Problems Caused by Threads

1. Create thread-safe libraries
2. Create cancel-safe libraries
3. Adapt semantics of UNIX's single-threaded model
4. Synchronize access to shared variables

1. Thread-safe Libraries

For library to be "thread safe" means: any number of threads may be executing functions from this library simultaneously

The danger: shared data

Library functions must be **reentrant**:

- Function does not return pointer to static
- Function does not write to shared `errno`
- Function does not use globals OR function gets lock before accessing globals

Thread-safe Libraries, II

During last two decades, C library implementations have been re-written to be thread safe:

- Function does not return pointer to static; e.g., `strerror_r` replaces `strerror`
- Each thread has its own `errno` located in per-thread storage
- Function gets lock before accessing globals; e.g., `malloc`

2. Cancelation

A thread may be canceled:

```
int pthread_cancel(pthread_t target)
```

Similar to sending a kill signal to UNIX process

A thread has:

- Cancel "state" – enabled or disabled
- Cancel "type" — asynchronous or deferred

Cancelation State

"Enabled" means thread can be canceled

"Disabled" means thread cannot be canceled

Cancelation Type

Assuming thread's cancelation state is "enabled" ...

- "Asychronous" means: when canceled, thread is killed immediately
- "Deferred" means: thread may be canceled only at certain "cancelation points" where implementation checks "have I been canceled?" (and if canceled it kills itself)

Deferred cancelation is intended to help with the problem (which exists with signals at the process level) of a thread being canceled at "the worst moment" — leaving some data structure only partly updated, for example

Danger of Asynchronous Cancelation

Q: How to ensure that canceled thread won't be half-done with some operation that should not be left partially done (e.g., transfer of funds)?

A: Can't. Therefore, set thread cancel type to deferred before any sensitive operation:

```
1. set cancel type to deferred

2. perform sensitive operation to completion

3. set cancel type to asynchronous

4. test if thread was canceled:

   if (pthread_testcancel())
       pthread_cancel(pthread_self())
```


Cancelation Points

4 cancelation points in Pthreads implementation:

1. `pthread_testcancel`
2. `pthread_join`
3. `pthread_cond_wait`
4. `pthread_cond_timed_wait`

POSIX states vendors *must* implement cancelation points in 23 specific library functions — roughly, those that may block

POSIX states vendors *may* implement cancelation points in approximately 50 other specific library functions

3. Adapting UNIX Semantics

Key issues:

- Process management
- Signals — to which thread should a signal be delivered?

Adapting UNIX Process Management

Examples of issues that arise when converting classic single-thread UNIX process model to modern multi-threaded processes ...

A. Does `fork` of N-thread process create another N-thread process or a 1-thread process?

B. Does `_exit(2)` terminate just one thread or whole process?

C. What happens to other threads when one thread calls `exec`?

A. Fork

When thread calls `fork` ...

- New 1-thread process is created
- Thread is replica of specific thread in parent process that called `fork`
- Address space of child duplicates that of parent — including all state created by other threads in parent (Yuck!)

Atfork

`pthread_atfork` function exists to help manage potential mess:

```
int pthread_atfork(void (*prepare)(void),
                   void (*parent)(void),
                   void (*child)(void));
```

`prepare` function called in parent before fork

`parent` function called in parent after fork

`child` function called in child after fork

B. Exit

Entire process terminates when any of these events occurs:

- Any thread calls `_exit(2)`
- Thread running `main` terminates
- Fatal signal is delivered to any thread

C. Exec

When any `exec` call happens:

- All existing threads terminated
- New thread created to run `main` of new executable file

Threads and Signals, I

POSIX added:

- Notion of per-thread signal mask
- Thread analogues of signal system calls: `pthread_kill(pthread_t, int)`, `pthread_sigmask`, etc.

Each thread can mask signals individually

How thread handles signal depends on how signal was generated

Threads and Signals, II

If signal generated by hardware or software exception (e.g., `SIGILL` or `SIGSEGV`) ...

then "effective target" of signal is thread that caused exception, so ...

signal is delivered to offending thread

Threads and Signals, III

If signal generated by `pthread_kill` ...

then "effective target" of signal is specific thread, so ...

signal is delivered to targeted thread

Threads and Signals, IV

If signal generated by external process ... (e.g., `SIGCHLD` or `SIGUSR1`) ...

then "effective target" is process, so ...

signal is delivered to *arbitrary thread that does not have signal blocked*

OS likely chooses the thread based on what's simplest to implement

4. Synchronizing Access to Shared Variables

Covered in next segment of the course ...