Sep 27, 2006
Lecture 7: Threads, Contd

September 5, 2007

1 ADMIN

Hw2 is out today. Since hw2 requires programming of processes that communicates through pipes, we will first discuss the mechanics of pipe communication.

Then we continue with discussion of threads.

2 Review

1. Q: Why do we say that the synchronization problems we have been studying (mutex, producer-consumer, etc) involves COOPERATING processes?

   A: We assume that each process have self-interest in getting the overall system to work properly. That is, each process will follow the correct protocol (e.g., P(S) before C.S. and V(S) after C.S.). If a single process fails to cooperate, nothing will work. This is in contrast to the problems of security where some processes may be NON-cooperating.

2. Q: Busy-waiting seems like a bad idea. But it actually has a use. When is it useful?

   A: When the wait is not expected to be long, it may be cheaper to busywait than to sleep and be woken up again. The latter involves extra context switching (beyond the usual scheduling context switching).

3. Q: What are the differences between Kernel Threads and User Threads?

   A: The former are kernel objects (the kernel has an entry in a thread table for each kernel thread), and they are schedulable. The latter cannot be scheduled directly. It is more expensive to create Kernel Threads, but cheap to create User Threads.

3 UNIX PIPES

This topic is necessary for doing homework 2.

In hw2, we must solve the problem of computing several GCD's using one master process and many child processes (one per GCD). Only the master process knows how to compute the remainder (modulo operation). Child processes must call the master process to do this computation. We intend to do this communication using pipes.

- Two unix processes can communicate via a pipe.
- A pipe has API like a file. Hence let us discuss files first.
- Actually, ALL UNIX I/O has a file-like API.
- Files that are currently open has an entry into a process table called the FILE DESCRIPTOR TABLE (FDT). An index into this table is called a FILE DESCRIPTOR (fd).

- Every process begins with three default opened files: stdin (fd=0), stdout (fd=1), stderr (fd=2).
- We can redefine these standard files: e.g.

```
% wc < main.c > stats.out
```

- Here is how you replace the standout output (fd=1) by a file named "foo".

```
int fid;
fid = open("foo", O_WRONLY | O_CREAT);     // probably fid=3!
close(1);         // close current stdout.
dup(fid);         // first available entry in FDT is assigned
close(fid);       // can close the this entry in FDT
```

  You are guaranteed to have redirected the process output to the file "foo".

- PIPES: here is how you open a pipe. It is basically a pair of file discriptors!

```
int pipeId [2];   // array of size two
...
pipe (pipeId);    // just created a pipe!
```

  HERE, pipeId[0] is the reading-end of the pipe and pipeId[1] is the writing-end of the pipe.

- HOW TO USE PIPES?
- Some useful techniques in using pipes:
  - To write a sequence of numbers into a pipe, and to read them back as numbers, use sprintf(buf,"then write into the pipe. The receiver reads from the pipe into its own buf, and then uses sscan(buf,"read it back.
  - You can set the control bits of a pipe to do a non-blocking read (so you will not block if there is nothing to read).

4 PThread Example

The models of threading is very diverse, depending on the programming language and the underlying OS.

There are two main models of threads that we will use in our exercises: Posix Threads (PTHREADS) and Java Threads.

The following PTHREAD example from the book illustrates a toy problem involving pipes: the idea is that the main process creates a thread to compute a number. Then the main process waits (join) for the thread to finish. Then the main process finishes.

```
#include <pthread.h>
#include <stdio.h>

int sum; // global data
void *runner(void *param) //thread

int main(int argc, char *argv[])
{
  pthread_t tid;
  pthread_attr_t attr;

  if (argc != 2) {
   fprintf(stderr, "usage: a.out <integer value>\n");
exit();
  }
  if (atoi(argv[1]) < 0) {
   fprintf(stderr, "argument (%d) must be positive\n",
atoi(argv[1]));
exit();
```

```
  }

  // INITIALIZE
  pthread_attr_init(&attr);
  pthread_create(&tid, &attr,runner, argv[1]);

  // WAIT
  pthread_join(tid,NULL);
  printf("sum = %d\n", sum);
}

/* runner thread */
void *runner(void *param)
{
  int i, upper=atoi(param);
  sum=0;

  if (upper >0)
     for (i=1; i<= upper; i++) sum +=i;

  pthread_exit(0);
        }
```

5 Java Thread Example