Thought question

• Before multicore CPUs (waaaaaaay back in the 90s), webservers could handle thousands of visitors at the same time, even though servers supported only 1 thread of execution. How did this work?
Announcements

• Project 1 due today at 5

• Monday (2/1) discussion section moved to 4:30-5:30, CSE 4140
  – Special session for those of you having trouble with sockets, recv(), send(), etc.

• Syllabus update:
  – Regrade requests/updates must be submitted within a week of the assignment being handed back

• Today’s plan:
  – Multitasking via processes
  – Multitasking via threads
  – Constrained multitasking
Supporting multiple clients

• Major service provider needs to handle many clients at a time
  – Google handles ~40,000 queries per second

• Thus far our code has been single-threaded
  – What happens to incoming clients while we’re serving a client socket?
    • Hint: listen(sock, backlog)
Concurrency vs. Parallelism

• Both deal with doing a lot at once, but aren’t the same thing
  – Given set of tasks \( \{T_1, T_2, \ldots, T_n\} \)

• Concurrency:
  – Progress of multiple elements of the set overlap

• Parallelism:
  – Progress on elements of the set occur \textit{at the same time}
Concurrency

• Might be parallel, might not be parallel

• A single thread of execution can time slice a set of tasks to make partial progress over time
  – Time 0: Work on first 25% of T0
  – Time 1: Work on first 25% of T1
  – Time 2: Work on first 25% of T2
  – Time 3: Work on first 25% of T3
  – Time 4: Work on second 25% of T0
  – Time 5: Work on second 25% of T1
  – ...
Parallelism

- Multiple execution units enable progress to be made simultaneously

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time 0: 1\textsuperscript{st} 25% of T1</td>
<td>Time 0: 1\textsuperscript{st} 25% of T2</td>
</tr>
<tr>
<td>Time 1: 2\textsuperscript{nd} 25% of T1</td>
<td>Time 1: 2\textsuperscript{nd} 25% of T2</td>
</tr>
<tr>
<td>Time 2: 3\textsuperscript{rd} 25% of T1</td>
<td>Time 2: 3\textsuperscript{rd} 25% of T2</td>
</tr>
<tr>
<td>Time 3: 4\textsuperscript{th} 25% of T1</td>
<td>Time 3: 4\textsuperscript{th} 25% of T2</td>
</tr>
<tr>
<td>Time 4: 1\textsuperscript{st} 25% of T3</td>
<td>Time 4: 1\textsuperscript{st} 25% of T4</td>
</tr>
</tbody>
</table>
Overview

• We’ll examine three ways to implement concurrency

• Degree of parallelism depends on underlying hardware
  – # CPUs, # CPU threads, # NICs, OS configuration, ...

• Degree of parallelism also a design decision
  – As we will see
Part 1
Multitasking via processes
Multiprocessing

• Run N copies of our echo server program to support N-way concurrency
  – Via multiple UNIX processes

• Key idea:
  – Our server program sets up a server socket
  – Each time a client socket is accept()’d, we spawn a new process specific to that client
    • Via the UNIX fork() command
Linux process memory layout

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE
Random stack offset
RLIMIT_STACK (e.g., 8MB)
Random mmap offset

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

1GB

3GB

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

Fork()

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, referred to as the child, is an exact duplicate of the
calling process, referred to as the parent, except for the following
points:

* The child has its own unique process ID, and this PID does not match
  the ID of any existing process group (setpgid(2)).
Fork() semantics

• New child process created w/ same memory, state, open file/socket descriptors as parent

• Only way to tell is via fork’s return value
  – Returns 0: you are the child
  – Returns >0: you are the parent

• Any open sockets now shared by both processes
  – Technically both can read/write/close
  – Doing so would cause concurrency violation—we need to avoid this
Fork demo

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int pid;

    printf("The parent process has pid %d\n", getpid());
    fflush(stdout);
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "fork()\n");
        exit(1);
    } else if (pid == 0) {
        printf("I'm the child process (pid= %d)\n", getpid());
        fflush(stdout);  
    } else {
        printf("I'm the parent process\n");
        fflush(stdout);
    }
    exit(0);
}
```

Both parent and child can access stdout
Fork() semantics

- A ‘zombie’ process is a terminated child process
- Keeps using OS state until it is ‘harvested’ by parent with the ‘waitpid’ call
Handling zombies

WAIT(2) Linux Programmer's Manual WAIT(2)

NAME
wait, waitpid, waitid – wait for process to change state

SYNOPSIS
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

Child to wait on (-1 for any child)
Filled in with the status of the child (e.g., the exit code)
• NULL if we don’t care
Set to WNOHANG to return immediately if there are no zombies to harvest
TCP Echo Server: Fork edition

• Some common code has been refactored into separate functions:
  – SetupTCPServerSocket(…)
  – clientSock = AcceptTCPConnection(servSock)

• Overview:
  – Parent accepts client socket
  – Fork()
    • Parent: close client socket, accept next connection
    • Child: close server socket, handle request, close client socket
  – Parent harvests any zombie children
Echo Server Demo
Part 2
Multitasking via threads
Downsides of fork()

- Heavy-weight
  - Must copy entire process into a new one
  - Expensive context switching between processes

- Alternative?
  - Concurrency via threading
  - Threads run within same process, so lighter-weight option than fork()
  - Depending on system/hardware, threads can be parallel
Introduction to pthreads

• Pthreads: POSIX common threads package
• Creating a new thread:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Handle to the thread

Arguments passed to new thread

The newly created thread’s main() function
Creating a new thread

• Define a function that will become the new thread’s main() function
• Define any arguments that thread will get
• Create the thread

• Note that the code that handles the client is the same between fork() and thread()
Some details

• In the thread’s main function, arrange for resources to be recovered when the thread exits

    // Guarantees that thread resources are deallocated upon return
    pthread_detach(pthread_self());
Part 3
Constrained multitasking
Flash traffic

• USGS Pasadena, CA office Earthquake site
• Oct 16, 1999 earthquake
Threading and performance

- Too much parallelism causes thrashing, excessive switching, lower performance
Constrained fork-based multiplexing

• Pre-spawn a set of N processes, all sharing a server socket

• Each process calls accept() on the server socket
  – Then handles that client

• accept() called from different threads/processes returns a client socket to *only one* of the callers
Demo: Constrained fork