Posix Threads

S-186

Thread Concepts

- Threads are "lightweight processes"
  - 10 to 100 times faster than `fork()`
- Threads share:
  - process instructions, most data, file descriptors, signal handlers/dispositions, current working directory, user/group Ids
- Each thread has its own:
  - thread ID, set of registers (incl. Program counter and stack pointer), stack (local vars, return addresses), `errno`, signal mask, priority
- Posix threads will (we think) be the new UNIX thread standard
- see CDF man pages for `pthread`, or read Chapter 23 in **Unix Network Programming** by Richard Stevens, 1998, Prentice Hall

S-187

Creating a PThread

```
#include <pthread.h>
int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   void *(*func)(void *), void *arg)
```

- `tid` is unique within a process, returned by function
- `attr`
  - sets priority, scheduling method, initial stack size, *daemon* status
  - can specify as `NULL`
- `func`
  - function to call to start thread
  - accepts one `void *` argument, returns one `void *`
- `arg` is the argument to pass to `func`

S-188

Creating a Pthread [cont'd]

- `pthread_create()` returns 0 if successful, a +ve error code if not
- **does not** set `errno`, but returns compatible codes
- can use `strerror()` to print error messages

Thread Termination

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **status)
```

- `tid`
  - the thread ID of the thread to wait for
  - cannot wait for any thread (*cf.* `wait()`)

S-189

Thread Termination [cont'd]

- `status`, if not NULL, returns the void * returned by the thread when it terminates
- a thread can terminate by
  - returning from `func()`
  - the `main()` function exiting (all threads terminate)
  - `pthread_exit()`

```
#include <pthread.h>
void pthread_exit(void *status);
```

- a second way to exit, returns status explicitly
- `status` must not point to an object **local** to thread, as these disappear when the thread terminates

S-190

"Detaching" Threads

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

- threads are either *joinable* or *detachable*
- if a thread is detached, its termination cannot be tracked with `pthread_join()` … it becomes a *daemon* thread

```
#include <pthread.h>
pthread_t pthread_self(void);
```

- returns the thread ID of the thread which calls it
- often will see `pthread_detach(pthread_self())`

S-191

Passing Arguments to Threads

```
pthread_t thread_ID;
int fd, result ;

result = pthread_create(&thread_ID,
(pthread_attr_t *)NULL, myThreadFcn, (void *)&fd);
if (result != 0)
  printf("Error: %s\n", strerror(result));
```

- we can pass any variable (including a structure or array) to our thread function; assumes thread function knows what type to expect

Thread-Safe Functions

- Not all functions can be called from threads (*e.g.* `strtok()`)
  - many use global/static variables
  - new versions of UNIX have *thread-safe* replacements, like `strtok_r()`
- **Safe:**
  - `ctime_r(), gmtime_r(), localtime_r(), rand_r(), strtok_r()`
- **Not Safe:**
  - `ctime(), gmtime(), localtime(), rand(), strtok(), gethostXXX(), inet_toa()`
- could use semaphores to protect access

PThread Semaphores

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *name,
               const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *name);
int pthread_mutex_lock(pthread_mutex_t *name);
int pthread_mutex_trylock(pthread_mutex_t *name);
int pthread_mutex_unlock(pthread_mutex_t *name);
```

- pthread semaphores are easier to use than `semget()` and `semop()`
- all mutexes must be global
- only the thread that locks a mutex can unlock it

PThread Semaphores [cont'd]

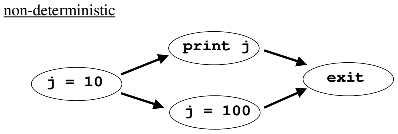
```
pthread_mutex_t myMutex ;
int status ;

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));
```


Concurrency Concepts

Non-determinism

- A process is <u>deterministic</u> when it always produces the same result when presented with the same data; otherwise a process is called <u>non-deterministic</u>



- Evaluation proceeds non-deterministically in one of two ways, producing an output of 10 or 100
- <u>Race conditions</u> lead to non-determinism, and are generally undesirable

Deadlocks

- A concurrent program is in deadlock if all processes are waiting for some event that will never occur
- Typical deadlock pattern:

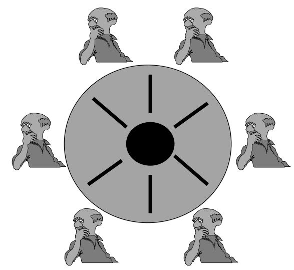
  Process 1 is holding resource X, waiting for Y
  Process 2 is holding resource Y, waiting for X

  Process 1 will not get Y until Process 2 releases it
  Process 2 will not release Y until it gets X,
  which Process 1 is holding, waiting for …

S-198

Dining Philosophers

- **N** philosophers are seated in a circle, one chopstick between each adjacent pair
- Each philosopher needs two chopsticks to eat, a left chopstick and a right chopstick
- A typical philosopher process alternates between eating and thinking (*see next slide*)



S-199

Philosopher Process

```
loop
   <get one chopstick>
   <get other chopstick>

      <eat>

   <release one chopstick>
   <release other chopstick>

      <think>

endloop
```

S-200

Deadlock Example

- For **N**=2, call philosophers P1 and P2, and chopsticks C1 and C2
- Deadlocking sequence:

  P1 requests; gets C1
  P2 requests; gets C2
  P1 requests; WAITS for C2
  P2 requests; WAITS for C1

  ** DEADLOCK **

- Can avoid deadlock if the philosopher processes request both chopsticks at once, and then the get both or wait until both are available

S-201

Comments on Deadlock

- In practice, deadlocks can arise when waiting for some reusable resources. For example, an operating system may be handling several executing jobs, none of which has enough room to finish (and free up memory for the others)
- Operating systems may detect/avoid deadlocks by:
  - checking continuously on requests for resources
  - refusing to allocate resources if allocation would lead to a deadlock
  - terminating a process that is responsible for deadlock
- One can have a process that sits and watches, and can break a deadlock if necessary. This process may be invoked:
  - on a timed interrupt basis
  - when a process wants to queue for a resource
  - when deadlock is suspected (i.e.: CPU utilization has dropped to 0)

S-202

Indefinite Postponement

- Indefinite postponement occurs when a process is blocked waiting for an even that <u>can, but will not occur</u> in some future execution sequence
- This may arise because other processes are “ganging up” on a process to “starve” it
- During indefinite postponement, the overall system does not grind to a halt, but treats some of its processes unfairly
- Indefinite postponement can be avoided by having priority queues which serve concurrent processes on a first-come, first-served basis
- UNIX *semaphores* do this, using a FIFO (first-in, first-out) queue for all requests

S-203