process' resources
  - Multithreading
    - Allowing multiple threads per process
  - Benefits of multithreading
    - Responsiveness (minimize time, concurrency)
    - Resource sharing (i.e., shared memory)
    - Economy (creation, switch)
    - Scalability (explore multi-core CPUs)

A process may have multiple threads, but a thread can only belong to one process



4

Process Properties



- Each process has its own:
  - Program Counter (PC)
  - Stack
  - Stack Pointer (SP)
  - Address space
- Processes may share:
  - Open files
  - Pipes

5

Thread View

- Operating System Perspective
  - An independent stream of instructions that can be scheduled to be run by the OS
- Programmer Perspective
  - Can be seen just as a "function" that executes independently from the main program
  - A single stream of instructions in a program
- Multiple threads can share a process
  - Multiple flows of control within a process
  - Share code and data space of process
  - Lower overhead, avoid context switches

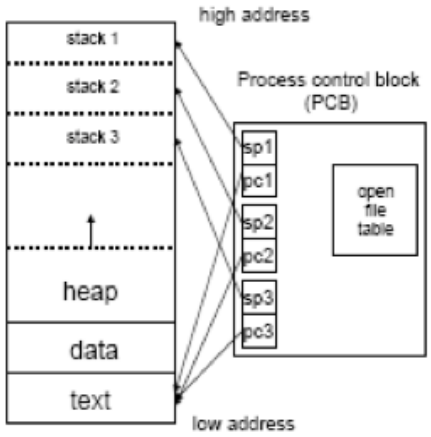
6

Execution Environment

- The process is the execution environment for a family of threads
  - A group of threads share the same resources (files, memory space, etc.)
    - Since threads are associated with their process, they are able to use the resources belonging to the process and duplicate only the required resources, such as the PC, stack, and SP, needed by the OS to schedule threads independently from their processes
  - This makes creating and switching between threads belonging to same process very simple and much more efficient than creating or switching between processes
    - Although switching for threads belonging to different processes is still as complex as classic process switch

7

Thread Properties



Since a thread exists within a process, if a process terminates, so does the thread

- Each thread has its own:
  - Program Counter (PC)
  - Stack
  - Stack Pointer (SP)
  - Registers
  - Scheduling properties
  - Set of pending/blocked signals
  - Thread specific data
- Threads share:
  - Address space
    - Variables
    - Code
  - Open files

8

Thread Consequences

- Since threads within the same process share resources
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
    - Any files opened inside a thread will remain open (unless explicitly closed) even after the thread is terminated
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory locations is possible
- Therefore requires explicit synchronization by the programmer

9

POSIX Threads

10

POSIX Threads

- The most commonly used thread package on Linux is POSIX threads (also known as pthreads)
  - Standards-based API to create, manage, and synchronize threads
- Thread package includes library calls for
  - Thread management (creation, destruction)
  - Mutual exclusion (synchronization, short-term locking)
  - Condition variables (waiting on events of unbounded duration)
- A run-time system manages threads in a transparent manner so that the user is unaware
  - Robust programs should not depend upon threads executing in a specific order

11

The Pthreads Library

- The `pthread.h` library includes the following operations:
  - **pthread_create** create a new thread
  - **pthread_detach** detach a thread, to release its resources when thread terminates
  - **pthread_exit** terminate calling thread, without terminating process
  - **pthread_join** wait for a specified thread to terminate
  - **pthread_equal** compare thread IDs
  - **pthread_kill** send a specified signal to a thread
  - **pthread_self** obtain the ID of the calling thread

  plus many more…

12

Creating Threads

```
int pthread_create(pthread_t *tid, const
   pthread_attr_t *tattr, void*
   (*start_routine) (void *), void *arg);
```

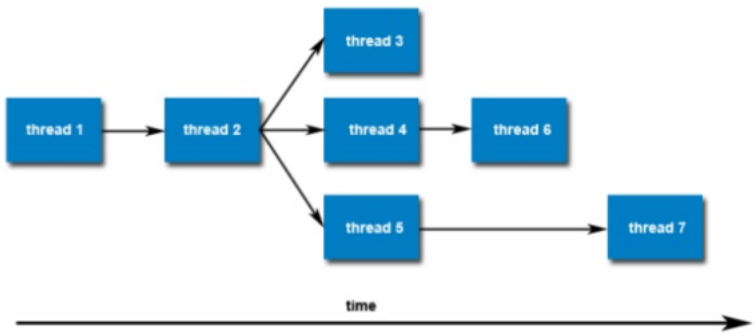
- *tid*: an unsigned long integer that indicates a thread's id
- *tattr*: attributes of the thread – usually NULL
- *start_routine*: the name of the function the thread starts executing
- *arg*: the argument to be passed to the start routine – only one

- After this function gets executed, a new thread has been created and is executing the function indicated by *start_routine*

13

Creating Threads

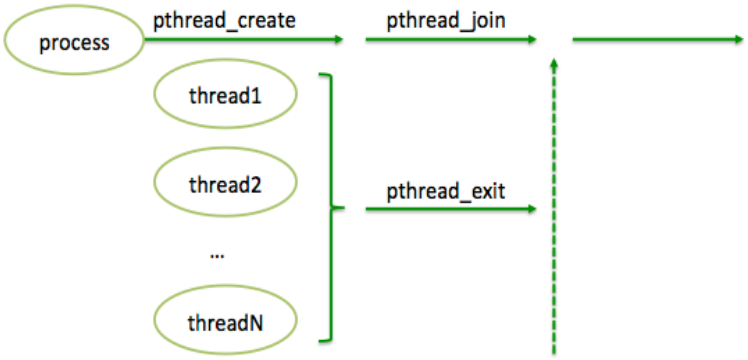
- Once created, threads are peers, and may create other threads
- There is no implied hierarchy or dependency between threads



14

Joining Threads

- Joining is one way to accomplish synchronization between threads where the process waits for all threads to complete
  - We call the function *pthread_join()* once for each thread



15

Wait for Completion of a Thread

```
int pthread_join(thread_t tid, void
  **status);
```

  - *tid*: identification of the thread to wait for
  - *status:* the exit status of the terminating thread – can be NULL

- The process that calls this function blocks its own execution until the thread indicated by *tid* terminates its execution
  - Finishes the function it started with, or
  - Issues a *pthread_exit()* command

16

Exiting a Thread

- `pthreads` exist in user space and are seen by the kernel as a single process
  - If one issues an *exit()* system call, all the threads are terminated by the OS
  - If the *main()* function exits, all of the other threads are terminated
- To have a thread exit, use *pthread_exit()*

```
void pthread_exit(void *status);
```

  - *status:* the exit status of the thread – passed to the *status* variable in the *pthread_join()* function of a thread waiting for this one

17

Detaching a Thread

- Indicates that system resources for the specified thread should be reclaimed when thread ends
  - If the thread has already ended, resources are reclaimed immediately
  - This routine does not cause the thread to end!
- Threads are detached
  - After a `pthread_detach()` call
  - After a `pthread_join()` call
  - If a thread terminates and PTHREAD_CREATE_DETACHED attribute was set on creation
- Failure to join or detach threads – memory and other resources will leak until the process ends

```
int pthread_detach(pthread_t tid);
```

  - *tid*: identification of the thread to detach

18

Example

```
int i, *retval, val[10];
pthread_t tid[10];

for (i = 0 ; i < 10 ; i++) {
    val[i] = i;
    pthread_create(&tid[i], NULL,
                   update_val, val + i));
}
for (i = 0 ; i < 10 ; i++) {
    pthread_join(tid[i], (void **) &retval);
    printf("Thread %d: %d %d\n", i, val[i], *retval);
}
```

Include: <pthread.h>
Library: -lpthread

19

Example (cont'd)

```
void * update_val(void *arg)
{
    int i, *count;

    count = (int *) arg;
    for (i = 0 ; i < 10 ; i++)
        *count += 1;

    pthread_exit(count);
}
```

Output:

```
Thread 0: 10 10
Thread 1: 11 11
Thread 2: 12 12
Thread 3: 13 13
Thread 4: 14 14
Thread 5: 15 15
Thread 6: 16 16
Thread 7: 17 17
Thread 8: 18 18
Thread 9: 19 19
```

20

Thread Safety

21

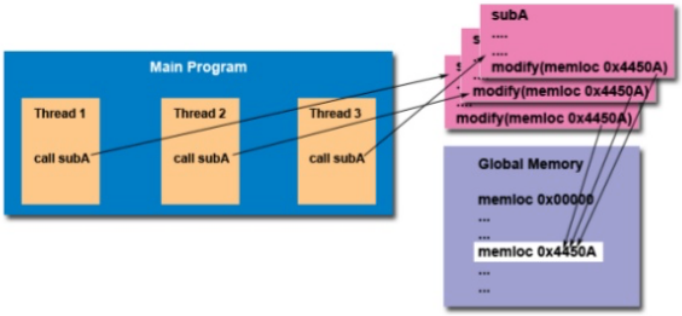
Thread Safety

- The threads for a process share the entire address space of the process
- Can modify global variables, access open file descriptors, etc.
  - Be careful when reusing variables passed by reference
- Need to ensure that multiple threads do not interfere with each other – synchronize thread execution
- A program or function is said to be thread safe if it can be called from multiple threads without unwanted interaction between the threads
  - That is, it should be able to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions

22

Thread Safety

- The implication to users of external library routines:
  - If you aren't 100% certain a routine is thread-safe, then you take your chances with problems that could arise
- Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness



23

Thread-Safe Functions

- Not all functions can be called from threads
  - Many use global/static variables
  - Many functions have thread-safe replacements with `_r` suffix
- Safe:
  - `ctime_r(), gmtime_r(), localtime_r(), rand_r(), strtok_r()`
- Not safe:
  - `ctime(),gmtime(),localtime(),rand(),strtok()`
- Could use semaphores to protect access, but this generally results in poor performance

24

Thread Synchronization

25

Data Race Example

```
static int s = 0;
```

Thread 0

```
…
for (i = 0; i < n/2 – 1; i++)
    s = s + f(A[i]);
…
```

Thread 1

```
…
for (i = n/2; i < n – 1; i++)
    s = s + f(A[i]);
…
```

- Also called a critical section problem
- A race condition or data race occurs when
  - Two processors (or two threads) access the same variable, and at least one does a write
  - The accesses are concurrent (not synchronized) so that could happen simultaneously

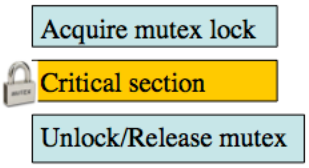
26

Synchronizing Threads

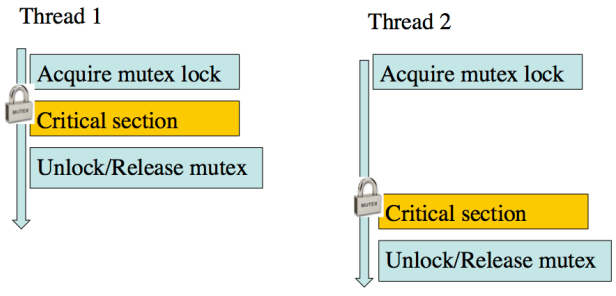
- Three basic synchronization primitives
  - Mutex Locks
    - Control thread's access to the data with simple mutual exclusion
  - Condition Variables
    - More complex synchronization
    - Let threads wait until a user-defined condition becomes true (i.e., based on the value of the data
    - Removes polling requirement
  - Semaphores
    - Signal-based synchronization
    - Allows sharing (not wait unless semaphore = 0)
    - Access to data granted/blocked based on semaphore value

27

Mutex Locks



- Mutex (mutual exclusion) is a special type of variable used to restrict access to a critical section to a single thread at a time
  - Guarantee that one thread "excludes" all other threads while it executes the critical section
  - When a thread waits on a mutex/lock, CPU resource can be used by others



28

Mutex Locks

- A mutex lock is created like a normal variable
  - *pthread_mutex_t mutex;*
- Easier to use than standard semaphores
- Only the thread that locks a mutex can unlock it
- Mutexes are often declared as global variables
- Mutexes must be initialized before being used
  - A mutex can only be initialized once

```
int pthread_mutex_init(pthread_mutex_t
  *mp, const pthread_mutexattr_t *mattr);
```

  - *mp:* a pointer to the mutex lock to be initialized
  - *mattr:* attributes of the mutex – usually NULL

29

Locking a Mutex

- To ensure mutual exclusion to a critical section, a thread should lock a mutex
  - When locking function is called, it does not return until the current thread owns the lock
  - If the mutex is already locked, calling thread blocks
  - If multiple threads try to gain lock at the same time, the return order is based on priority of the threads
    - Higher priorities return first
    - No guarantees about ordering between same priority threads

```
int pthread_mutex_lock(pthread_mutex_t
    *mp);
```

  - *mp:* mutex to lock

30

More Locking a Mutex

- A modified version of the lock function
  - Returns 0 if the mutex is acquired (i.e., locked), but does not block if not successful
    - This implies that we check the return code and only execute the critical section if successful

```
int pthread_mutex_trylock(pthread_mutex_t *mp);
```

- *mp:* mutex to unlock

31

Unlocking a Mutex

- When a thread is finished within the critical section, it needs to release the mutex
  - Calling the unlock function releases the lock
  - Then, any threads waiting for the lock compete to get it
  - Very important to remember to release mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

- *mp:* mutex to unlock

32

Example

```
pthread_mutex_t myMutex;
int status;

status = pthread_mutex_init(&myMutex, NULL);
if(status != 0)
    printf("Error: %s\n", strerror(status));

pthread_mutex_lock(&myMutex);
/* critical section here */
pthread_mutex_unlock(&myMutex);

status = pthread_mutex_destroy(&myMutex);
if(status != 0)
    printf("Error: %s\n", strerror(status));
```

A mutex must be destroyed (i.e., uninitialized) once you are done using it

33

Condition Variables

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met.
- This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock

34

Condition Variables

- A condition variable is an *explicit queue* that threads can put themselves in when some state of execution (i.e., some condition) is not as desired (by waiting on the condition)
- Another thread, when it changes the condition, can wake one (or more) of those waiting threads to allow them to continue (by signaling on the condition)
- A condition variable has two operations associated with it:
  - The wait() call is executed when a thread wishes to put itself to sleep
  - The signal() call is executed when a thread has changed something in the program and now wants to wake a sleeping thread waiting on this condition

35

Condition Variables

- Waiting and signaling on condition variables
- Routines

```
pthread_cond_wait(pthread_cond_t *c,
pthread_mutex_t *m);
```

  - Blocks the thread until the specific condition is signaled
  - Should be called with mutex locked
  - Automatically release the mutex lock while it waits
  - When return (condition is signaled), mutex is locked again

```
pthread_cond_signal(pthread_cond_t *c);
```

  - Wake up a thread waiting on the condition variable.
  - Called after mutex is locked, and must unlock mutex after

36

Condition Variables

- Routines (cont'd)

```
pthread_cond_init(pthread_cond_t *c,
const pthread_condattr_t *attr);
```

  - Initializes the condition variable with attributes specified by `attr`
  - If `attr` is NULL, default attributes are used

```
pthread_cond_destroy(pthread_cond_t *c);
```

  - Destroys the condition variable
  - Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior

37

Spurious Wakeup

- For performance reasons, the POSIX API allows the OS to wake up your thread even if the condition has not been fulfilled
  - This is called spurious wakeup, a complication that arises from the use of condition variables as provided by certain multithreading APIs where even after a condition variable appears to have been signaled from a waiting thread's point of view, the condition that was awaited may still be false
  - A thread might be awoken from its waiting state even though no thread signaled the condition variable
  - For correctness it is necessary, then, to verify that the condition is indeed true after the thread has finished waiting
  - Because spurious wakeup can happen repeatedly, this is achieved by waiting inside a loop that terminates when the condition is true

38

Semaphores

- `pthreads` allow the specific creation of semaphores
  - Semaphore is an integer variable and can be initialized to any value
    - Can do increments and decrements of semaphore value
  - Thread blocks if semaphore value is less than or equal to zero when a decrement is attempted
  - As soon as semaphore value is greater than zero, one of the blocked threads wakes up and continues
    - No guarantees as to which thread this might be

39

Creating Semaphores

- Semaphores are created like other variables
  - sem_t semaphore;
- Semaphores must be initialized

```
int sem_init(sem_t *sem, int pshared,
  unsigned int value);
```

  - *sem:* the semaphore value to initialize
  - *pshared:* share semaphore across processes – usually 0
  - *value:* the initial value of the semaphore

40

Decrementing a Semaphore

```
int sem_wait(sem_t *sem);
```

– *sem:* semaphore to try and decrement

- If the semaphore value is greater than 0, the *sem_wait* call return immediately
  - Otherwise it blocks the calling thread until the value becomes greater than 0
- The *sem_wait()* atomic operation has the following semantics

```
sem_wait(S)
{
    while S <= 0 wait in a queue;
    S--;
}
```

41

Incrementing a Semaphore

```
int sem_post(sem_t *sem);
```

– *sem:* the semaphore to increment

- Increments the value of the semaphore by 1
  - If any threads are blocked on the semaphore, they will be unblocked
- Doing a post (i.e., *sem_post*) to a semaphore always raises its value – even if it shouldn't!
- The *sem_post()* atomic operation has the following semantics

```
sem_post(S)
{
    S++;
    Wake up a thread that waits in the queue;
}
```

42

Destroying a Semaphore

```
int sem_destroy(sem_t *sem);
```

- *sem:* the semaphore to destroy

- Destroys the semaphore at the address pointed to by *sem*
- Only a semaphore that has been initialized by *sem_init* should be destroyed using *sem_destroy*
- Destroying a semaphore that other processes or threads are currently blocked on produces undefined behavior
  - Same with using a semaphore that has already been destroyed

43