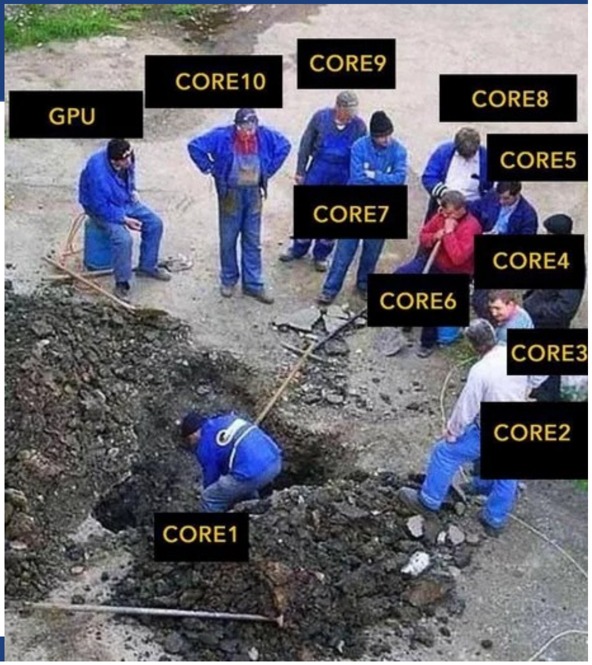
CMPSC 311 - Introduction to Systems Programming

PennState



Introduction to Concurrency

Suman Saha

(Slides are mostly by Professors Patrick McDaniel and Abutalib Aghayev)

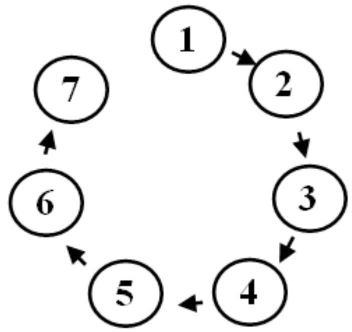
Sequential Programming

- Processing a network connection as it arrives and fulfilling the exchange completely is sequential processing
  - i.e., connections are processed in sequence of arrival

```
listen_fd = Listen(port);
while(1) {
  client_fd = accept(listen_fd);
  buf = read(client_fd);
  write(client_fd, buf);
  close(client_fd);
}
```

Whither sequential?

- Benefits
  - simple to implement
  - very little persistent state to maintain
  - few complex error conditions
- Disadvantages
  - Sometimes poor performance
    - one slow client causes all others to block
    - poor utilization of network, CPU



Think about it this way: if the class took the final exam sequentially, it would take 25 days to complete

An alternate design …

- Why not process multiple requests at the same time, interleaving processing while waiting for other actions (e.g., read requests) to complete?
- This is known as concurrent processing … process multiple requests at the same time
  - Process multiple requests concurrently

- Approaches to concurrent server design
  - Asynchronous servers (`select()`)
  - Multiprocess servers (`fork()`)
  - Multithreaded servers (`pthreads`)

Concurrency with processes

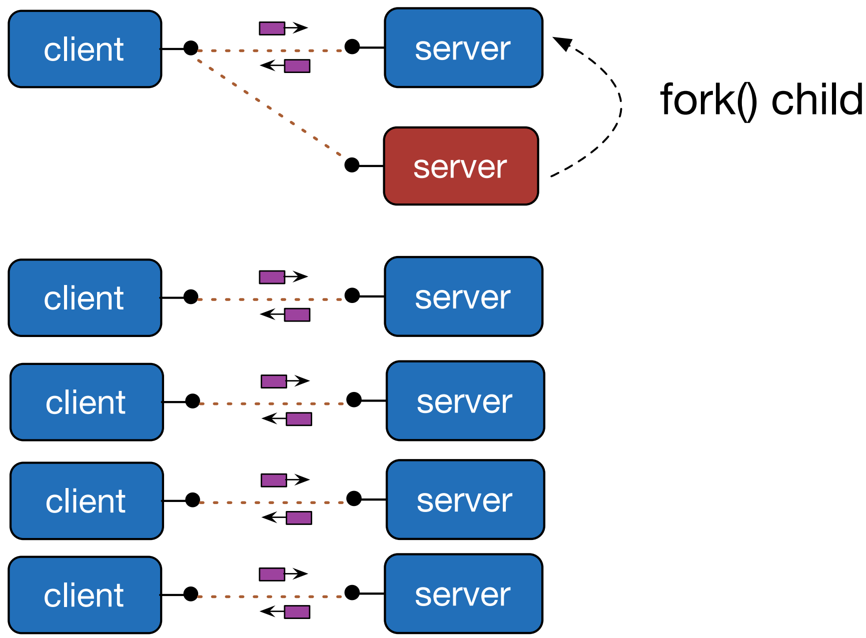
PennState

- The server process blocks on `accept()`, waiting for a new client to connect
  - when a new connection arrives, the parent calls `fork()` to create another process
  - the child process handles that new connection, and `exit()`'s when the connection terminates
- Children become "zombies" after death
  - `wait()` to "reap" children

child process

(Note: child process has the same thing as parent process)

Graphically



fork()

- The fork function creates a new process by duplicating the calling process.

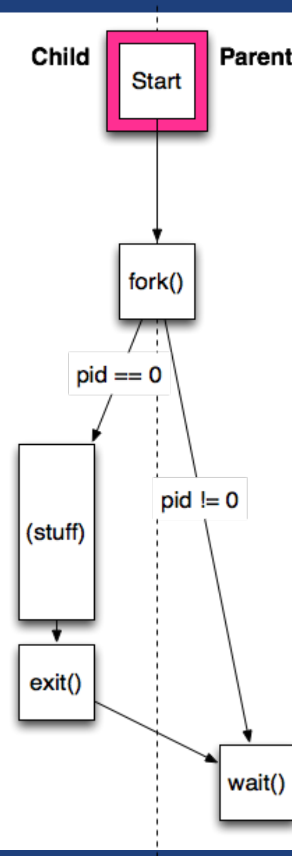
```
pid_t fork(void);
```

- The new child process is an exact duplicate of the calling parent process, except that it has its own process ID and pending signal queue
- The `fork()` function returns
  - 0 (zero) for the child process OR …
  - The child's process ID in the parent code

Idea: think about duplicating the process state and running ….

Process control

- Parent
  - fork (pid == child PID)
  - wait for child to complete (maybe) wait( )
- Child
  - begins at fork (pid == 0)
  - runs until done
  - calls exit



fork( )

pid==0

pid!=0

run

exit( )

child

wait( )

parent

exit()

PennState

- The `exit` causes normal process termination with a return value

`void exit(int status);`

- Where
  - status is sent to the to the parent

Status pass into exit( ) will be sent to parent process

- Note: `exit` vs. `return` from a C program
  - `return` is a language keyword
    - returns (possibly a value) from a function
  - `exit` is a function, eventually calls `_exit` a system call
    - terminates process immediately, returns status to parent
- `exit` and `return` are similar in main function

`wait()`

- The `wait` function is used to wait for state changes in a child of the calling process (e.g., to terminate)

```
pid_t wait(int *status);
```

- Where
  - returns the process ID of the child process
  - `status` is return value set by the child process

Putting it all together …

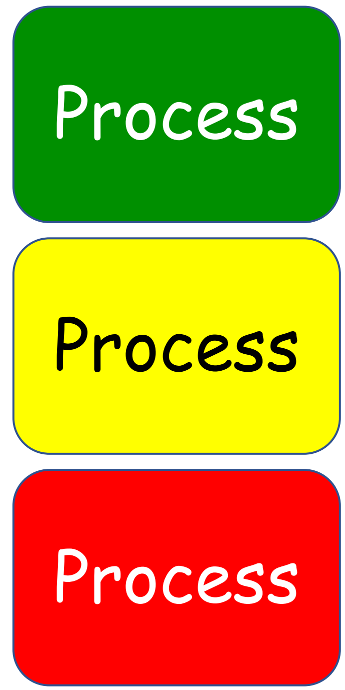
```
int main(void)
{
    printf("starting parent process\n");
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        printf("child processing\n");
        sleep(50);
        exit(19);
    } else {
        printf("parent forked a child with pid = %d\n", pid);
        int status;
        wait(&status);
        printf("child exited with status = %d\n", WEXITSTATUS(status));
    }
    return 0;
}
```

Concurrency with processes

PennState

- Benefits
  - almost as simple as sequential
  - in fact, most of the code is identical!
  - parallel execution; good CPU, network utilization
  - often better security (isolation)
- Disadvantages
  - processes are heavyweight — need large memeory space for child process.
  - relatively slow to fork
  - context switching latency is high → switch process takes times
  - communication between processes is complicated



as simple as sequential
most of the code are identical
better utilize CPU, network
better security

process are heavily need more mem space
folk is slow
context switching latency is high

Concurrency with threads

- A single process handles all of the connections
  - … but … a parent thread forks (dispatches) a new thread to handle each connection
  - the child thread:
    - handles the new connection
    - exits when the connection terminates

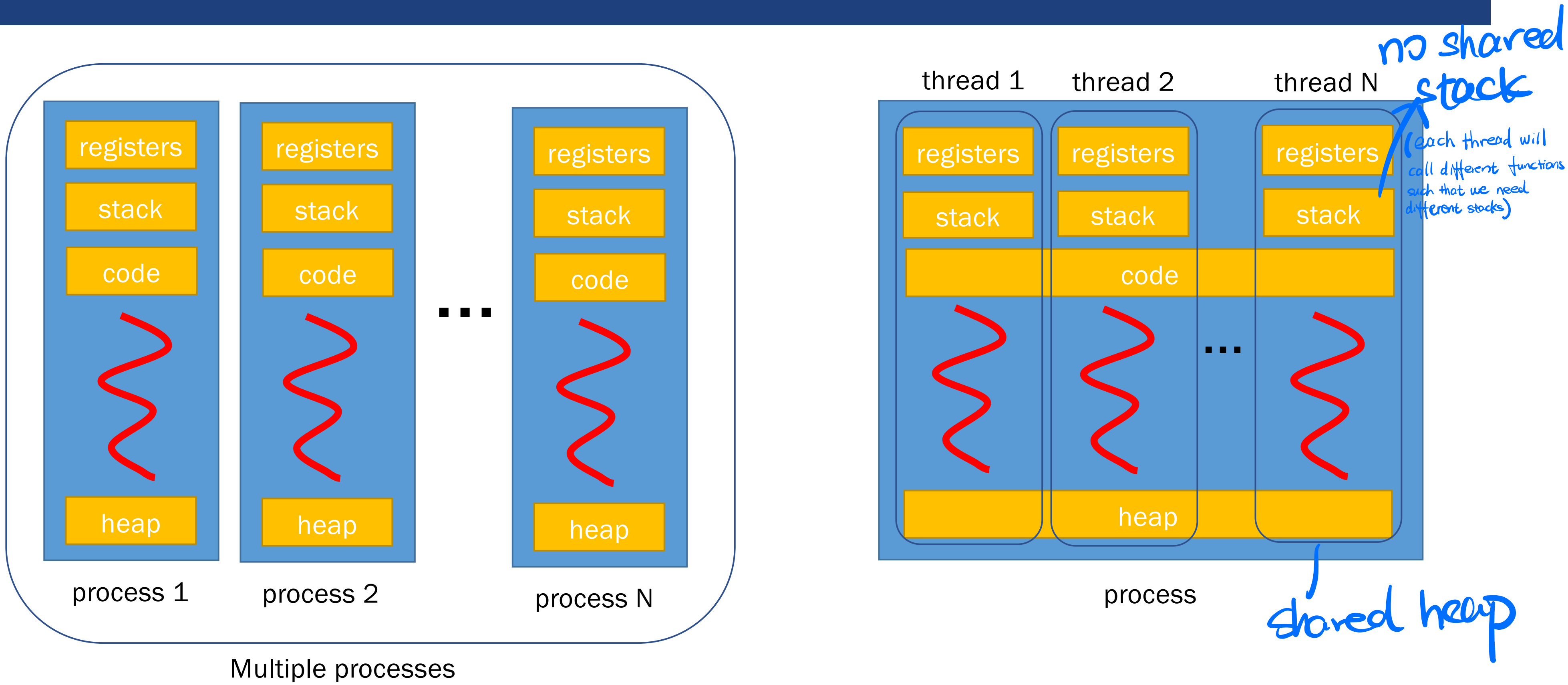
Note: you can create as many threads as you want (up to a system limit)

Threads

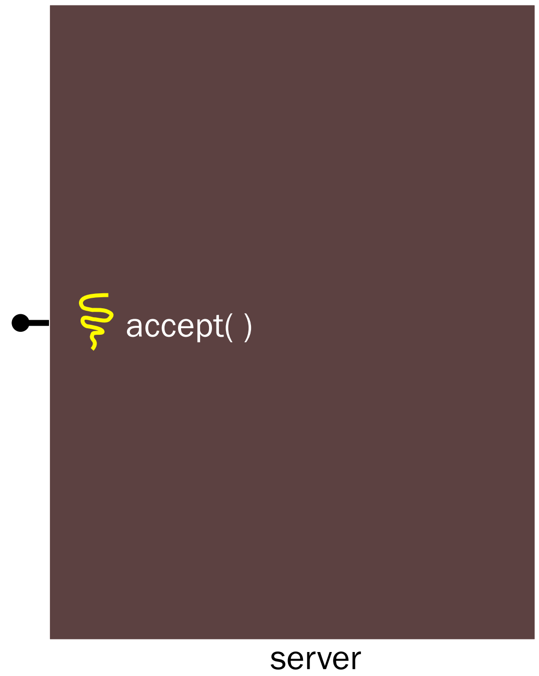
- A thread is defined as an $\boxed{\text{independent stream of instructions}}$ that can be scheduled to run as such by the operating system.
  - To the software developer, the concept of a "procedure" that runs independently from its main program.
  - To go one step further, imagine a main program that contains a number of procedures. Now imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "*multi-threaded*" program.

Idea: "forking" multiple threads of execution in one process!

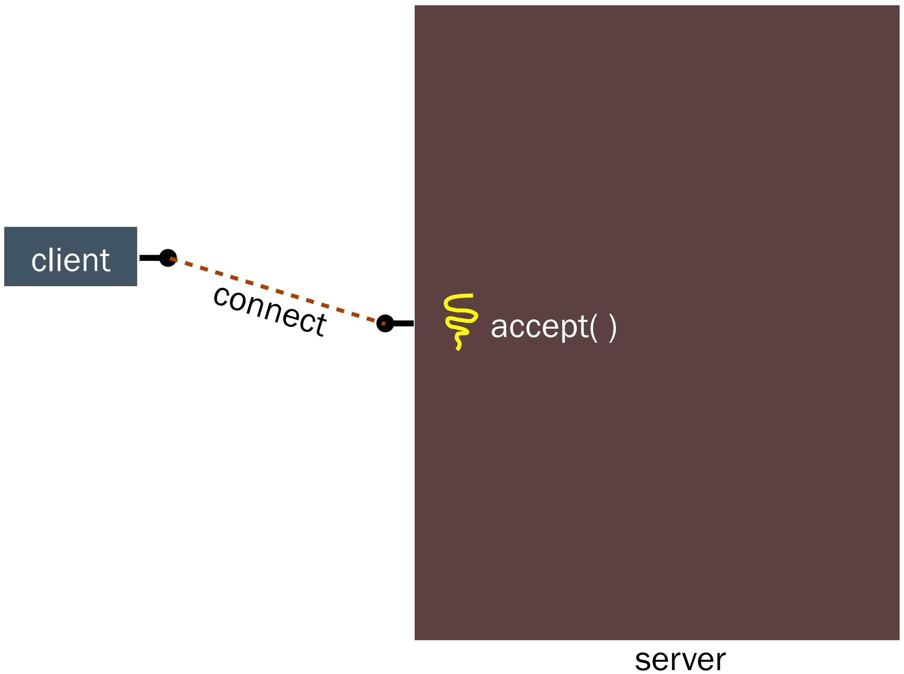
Multiple Processes vs Multiple Threads



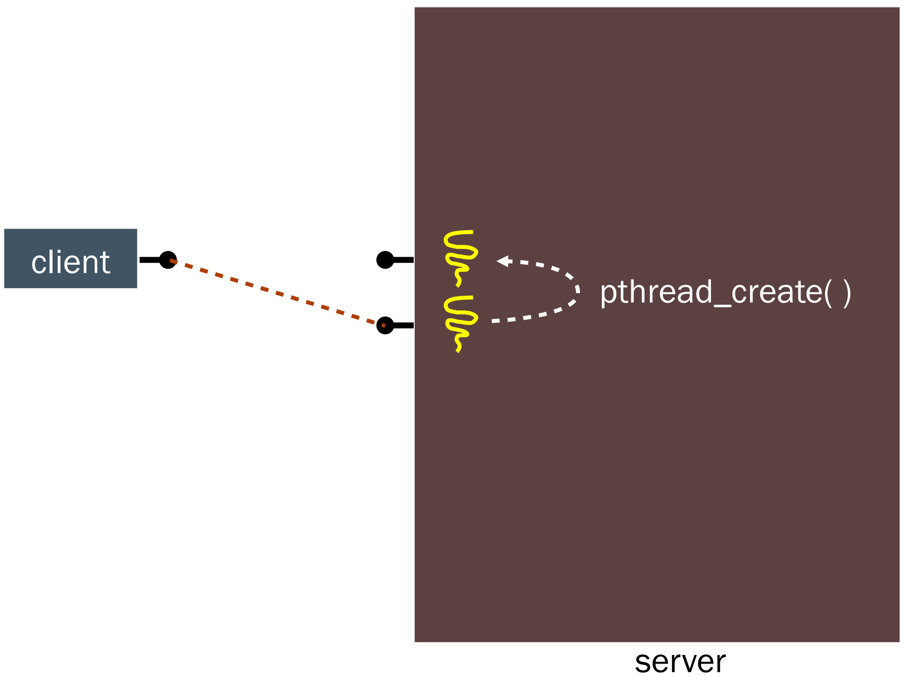
Threads (cont.)



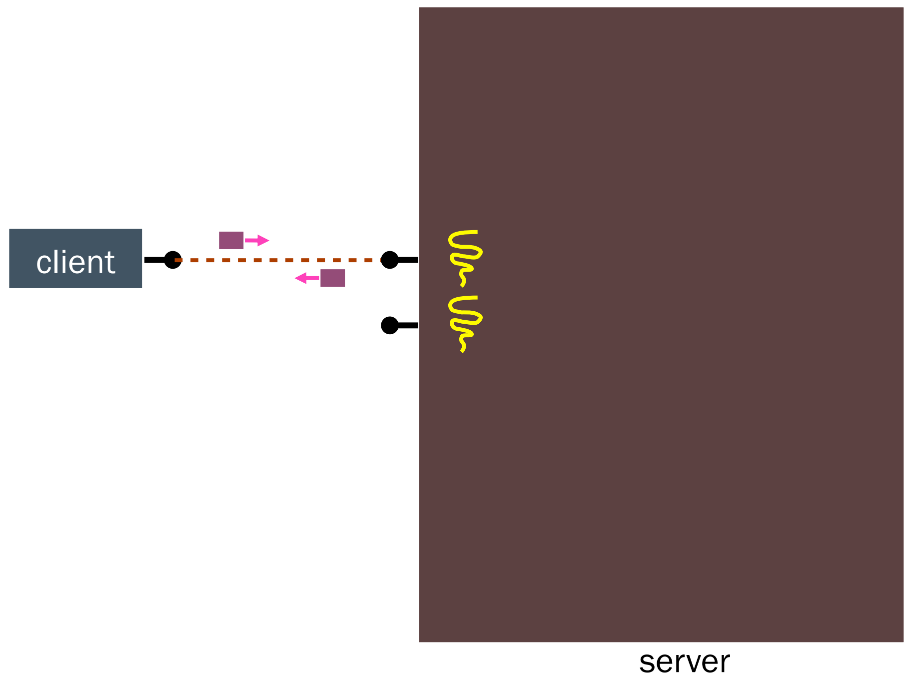
Threads (cont.)



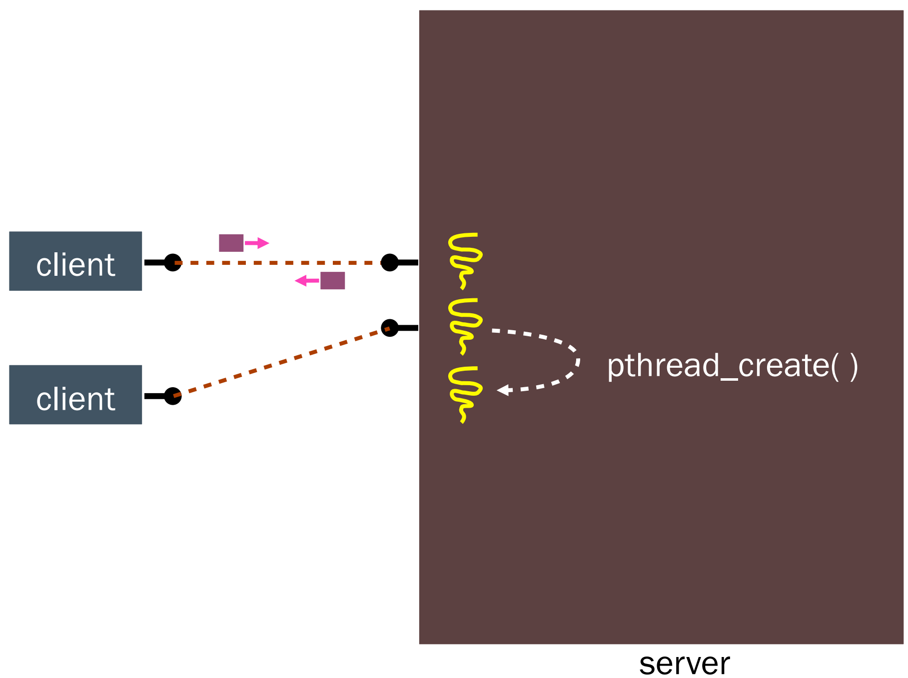
Threads (cont.)



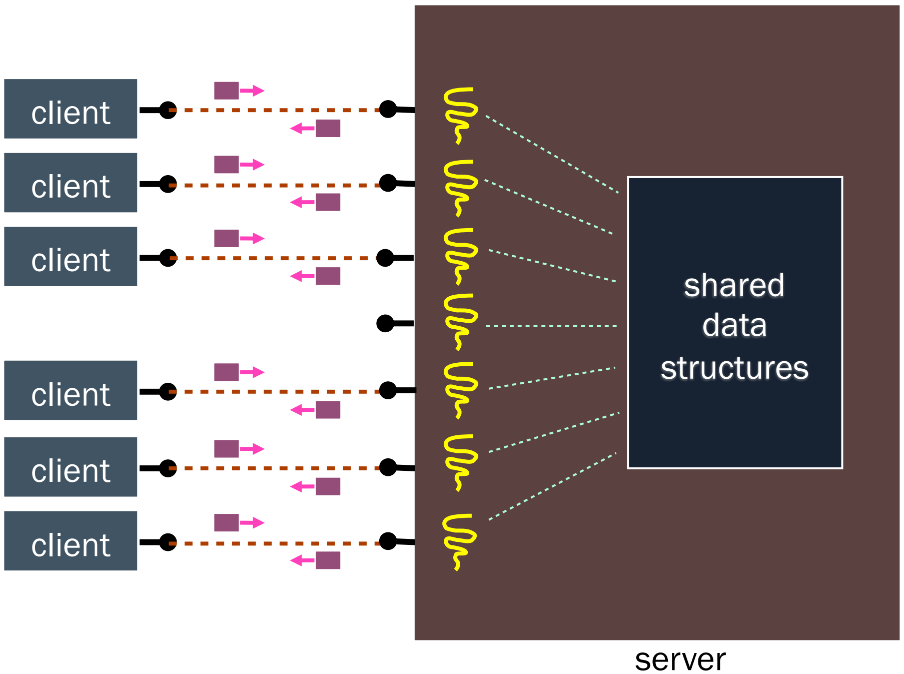
Threads (cont.)



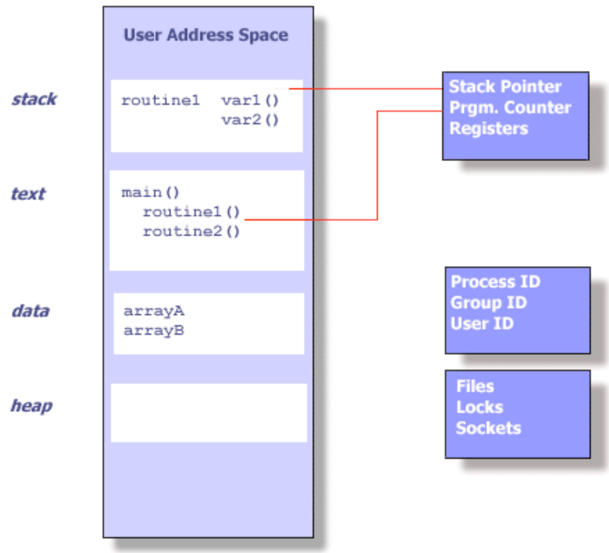
Threads (cont.)



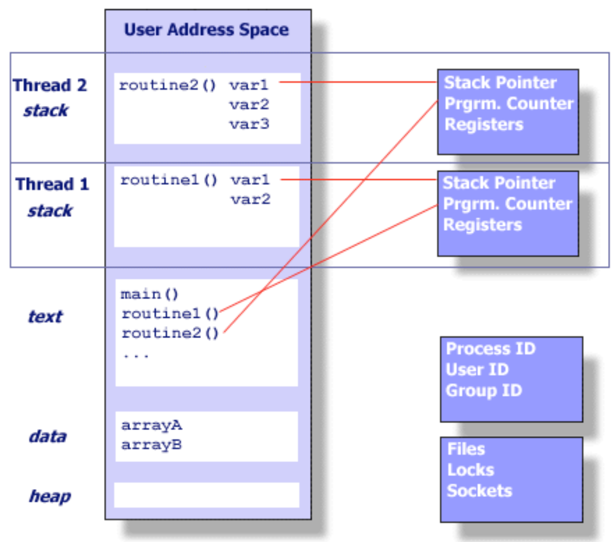
Threads (cont.)



Threads (cont.)



UNIX Process



... and with threads

Threads (cont.)

- This independent flow of control is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.

stack pointer
Register
Scheduling properties
set of pending and blocking signals
Thread specific data

Thread Summary

PennState

- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently “schedulable“
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies - or something similar
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

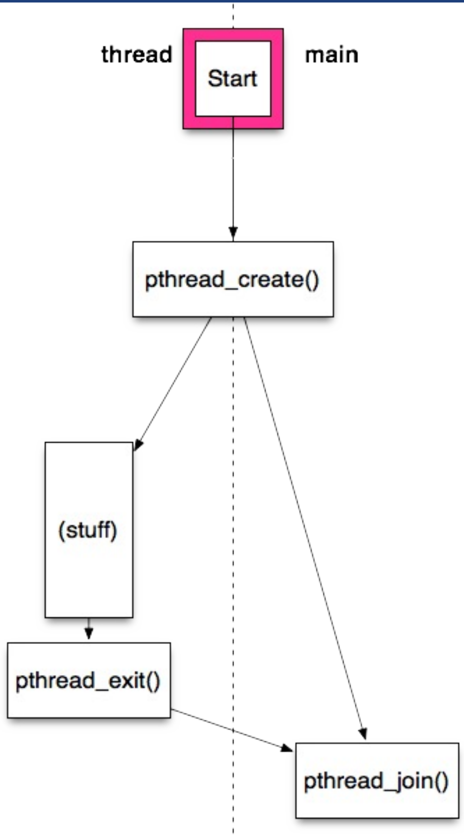
Caveats

- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

Warning: shared data between threads can cause conflicts, deadlocks, etc.

Thread control

- main
  - pthread_create() (create thread)
  - wait for thread to finish via pthread_join() (maybe)
- thread
  - begins at function pointer
  - runs until the return or pthread_exit()
- Library support
  - #include <pthread.h>
  - Compile with option –lpthread to link with the pthread library



pthread_create()

- The `pthread_create` function starts a new thread in the calling process.

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

- Where,
  - `thread` is a pthread library structure holding thread info
  - `attr` is a set of attributes to apply to the thread
  - `start_routine` is the thread function pointer
  - `arg` is an opaque data pointer to pass to thread

Thread with no arguments

```
void *func(void *arg) {
    printf("Hello from thread %lx\n", pthread_self());
    return NULL;
}

int main(void) {
    pthread_t t1, t2, t3;
    printf("main thread %lx starting a new thread\n", pthread_self());
    pthread_create(&t1, NULL, func, NULL);
    pthread_create(&t2, NULL, func, NULL);
    pthread_create(&t3, NULL, func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    return 0;
}
```

```
$ gcc -Wall t1.c -o t1 -lpthread
[0s euclid:~/tmp (master)]
$ ./t1
Hello from thread 7f475f9c7700
Hello from thread 7f475e6a5700
Hello from thread 7f475f036700
[0s euclid:~/tmp (master)]
```

```
$ ./t1
Hello from thread 7f2a426fb700
Hello from thread 7f2a4308c700
Hello from thread 7f2a41d6a700
[0s euclid:~/tmp (master)]
$
```

- Always check return values (omitted for brevity)
- Thread becomes alive in pthread_create – may even run before it returns

Thread with one argument

```
void *func(void *arg) {
    char *s = (char *) arg;
    printf("Hello from thread %s\n", s);
    return NULL;
}

int main(void) {
    pthread_t t1, t2, t3;
    pthread_create(&t1, NULL, func, "a");
    pthread_create(&t2, NULL, func, "b");
    pthread_create(&t3, NULL, func, "b");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    return 0;
}
```

```
$ ./t2
Hello from thread b
Hello from thread c
Hello from thread a
[0s euclid:~/tmp (master)]
$
```

```
$ ./t2
Hello from thread b
Hello from thread a
Hello from thread c
[0s euclid:~/tmp (master)]
$
```

- Run the above program in a loop to observe indeterminate scheduling

Thread with multiple arguments

```
typedef struct {
    int num;
    const char *str;
} foo_t;

void *func(void *arg) {
    foo_t *val = arg;
    printf("thread %lx was passed values %d, %s\n", pthread_self(), val->num, val->str);
    return NULL;
}

int main(void) {
    foo_t v = {5678, "bar"};
    pthread_t t;
    printf("main thread %lx starting a new thread\n", pthread_self());
    pthread_create(&t, NULL, func, &v);
    pthread_join(t, NULL);
    return 0;
}
```

```
$ ./t3
main thread 7f46f18d0740 starting a new thread
thead 7f46f18cf700 was passed values 5678, bar
[0s euclid:~/tmp (master)]
$
```

- The above is effectively a procedure call – real programs are more complex

pthread_join()

- The pthread_join function waits for the thread specified by thread to terminate.

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

- Where,
  - thread is a pthread library structure holding thread info
  - retval is a double pointer return value

pthread_join is called from another thread (usually the thread that created it) to wait for a thread to terminate and obtain its return value.

Returning values from a thread

```
typedef struct {
    int num;
    char *str;
} foo_t;

void *func(void *arg) {
    foo_t *a = arg;
    foo_t *b = malloc(sizeof(foo_t));
    b->num = a->num * 2;
    b->str = malloc(strlen(a->str) + 1);
    strcpy(b->str, a->str);
    for (char *p = b->str; *p; ++p)
        *p = toupper(*p);
    return b;
}

int main(void) {
    foo_t v = {5678, "bar"}, *p;
    pthread_t t;
    printf("main thread %lx starting a new thread\n", pthread_self());
    pthread_create(&t, NULL, func, &v);
    pthread_join(t, (void **) &p);
    printf("thread returned num = %d, str = %s\n", p->num, p->str);
    return 0;
}
```

```
$ ./t4
main thread 7ff9ec110740 starting a new thread
thread returned num = 11356, str = BAR
[0s euclid:~/tmp (master)]
$
```

Returning values from a thread

```
typedef struct {
    int num;
    const char *str;
} foo_t;

void *func(void *arg) {
    foo_t p;
    // fill p
    return &p;
}

int main(void) {
    foo_t v = {5678, "bar"}, *p;
    pthread_t t;
    printf("main thread %lx starting a new thread\n", pthread_self());
    pthread_create(&t, NULL, func, &v);
    pthread_join(t, (void **) &p);
    printf("thread returned num = %d, str = %s\n", p->num, p->str);
    return 0;
}
```

- The above will segfault! Do not return a pointer to a stack-allocated variable

`pthread_exit()`

- The `pthread_exit` function terminates the calling thread and returns a value

```
void pthread_exit(void *retval);
```

- Where,
  - `retval` is a pointer to a return value

pthread_exit is called from the thread itself to terminate its execution (and return result) early.

Threads accessing shared data

```
static int counter = 0;

void *func(void *arg) {
  for (int i = 0; i < 5000; ++i)
    ++counter;
  return NULL;
}

int main(void)
{
    pthread_t t1, t2;
    printf("counter = %d\n", counter);
    pthread_create(&t1, NULL, func, NULL);
    pthread_create(&t2, NULL, func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("both threads completed, counter = %d\n", counter);
    return 0;
}
```

```
main: counter = 0
main: both threads completed, counter = 10000

main: counter = 0
main: both threads completed, counter = 7401

main: counter = 0
main: both threads completed, counter = 9552
```

- What will this program output?

What is happening? A race condition!

multiple thread accessing one critical section at the same time.

- Race condition happens when the outcome of a program depends on the interleaving of the execution of multiple threads accessing critical section
- Critical section is a piece of code that accesses a shared variable and must not be concurrently executed by more than one thread

a piece of code that can only be accessed by one thread at each time

```
mov    0x2e50(%rip),%eax       # 4014 <counter>
add    $0x1,%eax
mov    %eax,0x2e47(%rip)       # 4014 <counter>
```

- Each instruction executed atomically → All writes to the same location are seen in the same order by all processers/threads
- Multiple threads executing the above instructions can result in different interleavings (and outcomes) due to uncontrollable OS scheduling

One Possible Interleaving

OS	Thread 1	Thread 2	PC	eax	counter
			(after instruction)		
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Avoiding race conditions

- To avoid race conditions we need to ensure that only a single thread can execute a critical section at any given time
- For simple cases we can use atomics (#include <stdatomic.h>)
  - modifying a variable results in a single CPU instruction
- In general, however, a critical section may contain complex logic
- We need primitives for mutual exclusion – a guarantee that only one thread is executing the critical section while others are prevented from doing so
- One way to achieve mutual exclusion is using locks:

```
lock_t mutex
lock(&mutex)
critical section
unlock(&mutex)
```

① Atomic Exchange: address for Lock

```
$t0=1
Lock: EXCH $t0, 0($s0)
      bne $t0, zero, Lock
```

② Using SC and LL (store condition, Load Linked)

```
Lock: LL $t0, 0($s0)
      bne $t0, zero, Lock
      addi $t0, $t0, 1
      SC $t0, 0($s0)
      beq $t0, zero, Lock
```

Lock $\boxed{0/1}$: 0 → unlocked, 1 → Locked

Threads accessing shared data

- Fixing race condition using atomics

```
#include <stdatomic.h>

static atomic_int counter = 0;

void *func(void *arg) {
  for (int i = 0; i < 5000; ++i)
    ++counter;
  return NULL;
}

int main(void)
{
    pthread_t t1, t2;
    printf("counter = %d\n", counter);
    pthread_create(&t1, NULL, func, NULL);
    pthread_create(&t2, NULL, func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("both threads completed, counter = %d\n", counter);
    return 0;
}
```

Threads accessing shared data – Fixed!

PennState

- Fixing race condition using mutexes

```
static int counter = 0;
pthread_mutex_lock_t lock = PTHREAD_MUTEX_INITIALIZER;

void *func(void *arg) {
  for (int i = 0; i < 5000; ++i) {
    pthread_mutex_lock(&lock);
    ++counter;
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}

int main(void)
{
    pthread_t t1, t2;
    printf("counter = %d\n", counter);
    pthread_create(&t1, NULL, func, NULL);
    pthread_create(&t2, NULL, func, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("both threads completed, counter = %d\n", counter);
    return 0;
}
```

Threads tradeoffs

PennState

- Benefits
  - still the case that much of the code is identical!
  - parallel execution; good CPU, network utilization
  - lower overhead than processes
  - shared-memory communication is possible
- Disadvantages
  - synchronization is complicated
  - shared fate within a process; one rogue thread can hurt you
  - security (no isolation)

- We scratched the surface – more advanced usage will be taught in 473

code is identical
good cpu, network utilization
less heavy than multi process

synchronization is complicated
not very safty
shared fate with a process
one rogue thread will hurt you