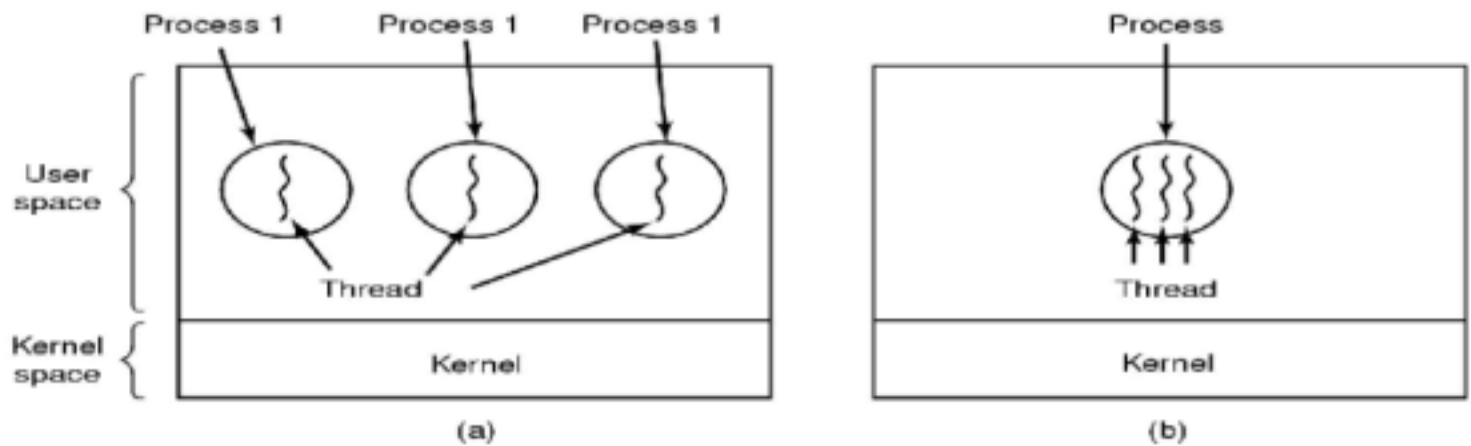


Threads

Threads

- Processes
 - Created with `fork()`
 - Expensive to create, manage, and context switch
 - Threads
 - “Light Weight Processes”
 - Every process already has at least one!
 - Threads are the “engine” inside a process.
-


Processes vs. Threads



- (a): Three processes, one thread /process
- (b): One process, three threads

Processes vs. Threads

- Processes are identified by a unique PID.
 - Threads are identified by a Thread ID (TID).



Each thread in a process has a unique TID, not PID.

Processes each have their **own** address space.

Each thread in a process **shares** the same address space. (Everyone has access to the same global variables – **race condition!**)

Processes vs. Threads

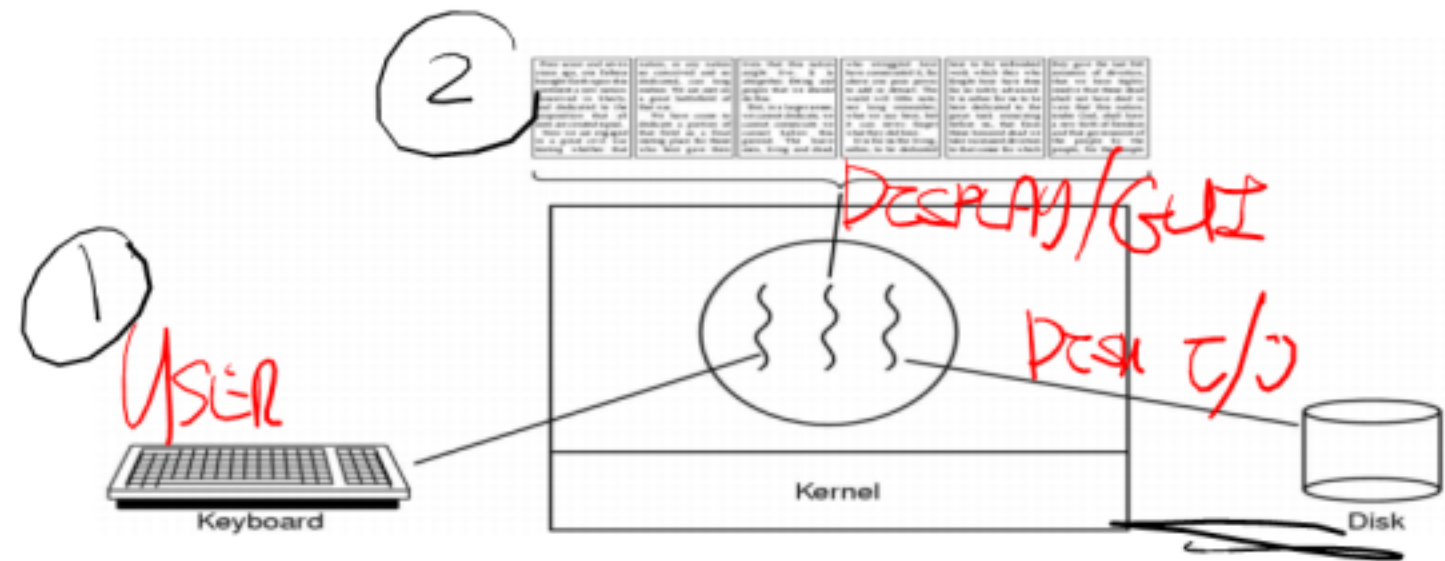
- Processes have separate file descriptors.
 - Threads within a process share file descriptors.
 - Processes are created with `fork()`, duplicating the entire process.
 - Threads are created with `pthread_create()`, which starts the thread at the beginning of a function.
-

Threads are everywhere!

- Consider a variety of processes:
 - **Game:** Diablo III, 1 process, 42 threads
 - **Browser:** Firefox, 1 process, 36 threads
 - **Office:** PowerPoint, 1 process, 8 threads
 - **SSH:** PuTTY, 1 process, 4 threads

Image Name	User Name	CPU	Memory (...)	Threads	I/O Read Bytes	I/O Write Bytes	I/O Other Bytes	Description
Diablo III.exe *32		01	331,532 K	42	88,540,966	27,046	30,285	Diablo III
firefox.exe *32		01	292,276 K	36	10,176,812,348	11,992,350,919	4,494,454,766	Firefox
POWERPNT.EXE *32		00	22,876 K	8	374,407	626,632	64,590	Microsoft Office PowerPoi
putty.exe *32		00	1,876 K	4	27,900	0	20,922	SSH, Telnet and Rlogin cl

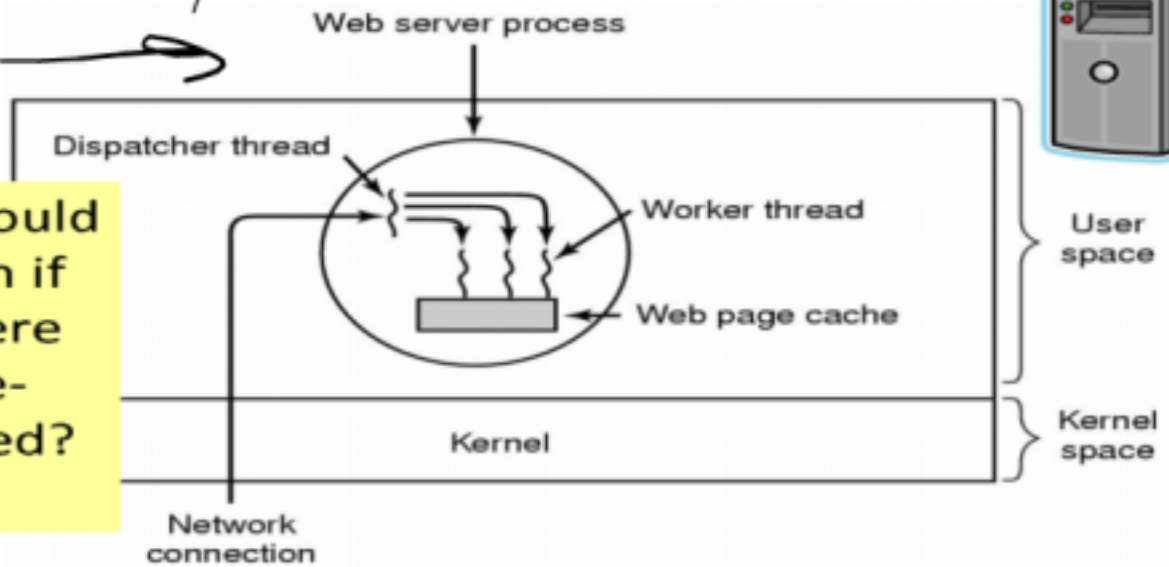
What are threads used for?



Q: What if this is all done by one thread?

What are threads used for?

100 req/sec



What would happen if this were single-threaded?

Things commonly done in separate threads...

- GUI
 - If a blocking operation occurs on the GUI threads, the GUI becomes unresponsive to user input.
 - Networking I/O
 - Disk I/O
- read()
- Long, blocking operations
-

Threads for computation...

- Each thread may run on a separate CPU.
 - Allows for multiple CPUs (“cores”) to work together to accomplish a task.
 - Allows for a single server to handle multiple simultaneous requests.
 - **Speedup**: The factor that a process speeds up when running on multiple cores.
-

Speedup

- Mathematically defined:

$$S_p = \frac{T_1}{T_p}$$

- S_p : Speed-up given **P** processors.
- T_1 : Time a given process takes to run on 1 processor.
- T_p : Time a given process takes to run on **P** processors.

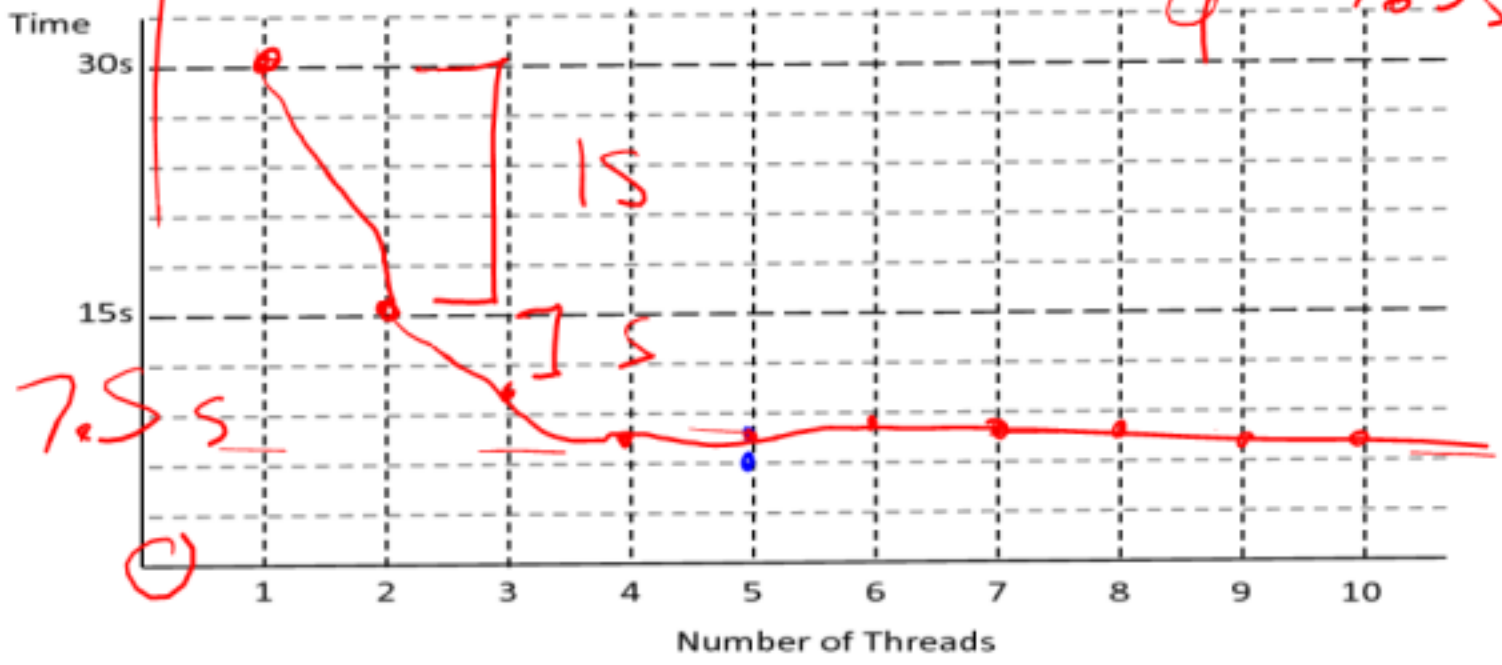
$$S_2 / S_8$$

- Ideal Speedup: $S_p = P$

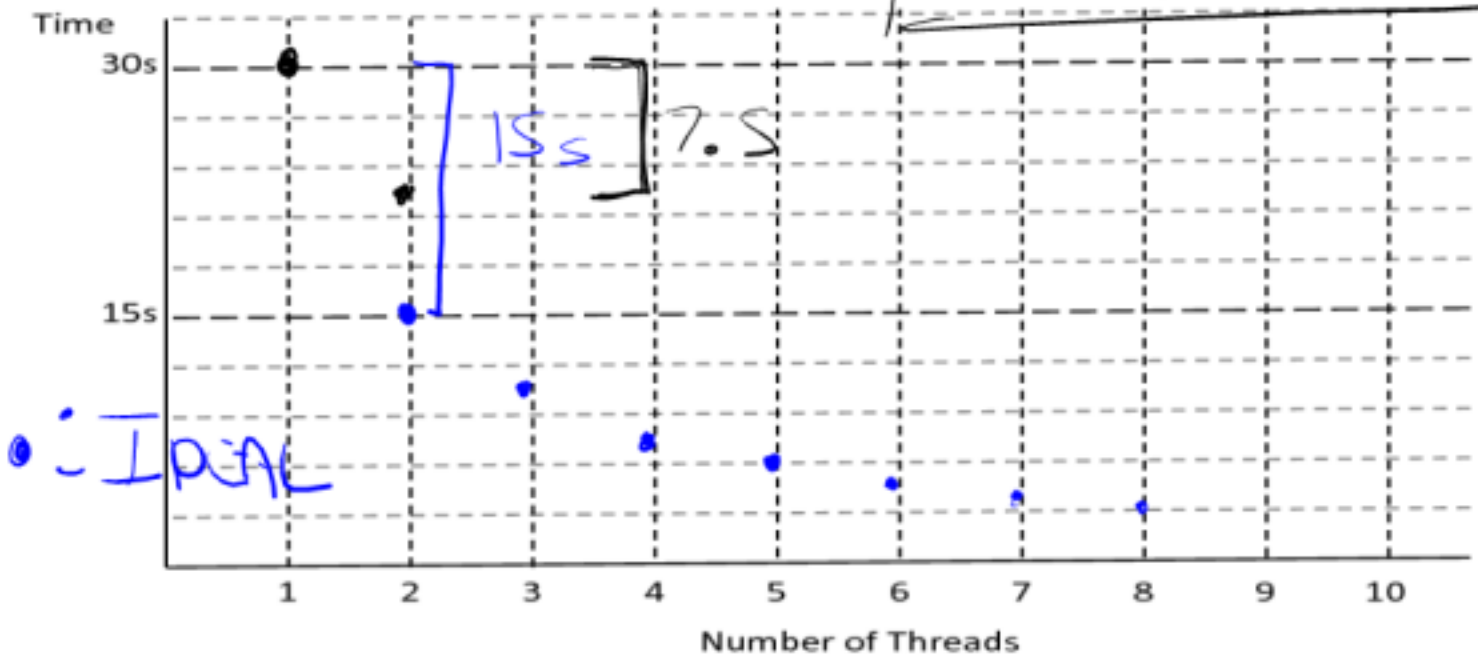
4 PROCESSORS | $T_1 = 30s$ | IDEAL SPEEDUP

$$\frac{30s}{4} = 7.5s$$

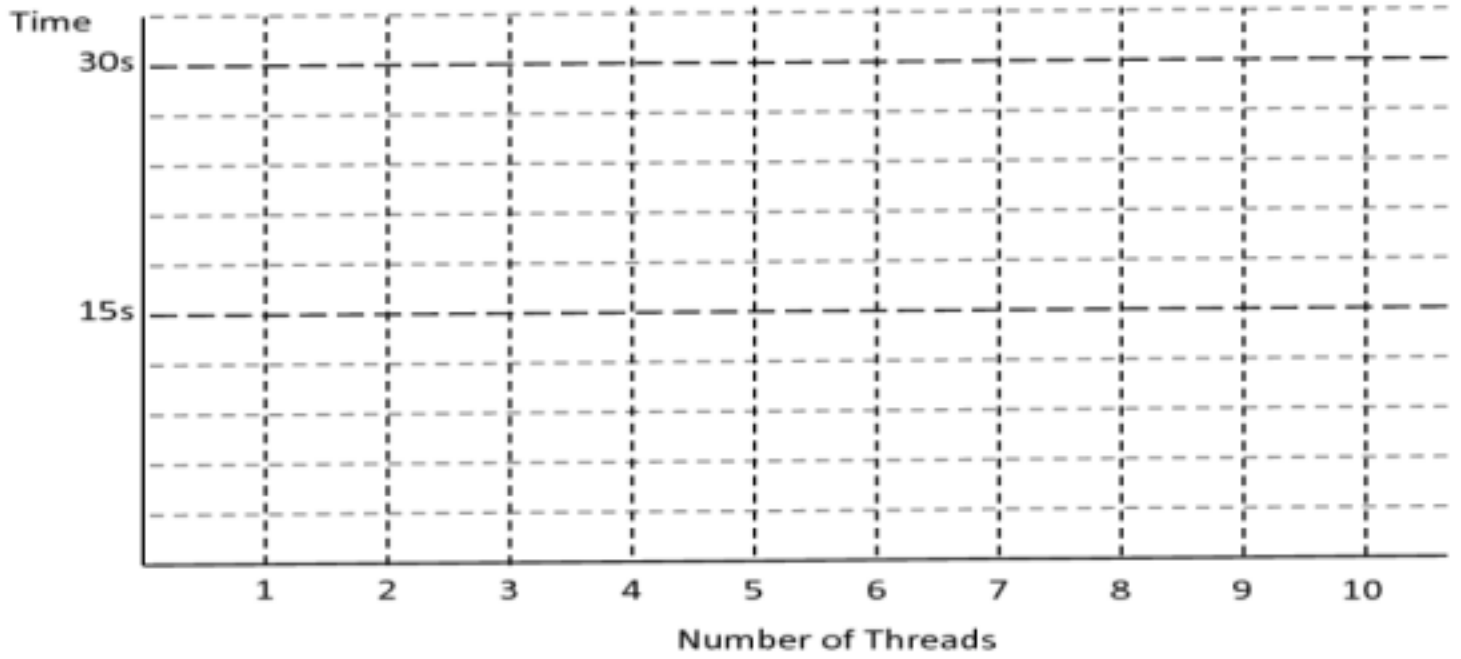
Speedup



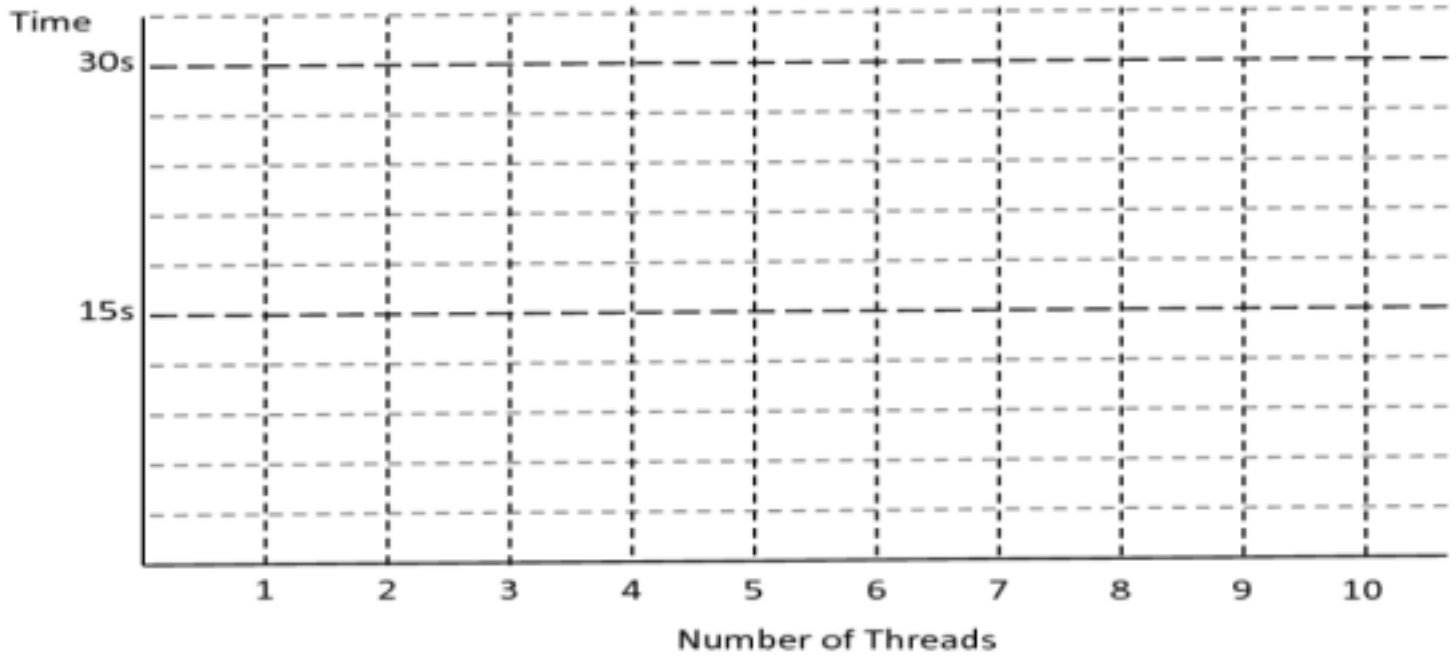
8 PROCESSORS $T_1 = 30$ Speedup = 50% FACTOR



Speedup

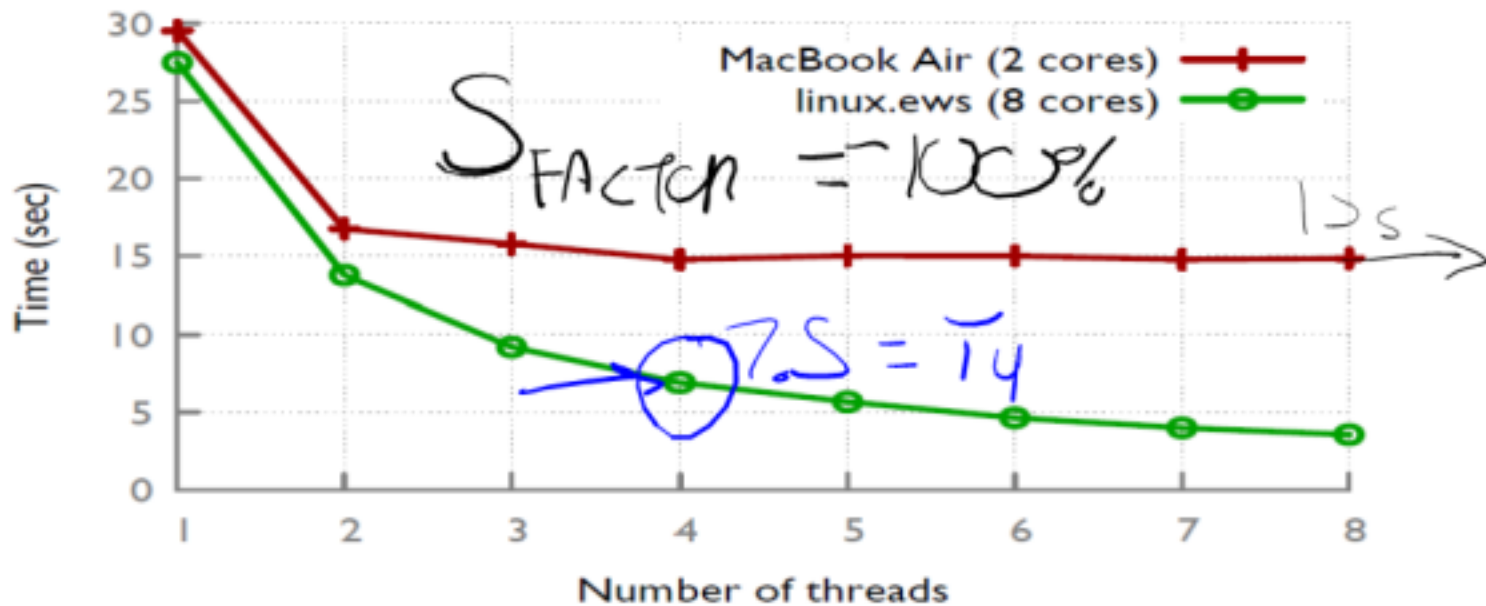


Speedup

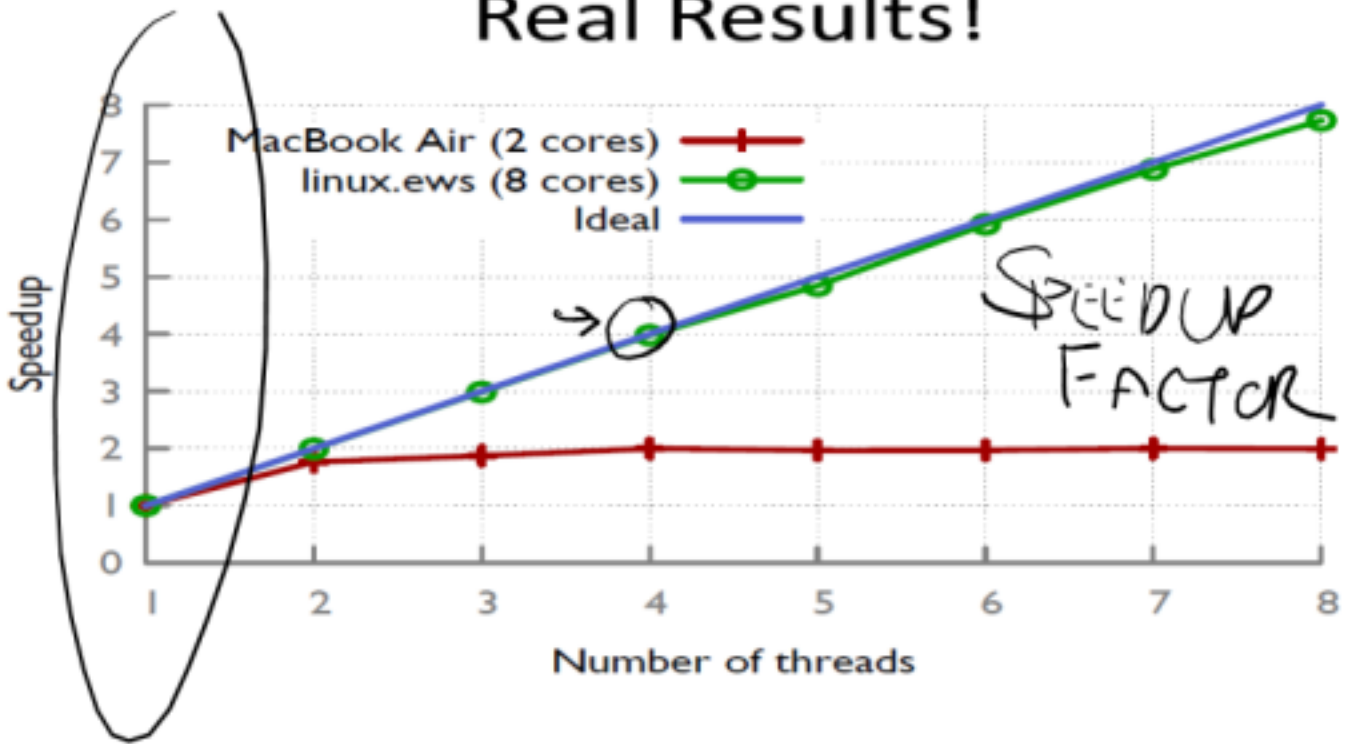


Real Results!

1,137,481,843



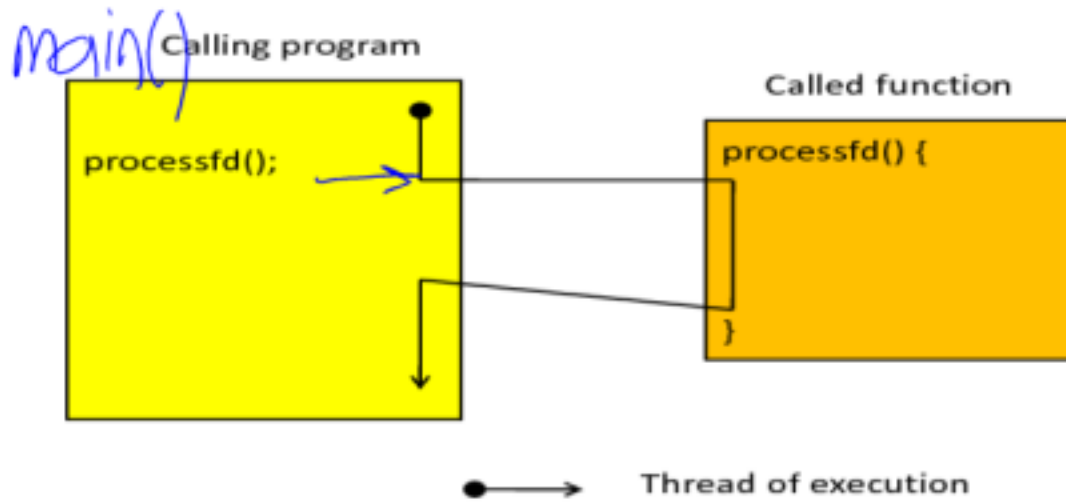
Real Results!



Using Threads

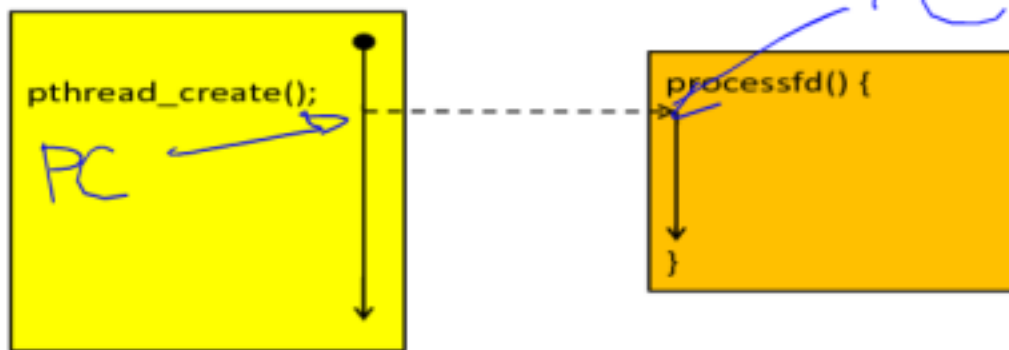
Using Threads

- Making a traditional function call:



Using Threads

- Launching a thread:



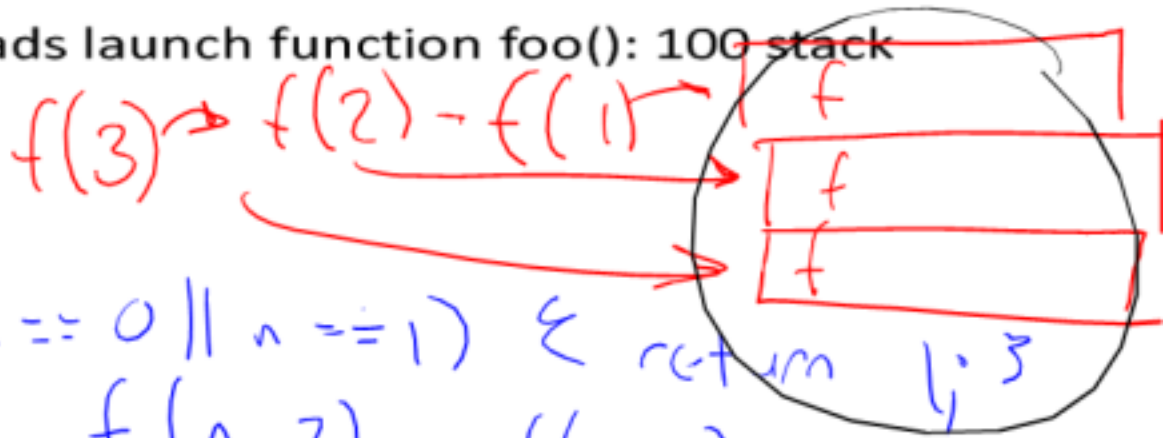
Using Threads

- Just like normal function calls, each function gets its own stack frame!

– 100 threads launch function foo(): 100 stack frames

$\text{fib}(n)$

$\{ \text{if } (n == 0 \parallel n == 1) \{ \text{return } 1; \}$
 $\text{else } f(n-2) + f(n-1);$
 $\}$



Using Threads

Per Process Items	Per Thread Items
Address space ✓ Global variables ✓ Open files ✓ Child processes ✓ Pending alarms ✓ Signals and signal handlers ✓ Accounting information ✓	Program counter ✓ Registers ✓ Stack ✓ State ✓

Using Threads

10x

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.3 GHz Opteron (16 cpus)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cpus)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus)	54.5	1.1	22.2	2.0	1.2	0.6

~ 50x

<http://www.llnl.gov/computing/tutorials/threads>.

Timings reflect 50,000 process/thread

Creations were performed with the time utility, and units are in seconds, no optimization flags.

Creating a Thread

```
int pthread_create (pthread_t* tid, pthread_attr_t*  
attr, void*(child_main), void* arg);
```

- ~~Spawn~~ a new posix thread
- Parameters:
 - **tid**:
 - Unique thread identifier returned from call
 - **attr**:
 - Use **NULL** [for default values]

Example #1

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

void *snow(void *data)
{
    printf("Let it snow ... %s\n", data);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t mythread;
    int result;
    char *data = "Let it snow.";
    result = pthread_create(&mythread, NULL, snow, data);
    printf("pthread_create() returned %d\n", result);
    return 0;
}
```

pthread_create()

~~Let it snow -- Let it snow.~~

Three ways to exit a process...

- A call to `exit()`
 - The “main” function returns a value
 - All threads complete executing
-



Waiting for Threads: `pthread_join()`

```
int pthread_join(pthread_t thread, void** retval);
```

- Suspend calling thread until target thread terminates
- Returns
 - 0 on success
 - Error code on failure
- Parameters
 - `thread`:
 - Target thread identifier
 - `retval`:
 - The value passed to `pthread_exit()` by the terminating thread is made available in the location referenced by `retval`



Example #2

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

void *snow(void *data)
{
    ② → printf("Let it snow ... %s\n", data);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t mythread;
    int result;
    char *data = "Let it snow.";
    ① → result = pthread_create(&mythread, NULL, snow, data);
    printf("pthread_create() returned %d\n", result);
    pthread_join(mythread, NULL);
    return 0;
}
```

Example #3

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

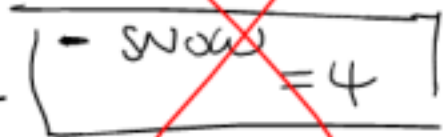
void *snow(void *data)
{
    int snow = 4;
    printf("Let it snow ... %s\n", data);
    return &snow;
}

int main(int argc, char *argv[])
{
    pthread_t mythread;
    int result;
    char *data = "Let it snow.";
    result = pthread_create(&mythread, NULL, snow, data);
    printf("pthread_create() returned %d\n", result);

    void *ret;
    pthread_join(mythread, &ret);
    printf("return value: %d\n", *((int *)ret));

    return 0;
}
```

snow



Example #4

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

```
void *snow(void *data)
{
    int *snow = malloc(
    &SNOW = 4; size(int));
    printf("Let it snow ... %s\n", data);
    return snow;
}
```

```
int main(int argc, char *argv[])
{
    pthread_t mythread;
    int result;
    char *data = "Let it snow.";
    result = pthread_create(&mythread, NULL, snow, data);
    printf("pthread_create() returned %d\n", result);

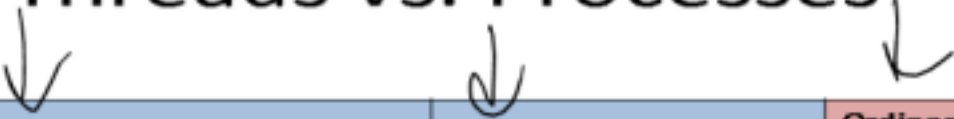
    void *ret;
    pthread_join(mythread, &ret);
    printf("return value: %d\n", *((int *)ret));
    free(ret);
    return 0;
}
```

- ① HEAD
- ② PASS IN A STRUCT w/ AN INT & FOR RETURN
- ③ GLOBALS/STATIC



EXAM REVIEW

Threads vs. Processes



Property	Processes created with fork	Threads of a process	Ordinary function calls
variables	Get copies of all variables	Share global variables	Share global variables
IDs	Get new process IDs	Share the same process ID but have unique thread ID	Share the same process ID (and thread ID)
Data/control	Must communicate explicitly, e.g., use pipes or small integer return value	May communicate with return value or carefully shared variables	May communicate with return value or shared variables
Parallelism (one CPU)	Concurrent	Concurrent	Sequential
Parallelism (multiple CPUs)	May be executed simultaneously	Kernel threads may be executed simultaneously	Sequential

A few other calls...

- **pthread_exit():** Exits the current thread.
↳ `exit()`
 - **pthread_self():** Returns the TID of the current thread.
↳ `getpid()`
 - **pthread_detach():** Frees thread-related memory without needing to `_join()`.
 - Use either `_detach()` or `_join()`, not both!
-

Orphans and Zombies

Orphan

- Often used in relation to a process, an **orphaned processes** is one where the parent has terminated by the child continues.
 - Re-parented by the init process.
 - Will have the PPID of 1.
-

Zombie

- A **zombie process** or **zombie thread** is a thread who has terminated but needs to return state information back to its parent.
 - Done via **wait()** and **waitpid()** for processes.
 - Done via **pthread_join()** for threads

`pthread_detach()`
