CS 3204
Operating Systems

Lecture 6

Godmar Back

Announcements

- Project 1 due Monday Sep 25, 11:59pm
- Project 1 help session slides online:
- Group formation:
  - See http://courses.cs.vt.edu/~cs3204/fall2006/gback/groups.txt for current list
  - Will only assign as a last resort.
- Announcements:
  - Microsoft "Meet The Company" tonight Sep 7, 7-9pm Pamplin 2030
  - Unix Crash Course by VTLUUG:
    - Monday Sep 11, 7-9pm Squires 236
    - Tuesday Sep 12, 6-8:30pm Squires 116
- Reading assignments:
  - Chapters 1, 2 – skim. Read carefully 1.5.
  - Read carefully Chapter 3.1-3.3
  - Read carefully Chapter 6.1-6.4

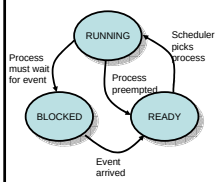
Project 1 Suggested Timeline

- Today Sep 7:
  - Have read project documentation, set up CVS, built and run your first kernel, designed your data structures for alarm clock
- Alarm clock by Sep 9
- Basic priority by Sep 13
- Priority Inheritance & Advanced Scheduler take the most time, start them in parallel – will take the most time to implement & debug
- Due date Sep 25

Processes & Threads

Continued

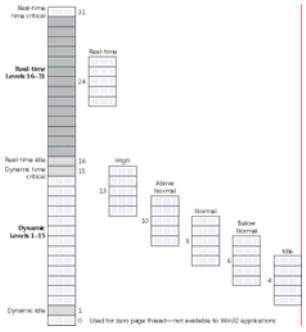
thread_yield()



- Current thread ("RUNNING") is moved to READY state, added to READY list.
- Then scheduler is invoked. Picks a new READY thread from READY list.
- Case a): there's only 1 READY thread. Thread is rescheduled right away
- Case b): there are other READY thread(s)
  - b.1) another thread has higher priority – it is scheduled
  - b.2) another thread has same priority – it is scheduled provided the previously running thread was inserted in tail of ready list.
- "thread_yield()" is a call you can use whenever you identify a need to preempt current thread.
- Exception: inside an interrupt handler, use "intr_yield_on_return()" instead

Windows XP

- Priority scheduler uses 32 priorities
- Scheduling class determines range in which priority are adjusted
- Source: Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™



Process Creation

- Two common paradigms:
  - Cloning vs. spawning
- Cloning: (Unix)
  - "fork()" clones current process
  - child process then loads new program
- Spawning: (Windows, Pintos)
  - "exec()" spawns a new process with new program
- Difference is whether creation of new process also involves a change in program

fork()

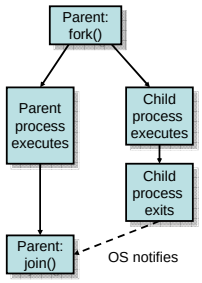
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int ac, char *av[])
{
    pid_t child = fork();
    if (child < 0)
        perror("fork"), exit(-1);
    if (child != 0) {
        printf ("I'm the parent %d, my child is  %d\n",
            getpid(), child);
        wait(NULL);    /* wait for child ("join") */
    } else {
        printf ("I'm the child  %d, my parent is %d\n",
            getpid(), getppid());

        execl("/bin/echo", "echo", "Hello, World", NULL);
    }
}
```


Fork/Exec Model

- Fork():
  - Clone most state of parent, including memory
  - Inherit some state, e.g. file descriptors
  - Important optimization: copy-on-write
    - Some state is copied lazily
  - Keeps program, changes process
- Exec():
  - Overlays current process with new executable
  - Keeps process, changes program
- Advantage: simple, clean
- Disadvantage: does not optimize common case (fork followed by exec of child)

The fork()/join() paradigm

- After fork(), parent & child execute in parallel
- Purpose:
  - Launch activity that can be done in parallel & wait for its completion
  - Or simply: launch another program and wait for its completion (shell does that)
- Pintos:
  - Kernel threads: thread_create (no thread_join)
  - exec(), you'll do wait() in Project 2



CreateProcess()

```
// Win32
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation );
```

- See also system(3) on Unix systems
- Pintos exec() is CreateProcess(), not like Unix's exec()

Thread Creation APIs

- How are threads embedded in a language?
- POSIX Threads Standard (in C)
  - pthread_create(), pthread_join()
  - Uses function pointer
- Java/C#
  - Thread.start(), Thread.join()
  - Java: Using "Runnable" instance
  - C#: Uses "ThreadStart" delegate
- C++
  - No standard has emerged as of yet
  - see ISO C++ Strategic Plan for Multithreading

Example pthread_create/join

```
static void * test_single(void *arg)
{
        // this function is executed by each thread, in parallel
}

/* Test the memory allocator with NTHREADS t
  pthread_t threads[NTHREADS];
  int i;
  for (i = 0; i < NTHREADS; i++)
    if (pthread_create(threads + i, (const pthread_attr_t*)NULL,
                       test_single, (void*)i) == -1)
      { printf("error creating pthread\n"); exit(-1); }

  /* Wait for threads to finish. */
  for (i = 0; i < NTHREADS; i++)
    pthread_join(threads[i], NULL);
```

Use Default Attributes – could set stack addr/size here

2nd arg could receive exit status of thread

Java Threads Example

```
public class JavaThreads {
   public static void main(String []av) throws Exception {
      Thread [] t = new Thread[5];
      for (int i = 0; i < t.length; i++) {
         final int tnum = i;
         Runnable runnable = new Runnable() {
            public void run() {
               System.out.println("Threa
            }
         };
         t[i] = new Thread(runnable);
         t[i].start();
      }
      for (int i = 0; i < t.length; i++)
         t[i].join();
      System.out.println("all done
   }
}
```

Threads implements Runnable – could have subclassed Thread & overridden run()

Thread.join() can throw InterruptedException – can be used to interrupt thread waiting to join via Thread.interrupt

Why is taking C++ so long?

- Java didn't – and got it wrong.
  - Took years to fix
- What's the problem?
  - Compiler must know about concurrency to not reorder operations past implicit synchronization points
  - See also Pintos Reference Guide A.3.5 Memory Barriers
  - See Boehm [PLDI 2005]: Threads cannot be implemented as a library

```
lock (&l);
flag = true;
unlock (&l);
```

→

```
lock (&l);
unlock (&l);
flag = true;
```

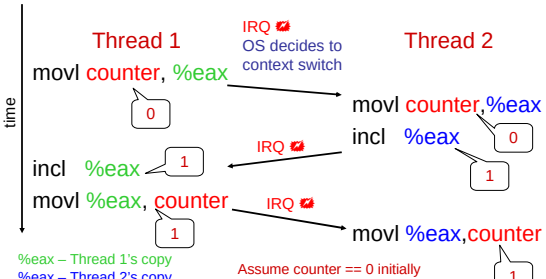

Concurrency & Synchronization

pthread_mutex example

```
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static int counter = 0; /* A global variable to protect. */

/* Function executed by each thread. */
static void *
increment(void *_)
{
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
}
```

```
movl   counter, %eax
incl   %eax
movl   %eax, counter
```


A Race Condition



Race Conditions

- Definition: *two or more threads read and write a shared variable, and final result depends on the order of the execution of those threads*
- Usually timing-dependent and intermittent
  - Hard to debug
- Not a race condition if all execution orderings lead to same result
  - Chances are high that you misjudge this
- How to deal with race conditions:
  - Ignore (!?)
    - Can be ok if final result does not need to be accurate
    - Never an option in CS 3204
  - Don't share: duplicate or partition state
  - Avoid "bad interleavings" that can lead to wrong result