So far

Processes do not, as a rule, share memory with one another.
(Exception: read-only pages that -- since there is no writing -- act functionally as if they weren't shared)
What if you want two processes to share read-write memory?

Why share memory?
For latency hiding when waiting for external actions (disk, network, ...)
For efficient communication during parallel computation with multiple cores in one machine.
To enable producer-consumer architectures.
Threads

○ A process includes an execution context containing
  ▪ Memory map
  ▪ PC and register values.
○ Switching between memory maps can take time.
○ Idea: share the memory map between a group of cooperating "lightweight processes".
  ▪ Do maintain separate PC, registers.
  ▪ Do maintain separate stacks.
  ▪ Don't maintain separate segments for other memory.
  ▪ All "lightweight processes" share most of memory.
  ▪ In a multi-core environment, threads can execute simultaneously.
  ▪ (Threads are the most used mechanism for parallel computing.)
○ Voila: we have the concept of a "thread".
○ Reading: Stallings Chapter 4.
Review of the memory map for a process

- Local variables
- Dynamic storage
- Global variables
- Program
- Read-only
- Saved context
- Unique to each thread
- Grows down w/calls of subroutine
- Grows up w/calls of malloc/stocking
A thread (of execution) within a process is an execution subcontext that

- Shares a memory map with other threads in the process.
- Has a separate PC and register values.
- Has its own execution stack.
○ PCB: process control block
○ TCB: thread control block contains
  ○ pointer to parent PCB
  ○ PC/registers for thread
  ○ stack location in memory map.
Common thread uses
   Hiding latency for multiple I/O requests.
   optimizes runtime.
   Enabling Producer/Consumer program architecture.
   simplify coding.
Latency hiding

Make multiple requests to servers.
Get answers asynchronously, one per thread.
Combine at end.
"fork-join parallelism"
Used in web browsers.
Latency hiding example

Latency hiding example

Browser

http: get

Response

Server processing time.
Without threads

Must wait for each response
Must maintain each connection
With threads:

```
T1
T2

overlapping requests.
```
There is a programming paradigm that is unique to threads:
  A producer creates things to do
  A consumer does them
This is a very interesting method, especially when there are multiple producers and multiple consumers.

Example: computer audio
Producer: tells the program what sounds to play.
Consumer: plays those sounds in a strict predetermined timing.

Nearly impossible to do this without threads. Easy with threads.
Why threads?

Producer and consumer share memory (unlike separate processes, which must communicate via a pipe).
More options for P/C interaction than with pipes.
Much faster than using pipes, which are stored in kernel memory.
#include <pthread.h>

struct inputdata { // input to thread
    char name[20];
};

struct outputdata { // output from thread
    int idno;
} x; // must have lifetime greater than thread

void *threaded_routine (void * v) {
    struct inputdata *d = (struct inputdata *)v;
    printf ("hello from the thread!\n");
    printf ("my name is %s\n",d->name);
    sleep(5);
    printf ("setting idno\n");
    x.idno = 42;
    printf ("bye from the thread!\n");
    return (void *) &x;
}

main()
{
    pthread_t thread;
    printf("hello from the parent... creating thread\n");
    strcpy(input.name,"George");
    struct outputdata *retptr;
}
strcpy(input.name,"George");
struct outputdata *retptr;
if (pthread_create( &thread, NULL,
    threaded_routine, (void *)&input)==0) {
    printf("pretend to do useful work....\n");
    pthread_join(thread,(void **)&retptr);
    printf("got id number %d\n",retptr->idno);
} else {
    printf("could not create thread!\n");
}
printf("bye from the parent\n");

See  [http://www.cs.tufts.edu/comp/111/examples/Threads/pthreads1.c](http://www.cs.tufts.edu/comp/111/examples/Threads/pthreads1.c)
Whoa there!
Lots of sophisticated stuff going on here:

```c
void *threaded_routine (void * v);
```
Threaded routine can return a pointer to any type; one must cast it!

In main:
```
pthread_create( &thread, NULL,
threaded_routine, (void *)&input)
&thread: pointer to place to store thread id.
threaded_routine: subroutine to call.
(void *)&input: a pointer to the subroutine argument, with its type "forgotten".
```

In threaded_routine:
```
struct inputdata *d
= (struct inputdata *)v;
"remember the type" of the argument.
```
Alternative: function pointer casts. Suppose we have:

```c
struct outputdata * sewingmachine
    (struct inputdata *d) { 
    ...
}
```

Which means we have the types:

```c
struct outputdata (*)(struct inputdata *)
```

But we want:

```c
void (*)(void *)
```

Solution:

```c
pthread_create( &thread, NULL,
    (void (*)(void *))sewingmachine,
    (void *)&input);
```
The void * arguments forget type information, which is remembered later.

In threaded routine:

```c
return (void *) &x;
```

Forget the type of x!

In main:

```c
pthread_join(thread, (void **) &retptr);
```

Remember that the return value is a (void *); but cast the returned pointer to the proper type.
Threads are dangerous:

```c
#include <pthread.h>

int x=0;

void *threaded_routine (void * v) {
    const int *n = (int *) v;
    int i;
    for (i=0; i<10; i++) {
        int y=x; y++; sleep(1); x=y;
        printf("%d: x=%d\n", *n, x);
    }
}

main()
{
    pthread_t thread1, thread2, thread3; void *retptr;
    int n1=1, n2=2, n3=3;
    pthread_create(&thread1, NULL,
        threaded_routine, (void *) &n1);
    pthread_create(&thread2, NULL,
        threaded_routine, (void *) &n2);
    pthread_create(&thread3, NULL,
        threaded_routine, (void *) &n3);
    pthread_join(thread1, (void **) &retptr);
    pthread_join(thread2, (void **) &retptr);
    pthread_join(thread3, (void **) &retptr);
    printf("x = %d (should be 30)\n", x);
}

http://www.cs.tufts.edu/comp/111/examples/Threads/pthreads2.c
```
What goes wrong?

y=x, y++; sleep(1);
    y=x, y++; sleep(1)

x=y
    \downarrow
    x=y

Result: second overwrites first.
The previous example prints "10"!

Thread 1

Thread 2

Thread 3

Oops!
Locking to prevent races:
#include <pthread.h>

pthread_mutex_t locker;
   // pthread_mutex_init(&locker, NULL);
   // pthread_mutex_lock(&locker);
   // pthread_mutex_trylock(&locker);
   // pthread_mutex_unlock(&locker);
   // pthread_mutex_unlock(&locker);
   // pthread_mutex_destroy(&locker);

int x=0;

void *threaded_routine (void * v) {
   const int *n = (int *)v;
   int i;
   for (i=0; i<10; i++) {
      pthread_mutex_lock(&locker);
      int y=x; y++; sleep(1); x=y;
      printf("%d: x=%d
",*n,x);
      pthread_mutex_unlock(&locker);
   }
   return NULL;
}

main()
{
   pthread_t thread1, thread2, thread3; void *retptr;
   int n1=1,n2=2,n3=3;
   pthread_mutex_init(&locker, NULL);
   pthread_create( &thread1, NULL,
      threaded_routine, (void *)&n1);
   pthread_create( &thread2, NULL,
      threaded_routine, (void *)&n2);
   pthread_create( &thread3, NULL,
      threaded_routine, (void *)&n3);
   pthread_join(thread1,(void **)&retptr);
   pthread_join(thread2,(void **)&retptr);
   pthread_join(thread3,(void **)&retptr);
   printf("x = %d (should be 30)
",x);
}
pthread_mutex_destroy(&locker);
}

http://www.cs.tufts.edu/comp/111/examples/Threads/pthreads3.c