CSE 451
Autumn 2003

Section 3

October 16

Questions from lecture

- Threads vs. processes?
- Kernel vs. user threads?

Homeworks

- Context Switches
- Threads vs. processes

Threads in the real world

- Linux:
  - Kernel only knows about processes (called *tasks*)
  - Threads are implemented by allowing portions of a process to be shared
  - Clone() system call implements fork with varying degrees of sharing
    - Nothing
    - Address space
    - Address space plus file descriptors
  - Implementation:???

Windows

- Full kernel thread support
- Process has no context information
- Thread has no resource information
- Process points to list of threads, threads point to containing process
- Scheduler only looks at threads
- Why the difference?

Who uses threads?

- Web servers
- Databases
- Web browsers
- Scientific programs
- Word processors

Project questions?

- You will implement threads
- You will implement mutexes and condition variables

Simple Threads

- sthread_new_ctx
  - creates a new thread context that can be switched to
    - calls the supplied function with no parameters
- sthread_fee_ctx
  - Deletes the supplied context
- sthread_switch:
  - saves current context
  - switches to supplied context
- sthread_queue.h: generic queue implementation: when do you need a queue?

Threads

- Hints:
  - Handling the initial thread
    - hint: you don’t need context information for a thread while it is running - only when it is waiting to run
  - Starting up a thread
    - The supplied routine for creating a thread (sthread_new_ctx) doesn’t pass parameters to the function it runs
    - How do you pass parameters to a function with no arguments?

Mutexes

- Simple locks that prevent two threads from executing
- Usage:
  - sthread_user_mutex_init() to initialize
  - sthread_user_mutex_lock()
    - Only one thread can do this at a time
  - sthread_user_mutex_unlock()
    - Lets another thread continue past lock
  - sthread_user_mutex_free()
    - Frees lock (can’t be any waiters)

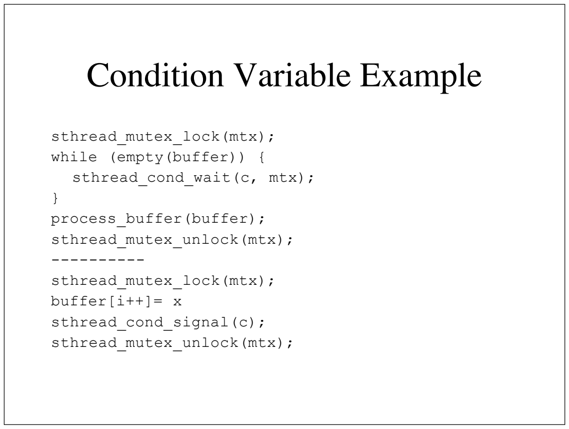
Mutex Example

```
int I = 0;

void update()
{
   sthread_mutex_lock(mtx);
   i++;
   sthread_mutex_unlock(mtx);
}
```

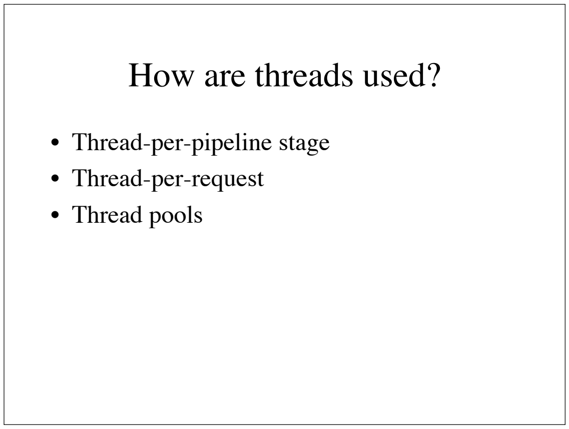
Condition variables

- Used to signal another thread that a condition is true
- Usage:
  - c = sthread_user_cond_init() to initialize
  - sthread_user_cond_free(c) to free
  - sthread_user_cond_wait(c,mtx)
    - Waits until signal
    - unlocks mtx before waiting
    - locks mtx before returning
  - sthread_user_cond_signal(c)
    - Wakes up one waiter
  - sthread_user_cond_broadcast(c)
    - Wakes up all users



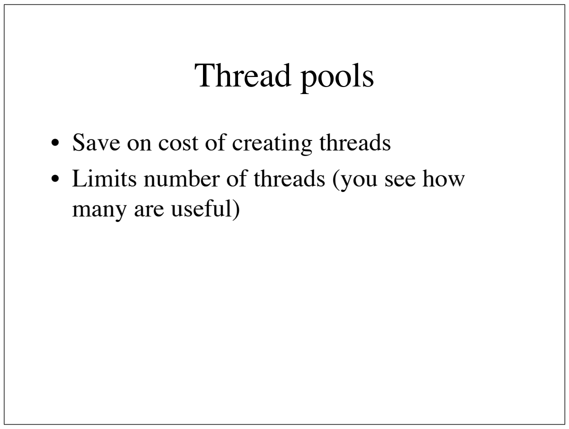
Condition Variable Example

```
sthread_mutex_lock(mtx);
while (empty(buffer)) {
  sthread_cond_wait(c, mtx);
}
process_buffer(buffer);
sthread_mutex_unlock(mtx);
----------
sthread_mutex_lock(mtx);
buffer[i++]= x
sthread_cond_signal(c);
sthread_mutex_unlock(mtx);
```



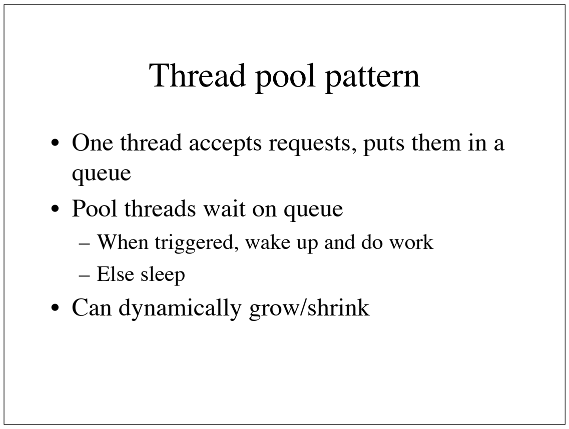
How are threads used?

- Thread-per-pipeline stage
- Thread-per-request
- Thread pools



Thread pools

- Save on cost of creating threads
- Limits number of threads (you see how many are useful)



Thread pool pattern

- One thread accepts requests, puts them in a queue
- Pool threads wait on queue
  - When triggered, wake up and do work
  - Else sleep
- Can dynamically grow/shrink