Multithreading (contd.)

Indranil Sen Gupta (odd section)
and Mainack Mondal (even section)
CS39002

Spring 2019-20

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

Today's class

- A recap of `pthread`
- Thread scheduling
- Thread cancellation
- Signal handling
- Thread mutex

Today's class

- A recap of `pthread`
- Thread scheduling
- Thread cancellation
- Signal handling
- Thread mutex

What is `pthread`?

- POSIX standard for describing a thread model
  - POSIX?

What is `pthread`?

- POSIX standard for describing a thread model
  - POSIX? Portable Operating System Interface (POSIX)

What is `pthread`?

- POSIX standard for describing a thread model
  - POSIX? Portable Operating System Interface (POSIX)
  - Family of standards for maintaining OS compatibility
  - Basically tells OS you need to support these function calls
  - Increase portability

- All major thread libraries in unix are POSIX compatible

How to use `pthread`?

- Include `pthread.h` in the main file
- Compile program with `-lpthread`
  - `gcc -o test test.c -lpthread`
  - may not report compilation errors otherwise but calls will fail
- Good idea to check return values on common functions

Recap: thread creation

- Types: `pthread_t` – type of a thread
- Function calls:

```
int pthread_create (&tid, &attr, runner, argv[1]);
int pthread_join(tid, NULL);
int pthread_detach();
void pthread_exit();
```

  - Call `pthread_exit` in main
  - Detached threads are those which cannot be joined (can also set this at creation)

exit() Vs. pthread_exit()

- exit() kills all threads
  - Including the main() thread
  - pthread_exit() only kills the running thread but keep the task alive

Attributes

- Type: `pthread_attr_t` (see previous day's example)
- Attributes define the state of the new thread

Attributes

- Type: `pthread_attr_t` (see previous day's example)
- Attributes define the state of the new thread
- State: system scope, joinable, stack size, inheritance
- Default behaviors with `NULL` in `pthread_create()`

Attributes

- Type: `pthread_attr_t` (see previous day's example)
- Attributes define the state of the new thread
- State: system scope, joinable, stack size, inheritance
- Default behaviors with `NULL` in `pthread_create()`

```
int pthread_attr_init(&attr);
pthread_attr_{set/get}{attribute}
```

Attributes

- Type: `pthread_attr_t` (see previous day's example)
- Attributes define the state of the new thread
- State: system scope, joinable, stack size, inheritance
- Default behaviors with `NULL` in `pthread_create()`

```
int pthread_attr_init(&attr);
pthread_attr_{set/get}{attribute}
```

- Example:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
```

Attributes

- Type: `pthread_attr_t` (see previous day's example)
- Attributes define the state of the new thread
- State: system scope, joinable, stack size, inheritance
- Default behaviors with `NULL` in `pthread_create()`

```
int pthread_attr_init(&attr);
pthread_attr_{set/get}{attribute}
```

- Example:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
```

Today's class

- A recap of pthread
- Thread scheduling
- Thread cancellation
- Signal handling
- Thread mutex

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 with 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)`

- `scope` can be:
  - `PTHREAD_SCOPE_PROCESS`
  - `PTHREAD_SCOPE_SYSTEM`

Today's class

- A recap of pthread
- Thread scheduling
- Thread cancellation
- Signal handling
- Thread mutex

Thread cancellation with `pthread`

- Terminates a thread before it has completed
  - `pthread_cancel(pthread_t tid)`

Thread cancellation with `pthread`

- Terminates a thread before it has completed
  - `pthread_cancel(pthread_t tid)`

- The exact effect of calling `pthread_cancel` depends
  - How the target thread is set up to handle the request
  - Basically this invoks something called a signal

Today's class

- A recap of pthread
- Thread scheduling
- Thread cancellation
- Signal handling
- Thread mutex

Signal handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.

Signal handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by signal handlers

Signal handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by signal handlers
- Every signal has a default handler that kernel runs when handling signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process

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
- signals are used for communication between CPU and OS

So, signals and interrupts are similar, right?

Not exactly!

- Interrupts are used for communication between CPU and OS
- Initiated by CPU (page fault), devices (input available), CPU instr. (syscalls)

- signals are used for communication between CPU and OS
- Initiated by kernel or by process.

So, signals and interrupts are similar, right?

Not exactly!

- Interrupts are used for communication between CPU and OS
- Initiated by CPU (page fault), devices (input available), CPU instr. (syscalls)
- Eventually managed by CPU, which interrupts the current task and invokes kernel provided ISR

- signals are used for communication between CPU and OS
- Initiated by kernel or by process.
- Eventually managed by kernel which delivers them to the target process (using either default or user provided routine)

So, signals and interrupts are similar, right?

Not exactly!

- Interrupts are used for communication between CPU and OS
- Initiated by CPU (page fault), devices (input available), CPU instr. (syscalls)
- Eventually managed by CPU, which interrupts the current task and invokes kernel provided ISR

- signals are used for communication between CPU and OS
- Initiated by kernel or by process.
- Eventually managed by kernel which delivers them to the target process (using either default or user provided routine)

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

Some of the POSIX signals

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

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

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

Today's class

- A recap of `pthread`
- Thread scheduling
- Thread cancellation
- Signal handling
- Thread mutex

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_singal(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);
…
```

Next class

- Process synchronization