Threads, Synchronization, and Timers

Processes & Threads

"Humans are actually quite good at doing two or three things at a time, and seem to get offended if their computer cannot do as much." -A. Birrell

- What is a Process?
  - A very separate entity.
    - Think: The CIA vs. CS78
  - Processes are separate programs in an OS
    - (e.g. Firefox and Blitz)
- What is a Thread?
  - A closely related entity.
    - Think: someone to do your laundry for you while you are sitting in class.
  - Threads are part of the same process, run independently, but share memory
    - (e.g. A server thread to talk to each IM chat participant)

Thread Warnings

- Crazy stuff can happen
- Easily
- Even if you are smart
- Heisenbug!
  - Defintion: A bug that disappears or alters its characteristics when it is researched
  - http://en.wikipedia.org/wiki/Heisenbug#Heisenbugs

Important Thread Concepts

- Code runs *concurrently*
  - What does this mean?
- Execution order is *unpredictable*
  - Consider two threads running concurrently

```
void* thread1(void)
{
  printf("A")
  printf("B")
}
```

```
void* thread2(void)
{
  printf("C")
  printf("D")
}
```

See unpredictable.c

  - What output is possible?
- Memory (variables) are *shared*
  - Global variables
  - Parameter variables that are passed at thread creation

Threads in C

- We will be using *pthreads*
  - They’re pretty easy

```
void* ThreadFunc(void* vargp)
{
  // do something fancy
}

int main()
{
  pthread_t t1;
  pthread_create(&t1, NULL, ThreadFunc, NULL);
    … //do something fancy-er
  pthread_join(t1,NULL);
}
```

  - What does pthread_join(…) do?

- I will distribute sample code today
  - Getting started will be quick

- See also: http://www.cs.dartmouth.edu/~pway/cs23/thread_tutorial.html

Thread Synchronization

- Why synchronize threads?
  - Antecdote: Bob needs to change the tires on the car; Alice needs to get to her board meeting
    - *It's probably best that Bob and Alice don't do this at the same time!*
  - The essense: Joe and Sue need access to a *shared resource*

- We synchronize threads to control access to shared resources
  - Variables, classes, anything else…

Synchronization Problems

- Does the following code solve our Alice & Bob problem?

```
bool car_free = true

void* BobThread()
{
  if (car_free == true)
  {
    car_free = false
    ChangeTires()
    car_free = true
  }
}
```

```
//bool car_free = true

void* AliceThread()
{
  if (car_free == true)
  {
    car_free = false
    GoToMeeting()
    car_free = true
  }
}
```

- Definition: *race condition*
  - A flaw in a system whereby the output is unexpectedly and critically dependent on the timing of other events
  - Where is the race condition in the code above?

Synchronization Tools

- How do we solve the Bob & Alice problem?
  - We must prevent the race conditions
  - We need *mutually-exclusive* access to `car_free`
  - So,… we use a lock, also known as a *mutex*
- Locks
  - Prevent race conditions by providing mutually exclusive access to shared resources

```
mutex car_lock = 0

void* BobThread()
{
  lock(car_lock)
  ChangeTires()
  unlock(car_lock)
}
```

```
//mutex car_lock = 0

void* AliceThread()
{
  lock(car_lock)
  DriveCar()
  unlock(car_lock)
}
```

See alicebob.c

More Synchronization Problems

- Do locks guarantee that everything will be OK?
  - Consider this code…

```
mutex car_lock = 0
mutex keys_lock = 0

void* BobThread()
{
  lock(keys_lock)
  lock(car_lock)
  ChangeTires()
  unlock(keys_lock)
  unlock(car_lock)
}
```

```
//mutex car_lock = 0
//mutex keys_lock = 0

void* AliceThread()
{
  lock(car_lock)
  lock(keys_lock)
  DriveCar()
  unlock(car_lock)
  unlock(keys_lock)
}
```

- Definition: *deadlock*
  - A deadlock is a situation wherein two (or more) threads are waiting for the other to finish, and thus neither ever does.

Deadlock Solutions

- Don’t do that?
- Be smart?
- Both?
  - Well, that’d be a start
- Real solutions
  - Do we take the blue pill or the red pill?
  - For this class… we choose blissful ignorance

Timers

- In this course you will use a very simple timer

```
int timer_done = 0;

void* TimerThread(…)
{
  sleep(some_time)
  timer_done = 1
}

int main()
{
  pthread_create(TimerThread)
  while (1)
  {
    // do other things
    if (timer_done)
      break;
  }
}
```

- How might you use this timer?

More correctly:
C’est fini.