CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2017  Lecture 7

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• System Commands: where to get details?
• POSIX Shared memory example
• Does OS manage pipes?
• Producer/Consumer: is the buffer in shared memory?
Search for man fork( )

http://man7.org/linux/man-pages/man2/fork.2.html

NAME
fork - create a child process

SYNOPSIS
#include <unistd.h>

pid_t fork(void);

DESCRIPTION
fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. …

The child process and the parent process run in separate memory spaces…

The child process is an exact duplicate of the parent process except for the following points: …

RETURN VALUE
On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

EXAMPLE
See pipe(2) and wait(2).

…

erro is a global variable in errno.h
Executable Examples

Self exercises
• Piazza forum
• Complete code given
• Compile/Execute as given

errno is a global variable in errno.h
#int main()
{
    pid_t cid;

    /* fork a child process */
    cid = fork();
    if (cid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
    else if (cid == 0) { /* child process */
        printf("I am the child %d, my PID is %d\n", cid, getpid());
        execvp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        printf("I am the parent with PID %d, my parent is %d, my child is %d\n",getpid(), getppid(), cid);
        wait(NULL);

        printf("Child Complete\n");
    }

    return 0;
}
• Process group is a collection of processes
• Each process has a process group ID
• Process group leader?
  – Process with pid==pgid
• kill treats negative pid as pgid
  – Sends signal to all constituent processes
• A child Inherits parent’s process group ID
  – Parent can change group ID of child by using setpgid
  – Child can give itself new process group ID
Process Groups

By default, comprises:

• Parent (and further ancestors)
• Siblings
• Children (and further descendants)

A process can only send *signals* to members of its process group

• Signals are a limited form of inter-process communication used in Unix.
Producer-Consumer Problem

• Paradigm for cooperating processes, *producer* process(s) produces information that is consumed by a *consumer* process(s)
  – *bounded-buffer* assumes that there is a fixed buffer size.
  – “*Circular buffer*” with *in* and *out* index pointers, which are incremented after an insertion/removal.
  – Need to ensure only one process access the buffer at a time.
Pipes

• Command line:
  – Set up pipe between commands
    `ls | more`
  Output of `ls` delivered as input to `more`

• **Ordinary ("anonymous") pipes** – Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. Cannot be accessed from outside the process that created it.

• **Named pipes ("FIFO")** – can be accessed without a parent-child relationship.
UNIX pipe example

```c
#define READ_END 0
#define WRITE_END 1

int fd[2];

create the pipe:
    if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
    }

fork a child process:
    pid = fork();

parent process:
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* close the write end of the pipe */
    close(fd[WRITE_END]);
```

Child inherits the pipe
UNIX pipe example

child process:

/* close the unused end of the pipe */
close(fd[WRITE_END]);

/* read from the pipe */
read(fd[READ_END], read_msg, BUFFER_SIZE);
printf("child read %s\n", read_msg);

/* close the write end of the pipe */
close(fd[READ_END]);
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2016 Lecture 10

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
FAQ

- Context switch in real-time processes (with fixed response times)?
- Logical links? Software abstractions
- POSIX shared memory example: see canvas discussions
- Isn’t message passing also shared memory?
- What kind of messages are sent?
- Is a pipe a form of direct communication?
- What are pipes? Functions, arrays, strings? Special kind of files
- Synchronous (blocking) vs asynchronous (non-blocking)
- When to use shared memory instead of message passing?
- Can a pair of processes have multiple links using the same mailbox?
- When are sockets used? all internet connections
Chapter 4: Threads

Objectives:

• Thread—basis of multithreaded systems
• APIs for the Pthreads and Java thread libraries
• implicit threading, multithreaded programming
• OS support for threads
Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples
Modern applications are multithreaded

• Most modern applications are multithreaded
  – Became common with GUI
• Threads run within application
• Multiple tasks with the application can be implemented by separate threads
  – Update display
  – Fetch data
  – Spell checking
  – Answer a network request
• Process creation is heavy-weight while thread creation is light-weight
• Can simplify code, increase efficiency
• Kernels are generally multithreaded
Multithreaded Server Architecture

1. Request
2. Create new thread to service the request
3. Resume listening for additional client requests
Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation (10-100 times), thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- **Parallelism** implies a system can perform more than one task simultaneously
  - Extra hardware needed for parallel execution
- **Concurrency** supports more than one task making progress
  - Single processor / core: scheduler providing concurrency
Multicore Programming (Cont.)

• Types of parallelism
  – **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  – **Task parallelism** – distributing threads across cores, each thread performing unique operation

• As # of threads grows, so does architectural support for threading
  – CPUs have cores as well as *hardware threads*
    • *e.g. hyper-threading*
  – Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

  ![Diagram showing concurrent execution on a single core system]

- **Parallelism on a multi-core system:**

  ![Diagram showing parallel execution on multiple cores]

---

**Colorado State University**
Single and Multithreaded Processes

![Diagram showing single-threaded and multithreaded processes.](image-url)
Process vs Thread

• All threads in a process have same address space (text, data, open files, signals etc.), same global variables

• Each thread has its own
  – Thread ID
  – Program counter
  – Registers
  – Stack: execution trail, local variables
  – State (running, ready, blocked, terminated)

• Thread is a schedulable entity
Amdahl’s Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components.
- $S$ is serial portion (as a fraction).
- $N$ processing cores.

\[
\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}
\]

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times.
- As $N$ approaches infinity, speedup approaches $1 / S$.

Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

- But does the law take into account contemporary multicore systems?
User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
  - Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads

- **Kernel threads** - Supported by the Kernel

Examples – virtually all general purpose operating systems, including:
- Windows
- Solaris
- Linux
- Mac OS X
Questions from last time

• How threads and processes work together?
  – Process has one of more threads
• What are pipes? Functions, arrays, strings?
  – Special kind of files
• Benefits of having pipes
• Pipes vs other IPC methods
• Pipes vs message passing?
• Questions of threads:
  – Multithreading
  – Implicit threading
Multithreading Models

How do kernel threads support user process threads?

- Many-to-One
- One-to-One (now common)
- Many-to-Many
Many-to-One

- Many user-level threads mapped to single kernel thread (thread library in user space)
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads for Java 1996
  - GNU Portable Threads 2006
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later
Many-to-Many Model

- Allows many user level threads to be mapped to smaller or equal number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9 2002-3
- Windows with the *ThreadFiber* package NT/2000
Two-level Model

• Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

• Examples
  – IRIX -2006
  – HP-UX
  – Tru64 UNIX
  – Solaris 8 and earlier
Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- **Two primary ways of implementing**
  - Library entirely in user space
  - Kernel-level library supported by the OS
Pthreads

• May be provided either as user-level or kernel-level
• A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization 1991
• *Specification*, not *implementation*
• API specifies behavior of the thread library, implementation is up to development of the library
• Common in UNIX operating systems (Solaris, Linux, Mac OS X)
### Some Pthread management functions

<table>
<thead>
<tr>
<th>POSIX function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cancel</td>
<td>Terminate a thread</td>
</tr>
<tr>
<td>pthread_create</td>
<td>Create a thread</td>
</tr>
<tr>
<td>pthread_detach</td>
<td>Set thread to release resources</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Exit a thread without exiting process</td>
</tr>
<tr>
<td>pthread_kill</td>
<td>Send a signal to a thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread</td>
</tr>
<tr>
<td>pthread_self</td>
<td>Find out own thread ID</td>
</tr>
</tbody>
</table>

- Return 0 if successful
POSIX: Thread creation  

- Automatically makes the thread runnable without a start operation
- Takes 3 parameters:
  - Points to ID of newly created thread
  - Attributes for the thread
    - Stack size, scheduling information, etc.
  - Name of function that the thread calls when it begins execution

/* create the thread */

pthread_create(&tid, &attr, runner, argv[1]);
• **`pthread_detach()`**
  - Sets internal options to specify that storage for thread can be reclaimed when it exits
  - 1 parameter: Thread ID of the thread to detach

• **Undetached threads don’t release resources until**
  - Another thread calls `pthread_join` for them
  - Process exits

• **`pthread_join`**
  - Takes ID of the thread to wait for
  - Suspends calling thread till target terminates
  - Similar to `waitpid` at the process level

```c
pthread_join(tid, NULL);
```
POSIX: Exiting and cancellation

- If a process calls exit, **all** threads terminate
- Call to pthread_exit causes only the calling thread to terminate

`pthread_exit(0)`

- Threads can force other threads to return through a *cancellation* mechanism
  - `pthread_cancel`: takes thread ID of target
  - Depends on *type* and *state* of thread
This process will have two threads

- Initial/main thread to execute the main ( ) function. It creates a new thread and waits for it to finish.
- A new thread that runs function runner ( )
  - It will get a parameter, an integer, and will compute the sum of all integers from 1 to that number.
  - New thread leaves the result in a global variable sum.
- The main thread prints the result.
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[]) {
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  
  ```java
  public interface Runnable
  {
    public abstract void run();
  }
  ```

  - Extending Thread class
    • Override its run() method
  - More commonly, implementing the Runnable interface

  1. Has 1 method `run()
  2. Create new Thread class by passing a Runnable object to its constructor
  3. `start()` method creates a new thread by calling the `run()` method.
Java Thread States

- New
- Runnable
- Running
- Dead
- Waiting

**Transition Rules:**
- New to Runnable by `start()` method
- Runnable to Running by `run()` method
- Running to Dead by `end of execution`
- Running to Waiting by `sleep()`, `wait()` methods
- Dead to Waiting by `sleep()`, `wait()` methods

**Notes:**
- `start()` is used to start a new thread.
- `run()` is the method that gets executed in the Runnable state.
- `end of execution` marks the end of a thread's execution.
- `sleep()`, `wait()` methods are used to pause a thread.
Java version of a multithreaded program that computes summation of a non-negative integer.

class Sum {
    private int sum;

    public int get() {
        return sum;
    }

    public void set(int sum) {
        this.sum = sum;
    }
}
class `Summation` implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        if (upper < 0)
            throw new IllegalArgumentException();

        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;

        for (int i = 0; i <= upper; i++)
            sum += i;

        sumValue.set(sum);
    }
}
public class Driver {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage Driver <integer>");
            System.exit(0);
        }
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);

        Thread worker = new Thread(new Summation(upper, sumObject));
        worker.start();
        try {
            worker.join();
        } catch (InterruptedException ie) { }
        System.out.println("The sum of " + upper + " is " + sumObject.get());
    }
}