Threads

Jo, Heeseung

Today's Topics

Why threads?

Threading issues

Processes

Heavy-weight

- A process includes many things:
  - An address space (all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Hardware execution state (PC, SP, registers, etc.)
- Creating a new process is costly because all of the data structures must be allocated and initialized
  - Linux: over 100 fields in task_struct
    (excluding page tables, etc.)
- Inter-process communication is costly, since it must usually go through the OS
  - Overhead of system calls and copying data

Concurrent Servers: Processes

Web server example

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task

```
while (1) {
      int sock = accept();
      if ((pid = fork()) == 0) {
            /* Handle client request */
      } else {
            /* Close socket */
      }
}
```

Cooperating Processes

Example

- A web server, which forks off copies of itself to handle multiple simultaneous tasks
- Any parallel program on a multiprocessor

We need to:

- Create several processes that execute in parallel
- Cause each to map the same address space to share data
  - e.g., shared memory
- Have the OS schedule these processes in parallel

This is very inefficient!

- Space: PCB, page tables, etc.
- Time: creating OS structures, fork and copy address space, etc.

Rethinking Processes

What's similar in these cooperating processes?

- They all use (share?) the same code and data (address space)
- They all use the same privilege
- They all use the same resources (files, sockets, etc.)

What's different?

- Each has its own hardware execution state:
  PC, registers, SP, and stack

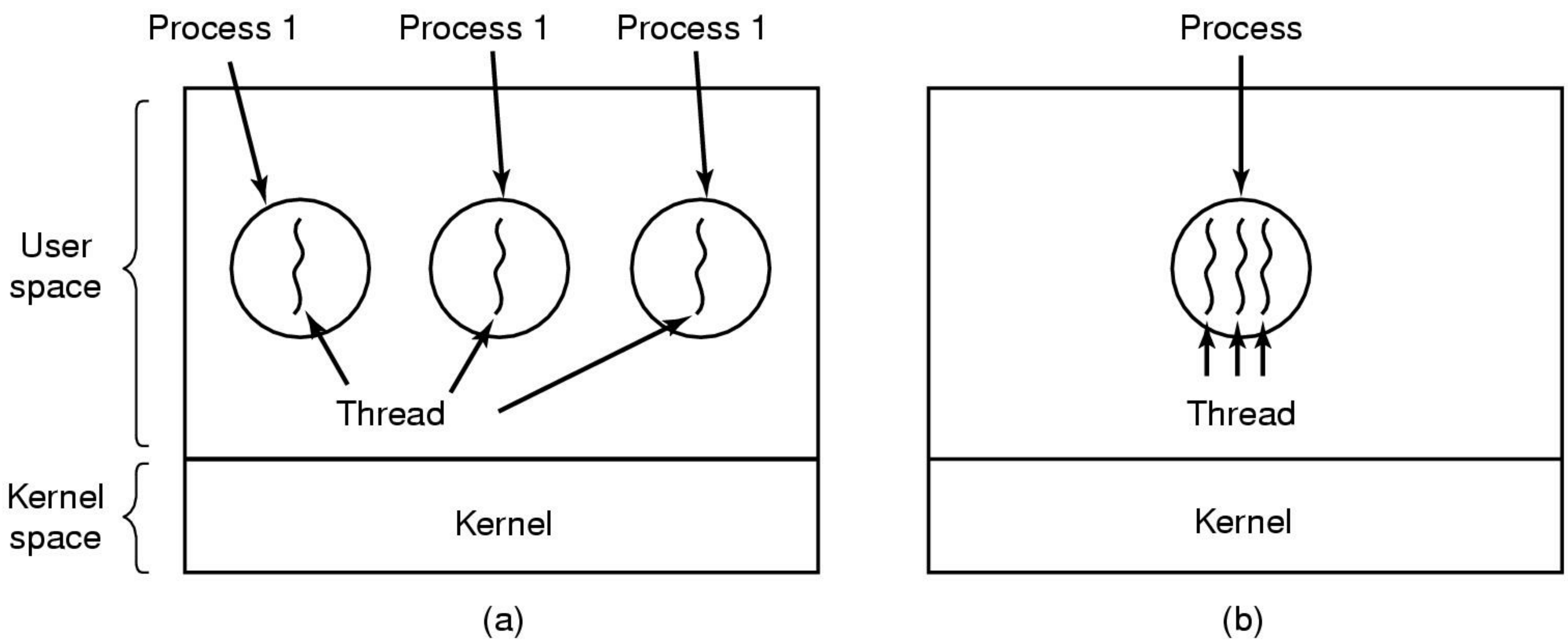
Key Idea (1)

Separate the concept of a process from its execution state

- Process: address space, resources, other general process attributes
  - e.g., privileges
- Execution state: PC, SP, registers, etc.

- This execution state is usually called
  - Thread
  - Lightweight process (LWP)
  - Thread of control

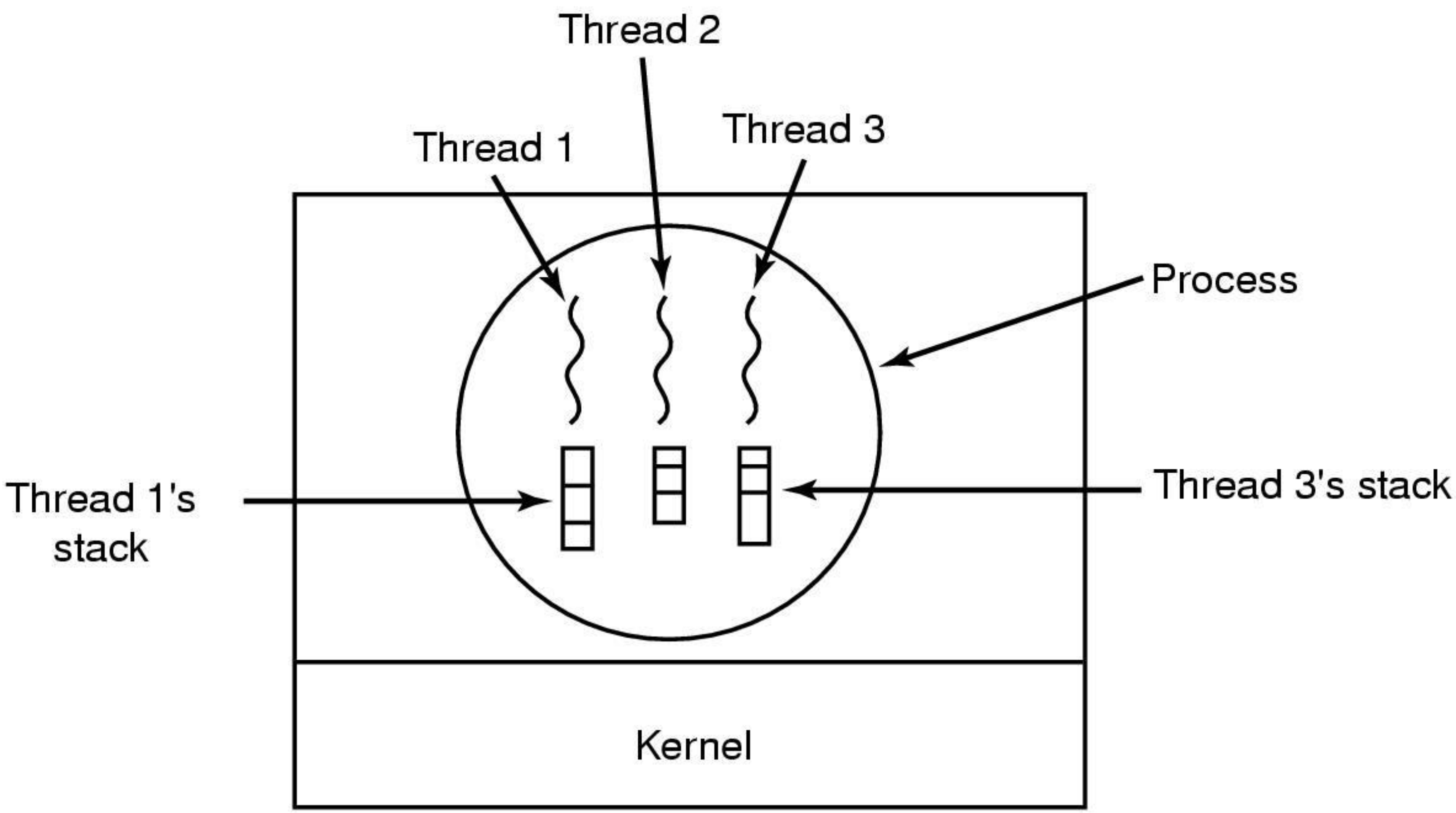
Key Idea (2)



Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

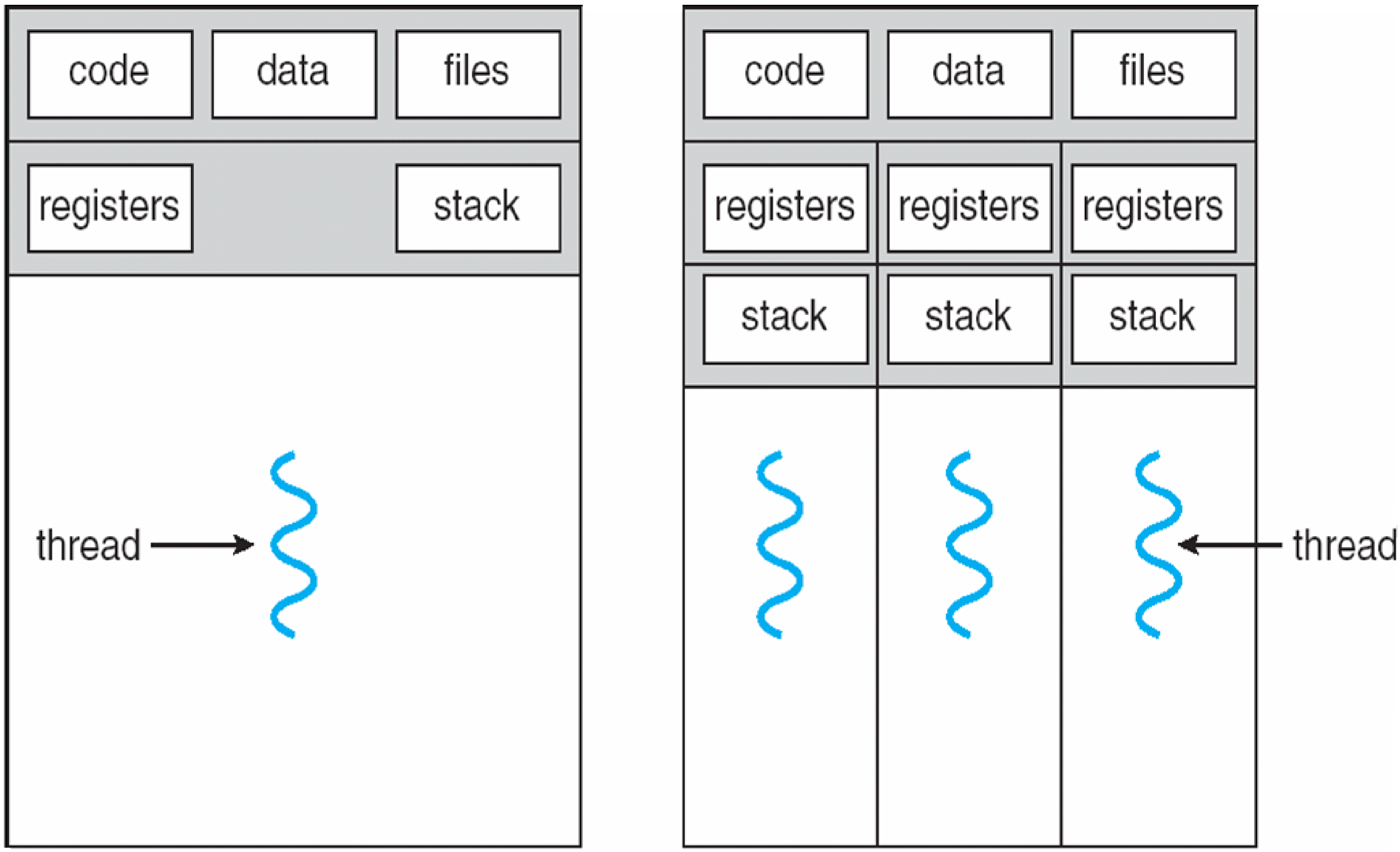
Key Idea (3)

Each thread has its own stack



Key Idea (4)

Each thread has its own stack



single-threaded process

multithreaded process

What is a Thread?

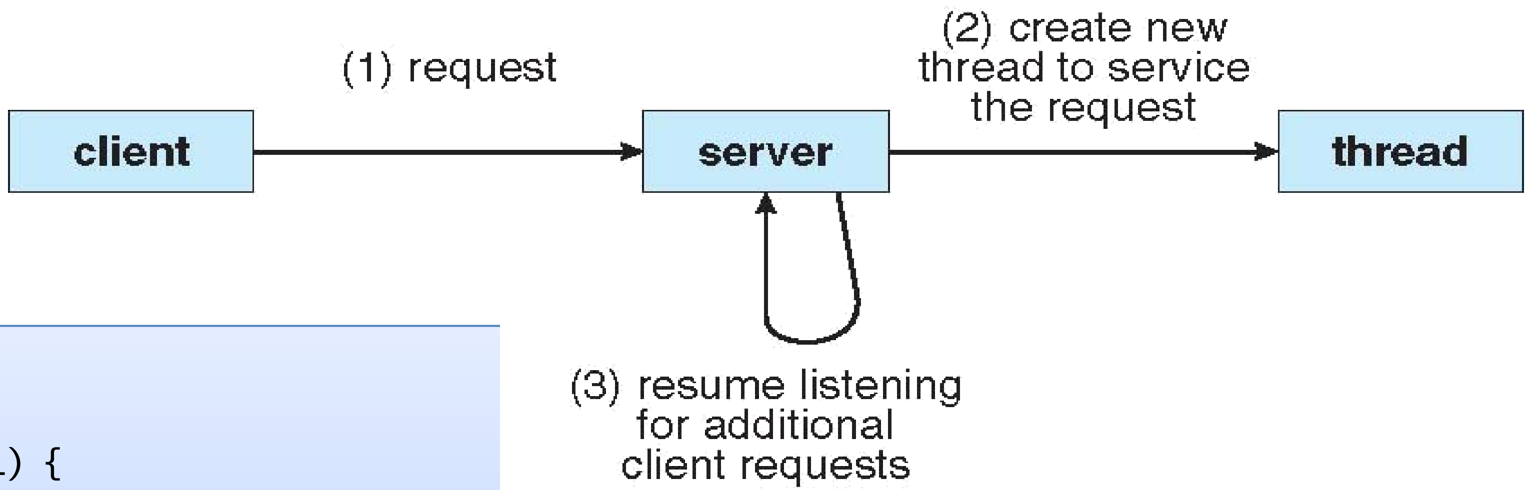
A thread of control (or a thread)

- A sequence of instructions being executed in a program
- Usually consists of
  - A program counter (PC), general registers
  - A stack to keep track of local variables and return addresses
- Threads share the process instructions and most of its data
  - A change in shared data by one thread can be seen by the other threads in the process
- Threads also share most of the OS state of a process

Concurrent Servers: Threads

Using threads

- We can create a new thread for each request




```
webserver ()
{
     while (1) {
          int sock = accept();
          create_thread (handle_request, sock);
     }
}
handle_request (int sock)
{
     /* Process request */
     close (sock);
}
```

Multithreading

Benefits

- Creating concurrency is cheap
  - Time and memory consumption
- Improves program structure
- Higher throughput
  - By overlapping computation with I/O operations
- Better responsiveness (User interface / Server)
  - Can handle concurrent events (e.g., web servers)
- Better resource sharing
- Utilization of multiprocessor architectures
  - Allows building parallel programs

Processes vs. Threads (1)

Processes vs. Threads

- A thread is bound to a single process
- A process, however, can have multiple threads
- Sharing data between threads is cheap
  - All see the same address space
- Threads become the unit of scheduling
- Processes are now containers in which threads execute

Processes vs. Threads (2)

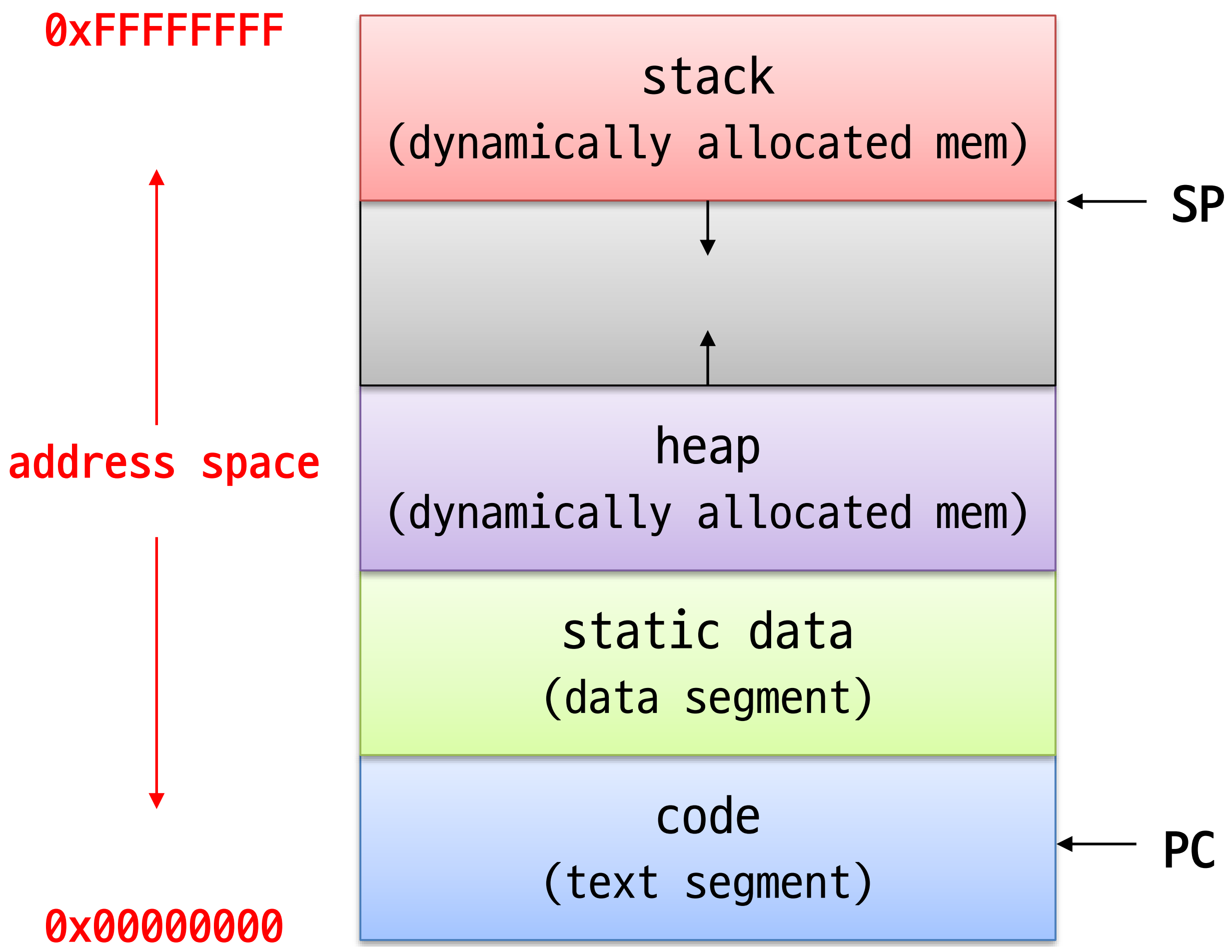
How threads and processes are similar

- Each has its own logical control flow
- Each can run concurrently with others (possibly on different cores)
- Each is context switched

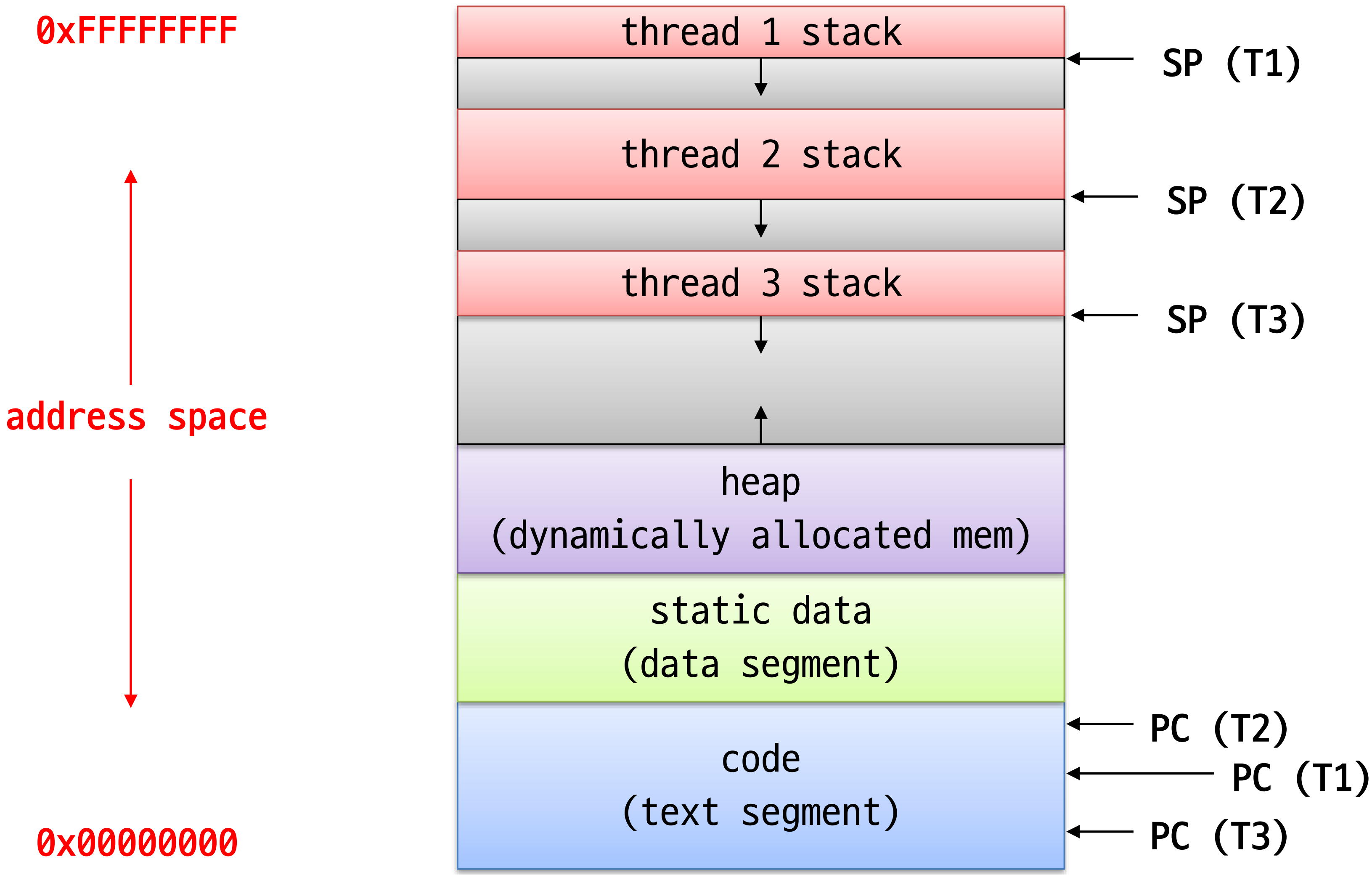
How threads and processes are different

- Threads share code and some data
  - Processes (typically) do not – use the same code and data copies
- Threads are somewhat less expensive than processes
  - Process control (creating and reaping) is twice as expensive as thread control
  - Linux numbers:
    ~20K cycles to create and reap a process
    ~10K cycles (or less) to create and reap a thread

Process Address Space



Address Space with Threads



Classification

# of addr spaces / # threads per addr space	One	Many
One	· MS-DOS · Early Macintosh	· Traditional UNIX
Many	· Many embedded OSes · VxWorks · uClinux	· Mach · OS/2 · Linux · Windows · Mac OS X · Solaris · HP-UX

Threads Interface (1)

pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library
- Implementation is up to development of the library
- Common in UNIX operating systems

Threads Interface (2)

POSIX-style threads

- pthreads
- DCE threads (early version of pthreads)
- Unix International (UI) threads (Solaris threads)
  - Sun Solaris 2, SCO Unixware 2

Microsoft-style threads

- Win32 threads
  - Microsoft Windows 98/NT/2000/XP
- OS/2 threads
  - IBM OS/2

pthreads (1)

Thread creation/termination

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

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

```
int pthread_join   (pthread_t tid,
                    void **thread_return);
```

The Pthreads "hello, world" Program

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

void *threadfunc(void *vargp);

/* thread routine */
void *threadfunc(void *vargp) {
  sleep(1);
  printf("Hello, world!\n");
  return NULL;
}

int main() {
  pthread_t tid;

  pthread_create(&tid, NULL, threadfunc, NULL);
  printf("main\n");
  pthread_join(tid, NULL);
  printf("main2\n");
  sleep(2);
  return 0;
}
```

```
# gcc ex.c -lpthread
# ./a.out
main
Hello, world!
main2
```

pthreads (2)

Mutexes

```
int pthread_mutex_init
               (pthread_mutex_t *mutex,
                const pthread_mutexattr_t *mattr);
```

```
void pthread_mutex_destroy
               (pthread_mutex_t *mutex);
```

```
void pthread_mutex_lock
               (pthread_mutex_t *mutex);
```

```
void pthread_mutex_unlock
               (pthread_mutex_t *mutex);
```

Threads using shared data

```
#include <pthread.h>
#define MAX_THREAD 20

void *threadcount(void *data) {
	int *count = (int *)data;
	int i;
	for (i=0; i<100; i++) {
		*count = *count+1;
	}
}
int main(int argc, char **argv) {
	pthread_t thread_id[MAX_THREAD];
	int i = 0;
	int count = 0;
	for(i = 0; i < MAX_THREAD; i++) {
		pthread_create(&thread_id[i], NULL, threadcount, (void *)&count);
	}
	for(i = 0; i < MAX_THREAD; i++) {
		pthread_join(thread_id[i], NULL);
	}
	printf("Main Thread : %d\n", count);
	return 0;
}
```

```
# gcc ex.c -lpthread
# ./a.out
Main Thread : 2000
# ./a.out
Main Thread : 1957
```

pthreads (3)

Condition variables

```
int pthread_cond_init
                (pthread_cond_t *cond,
                 const pthread_condattr_t *cattr);
```

```
void pthread_cond_destroy
                (pthread_cond_t *cond);
```

```
void pthread_cond_wait
                (pthread_cond_t *cond,
                 pthread_mutex_t *mutex);
```

```
void pthread_cond_signal
                (pthread_cond_t *cond);
```

```
void pthread_cond_broadcast
                (pthread_cond_t *cond);
```

Threading Issues (1)

fork() and exec() can be issue

When a thread calls fork()

- Does the new process duplicate all the threads?
- Is the new process single-threaded?

Some UNIX systems support two versions of fork()

- In pthreads,
  - fork() duplicates only a calling thread
- In the Unix international standard,
  - fork() duplicates all parent threads in the child
  - fork1() duplicates only a calling thread

Normally, exec() replaces the entire process

If a thread call exit()?

If the main thread dies(return, exit()) before child threads?

Threading Issues (2)

Thread cancellation

- The task of terminating a thread before it has completed

Asynchronous cancellation

- Terminates the target thread immediately
- What happens if the target thread is holding a resource, or it is in the middle of updating shared resources?

Deferred cancellation

- The target thread is terminated at the cancellation points
- The target thread periodically check if it should be cancelled

pthreads API supports both asynchronous and deferred cancellation

Threading Issues (3)

Signal handling

- Where should a signal be delivered?

To the thread to which the signal applies

- for synchronous signals

To every thread in the process

To certain threads in the process

- Typically only to a single thread found in a process that is not blocking the signal
- pthreads: per-process pending signals, per-thread blocked signal mask

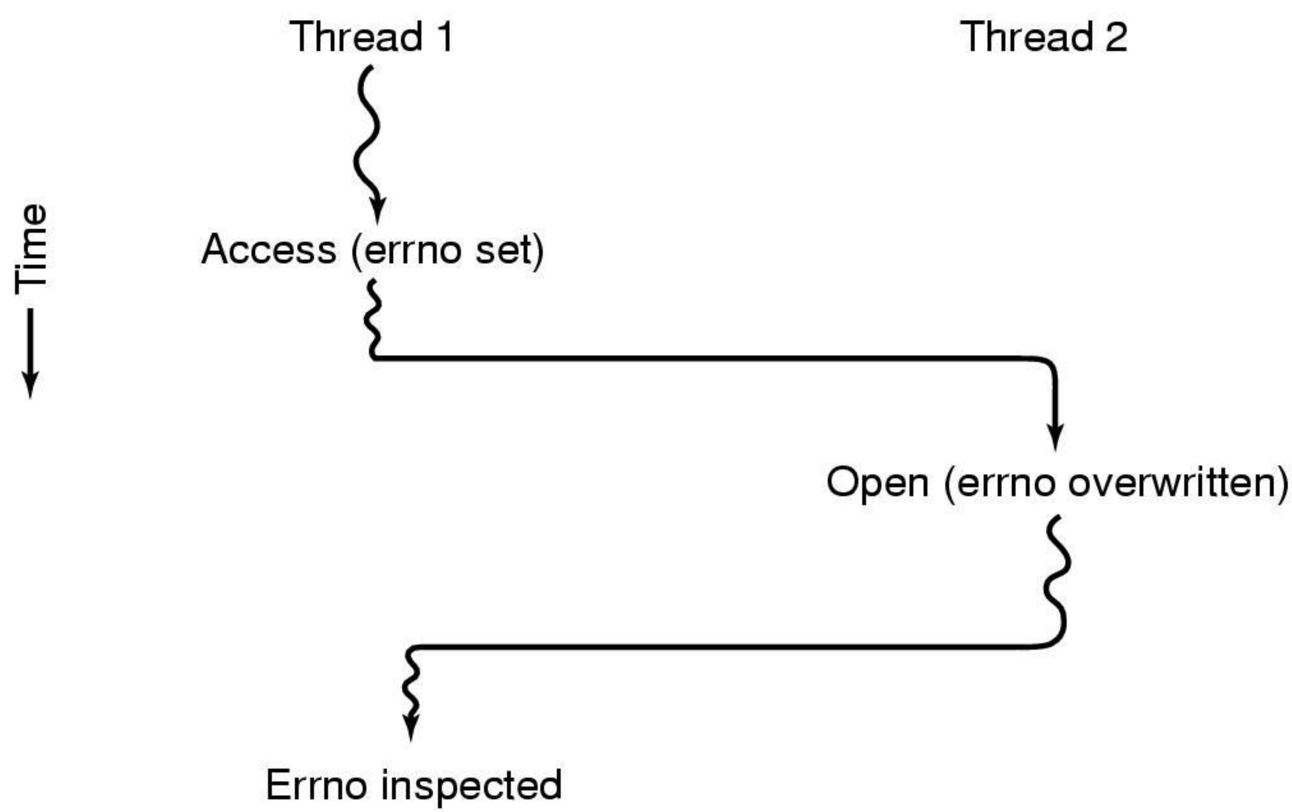
Assign a specific thread to receive all signals for the process

- Solaris 2

Threading Issues (4)

Using libraries having internal variables

- errno
  - #include <errno.h>
  - Each thread should have its own independent version of the errno variable



Threading Issues (4)

Multithread-safe (MT-safe)

- A set of functions can be said to be multithread-safe or reentrant, when the functions may be called by more than one thread at a time
- Functions that access no global data or read-only global data are trivially MT-safe
- Functions that modify global state must be made MT-safe by synchronizing access to the shared data

Threads can have private global variables