

Multithreading (contd.)

Saptarshi Ghosh and Mainack Mondal
CS39002

Spring 2020-21



The story so far

- What is a thread?
- Why do you need threads?
- How are threads used in real-world?
- Multithreading models
- POSIX Pthread library

Topics for this lecture

- Thread scheduling
- Thread creation
- Thread cancellation
- Signal handling

Thread scheduling with `pthread`

- One distinction between user-level and kernel-level threads
 - How are they scheduled
- Two scheduling paradigms
 - Process contention scope (PCS)
 - System contention scope (SCS)

Process contention scope (PCS)

- The thread library schedules user-level threads to run within assigned time quantum for the process
 - In many-to-one and many-to-many models
 - Competition for CPU takes place among threads belonging to same process
 - Also called unbound thread

System contention scope (SCS)

- Deciding which kernel-level thread to schedule in CPU
 - Competition for CPU takes place among all threads in the process
 - Also called bound thread

Contention scope with `pthread`

- `pthread` identifies the following contention scope values
 - `PTHREAD_SCOPE_PROCESS` (PCS)
 - `PTHREAD_SCOPE_SYSTEM` (SCS)

Contention scope with `pthread`

- `pthread` identifies the following contention scope values
 - `PTHREAD_SCOPE_PROCESS` (PCS)
 - `PTHREAD_SCOPE_SYSTEM` (SCS)
- `pthread` defines two functions
 - `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
 - `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

Contention scope with `pthread`

- `pthread` identifies the following contention scope values
 - `PTHREAD_SCOPE_PROCESS` (PCS)
 - `PTHREAD_SCOPE_SYSTEM` (SCS)
- `pthread` defines two functions
 - `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
 - `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
- Linux supports `PTHREAD_SCOPE_SYSTEM` but **not** `PTHREAD_SCOPE_PROCESS`

Topics for this lecture

- Thread scheduling
- Thread creation
- Thread cancellation
- Signal handling

Thread creation

- Semantics of `fork()` and `exec()` in a multithreaded environment
- If one thread in a process calls `fork()`, does the new process duplicate all threads in the original process, or is the new process single-threaded?
 - Some Unix systems have two versions of `fork()`, one duplicates all threads, the other duplicates only the thread that invoked `fork()`
- If one thread in a process calls `exec()`, the program specified in the parameter to `exec()` typically used to replace the entire process

Thread creation in Linux

- Provides `fork()` system call to create a new process
- Provides `clone()` system call to create a new thread
- A set of parameters passed to `clone()` to indicate how much sharing is to take place between parent and child
 - File-system information
 - Memory space
 - Open files
 - ... and others

Topics for this lecture

- Thread scheduling
- Thread creation
- Thread cancellation
- Signal handling

Thread cancellation

- Terminate a thread before it has completed
 - E.g., using multiple threads to concurrently search through a database, and one thread returns the result; cancel others
- Thread that is to be canceled referred to as the **target thread**

Thread cancellation: two types

- Asynchronous cancellation
 - Some other thread **immediately** terminates the target thread
 - Can lead to problems in certain situations, e.g.,
 - ... resources have been allocated to the target thread, or
 - ... the target thread is canceled while in midst of updating data shared with other threads

Thread cancellation: two types

- Asynchronous cancellation
 - Some other thread **immediately** terminates the target thread
 - Can lead to problems in certain situations, e.g.,
 - ... resources have been allocated to the target thread, or
 - ... the target thread is canceled while in midst of updating data shared with other threads
- **Deferred** cancellation
 - Some other thread indicates that a target thread is to be terminated
 - Target thread periodically checks whether it should terminate
 - Allows the target thread to terminate itself in orderly fashion, at a suitable point of time

Thread cancellation with `pthread`

- Allows threads to choose one of several cancellation modes
- Allows threads to disable or enable cancellation
 - A thread cannot be canceled if cancellation is disabled
 - Cancellation request remains pending, so the thread can later enable cancellation and respond to the request
- Default: deferred cancellation (asynchronous cancellation not recommended in Pthreads)
 - Cancellation occurs only when a thread reaches a **cancellation point**
 - Cancellation point established by invoking `pthread_testcancel()`
 - If a cancellation request is found pending, a function known as a **cleanup handler** is invoked

Thread cancellation with `pthread`

- Terminate a thread before it has completed
 - `pthread_cancel(pthread_t tid)`
 - `tid`: id of target thread
- Invoking `pthread_cancel` indicates only a request to cancel the target thread
- The exact effect of calling `pthread_cancel` depends on
 - How the target thread is set up to handle the request
 - Basically, this invokes something called a **signal**

Topics for this lecture

- Thread scheduling
- Thread creation
- Thread cancellation
- Signal handling

Signal

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Two types of signals – synchronous and asynchronous

Signal

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Two types of signals – synchronous and asynchronous
- Synchronous signals
 - Usually generated by some invalid operation such as illegal memory access or division by zero
 - Delivered to the same process that performed the operation that caused the signal
- Asynchronous signals
 - Generated by an event external to a running process (that receives the signal)
 - E.g., user terminating a process, or having a timer expire

So, signals and interrupts are similar, right?

Not exactly!

So, signals and interrupts are similar, right?

Not exactly!

- Interrupts are used for communication between CPU and OS kernel
- Initiated by CPU (page fault), devices (input available), CPU instructions (syscalls)
- Eventually managed by CPU, which interrupts the current task and invokes kernel provided ISR
- Signals are used for communication between OS kernel and other processes
- Initiated by the kernel or by some other process.
- Eventually managed by kernel which delivers them to the target process/thread (using either default handler or process-provided handler)

So, signals and interrupts are similar, right?

Not exactly!

- Interrupts are used for communication between CPU and OS kernel
- Initiated by CPU (page fault), devices (input available), CPU instructions (syscalls)
- Eventually managed by CPU, which interrupts the current task and invokes kernel provided ISR
- Signals are used for communication between OS kernel and other processes
- Initiated by the kernel or by some other process.
- Eventually managed by kernel which delivers them to the target process/thread (using either default handler or process-provided handler)

ctrl-c sends a signal SIGINT, is it signal or interrupt?

Some of the POSIX signals

- SIGABRT - Abort
- SIGBUS - Bus error
- SIGILL - Illegal instr.
- SIGKILL - Kill process
- SIGQUIT - Terminal quit
- SIGSEGV - Invalid memory reference
- SIGUSR1/ SIGUSR2 - user defined signal
- SIGINT - Interrupt (ctrl-c)

Signal handling

- How a signal is processed
 - Generated by occurrence of a particular event
 - Delivered to a process
 - Handled by **signal handler** functions
- A signal may be handled by
 - A default handler (that kernel runs when handling the signal)
 - A user-defined or process-defined handler (used to override default handler)

Signal handling

- For single-threaded process, signal delivered to process
- For multithreaded programs, to which thread should the signal be delivered? Several options:
 - Deliver signal to every thread in the process
 - Deliver signal to some particular thread(s) in the process
 - Assign a specific thread to receive all signals sent to this process
- To which thread signal is delivered - depends on type of signal
 - A synchronous signal needs to be delivered to the thread causing the event which generated the signal
 - Some asynchronous signal such as SIGINT (Ctrl-c) should be sent to all threads

Let's write a signal handler

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo){
    if(signo == SIGINT)
        printf("\n Received SIGINT\n");
}

void main(){
    signal(SIGINT, sig_handler);
    while(1)
        sleep(1);
}
```

How to send signal to a specific process?

// via c code

```
kill(pid_t pid, int signal);
```

//via shell

```
kill -signalNumber <pid>
```

```
kill -signalName <pid>
```

```
kill -s signalName <pid>
```

How to send signal to a specific thread?

Sending signal to a specific thread of same process
(provided by POSIX Pthreads)

```
pthread_kill(pthread_t tid, int signal)
```

More about signals

<https://users.cs.cf.ac.uk/Dave.Marshall/C/node24.html>

Manpage for signals (in section 7):

<https://man7.org/linux/man-pages/man7/signal.7.html>

Topics for this lecture

- A recap of `pthread`
- Thread scheduling
- Thread cancellation
- Signal handling
- Thread mutex – left for self-study after process synchronization is covered

General working principle

acquire mutex

while (condition is true)

 wait on condition variable

perform computation on shared variable

update conditional;

signal sleeping thread(s)

Release mutex

pthread mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t  
                        *attr);  
int pthread_mutex_destroy(pthread_mutex_t  
    *mutex);  
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t  
    *mutex);
```

Used for protecting (locking) shared variables

pthread conditional variables

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t  
                      *attr);  
  
int pthread_cond_destroy(pthread_cond_t *cond);  
  
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
  
int pthread_cond_signal(pthread_cond_t *cond);  
  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Example

```
...  
pthread_mutex_lock (&m);  
...  
while (WAITING_CONDITION_IS_TRUE)  
    pthread_cond_wait (&var_this_thread, &m);  
/* now execute*/  
...  
pthread_mutex_unlock (&m);  
pthread_cond_signal (&var_other_thread);  
...
```