TU Dublin
Operating Systems
COMP 3602

Lecture 5
Threads

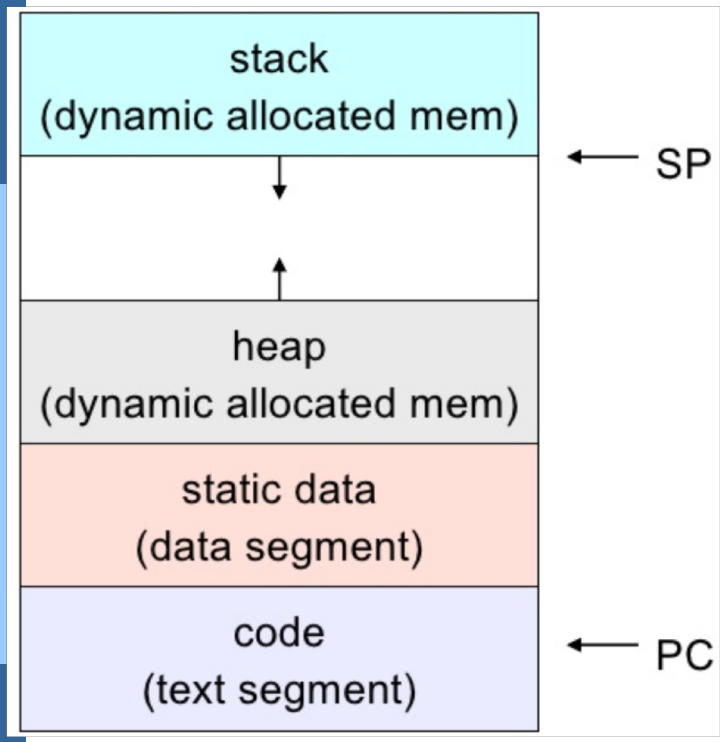
Process (definitions)

- Process is a program in execution (the unit of work in a modern computing system)
  - A program is a file stored on a hard disk and containing a list of instructions (i.e. executable file)
  - A program becomes a process when an executable file is loaded into memory.

Process components

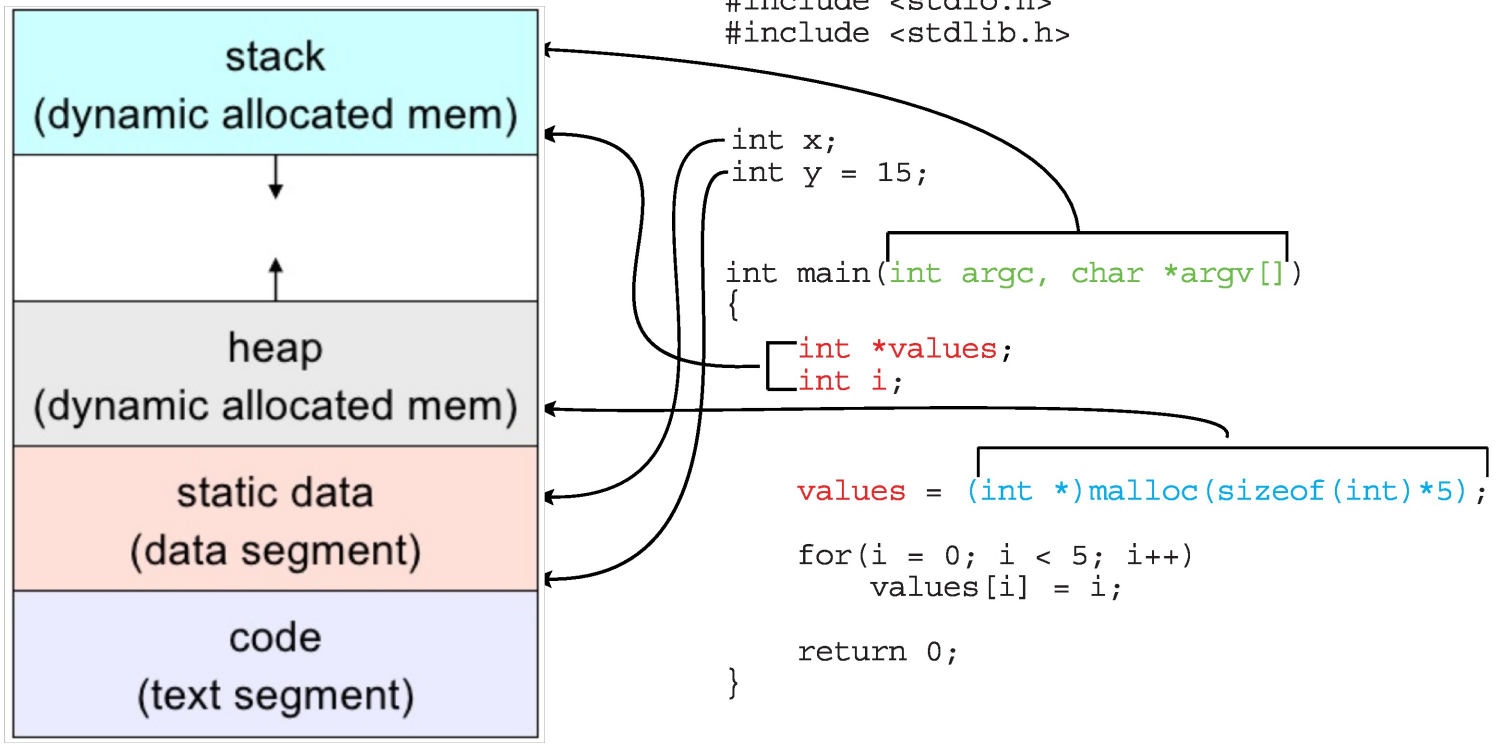
- A process consists of everything needed to run the program
  - An address space (sections in memory)
    - the data needed for the running program
      - heap
      - stack
      - static data
    - the code (instructions) for the running program
  - CPU state (values of CPU registers)
    - The program counter (PC), indicating the next instruction
    - The stack pointer (SP)
    - Other general purpose register
  - A set of OS resources
    - open files
    - I/O devices

The memory layout of a process



- Memory sections needed for the running program
  - **stack**—temporary data (local variables, function parameters, return addresses)
  - **heap**—memory that is dynamically allocated during program run time
  - **static data**—global variables
- **Text section**—the code (instructions) for the running program
- **CPU registers:**
  - The program counter (PC), indicating the next instruction
  - The stack pointer (SP)

Memory Layout of a C Program

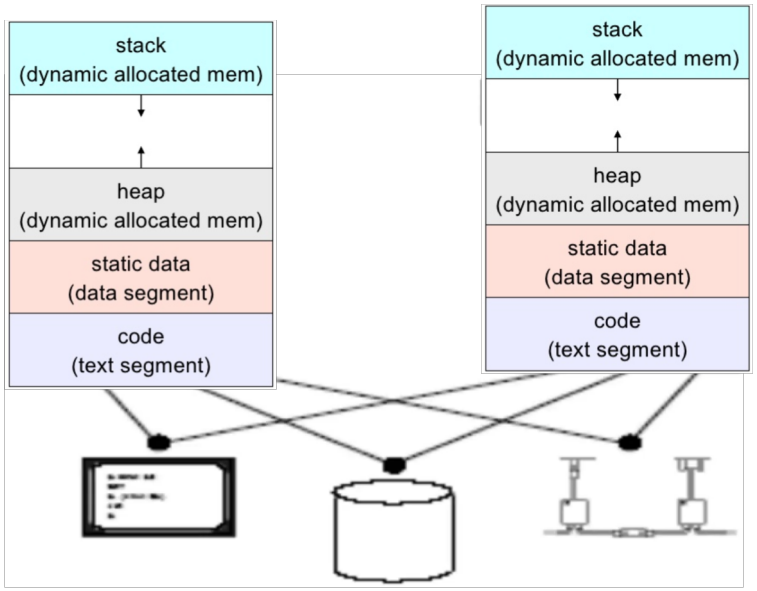


Process creation using *fork()*

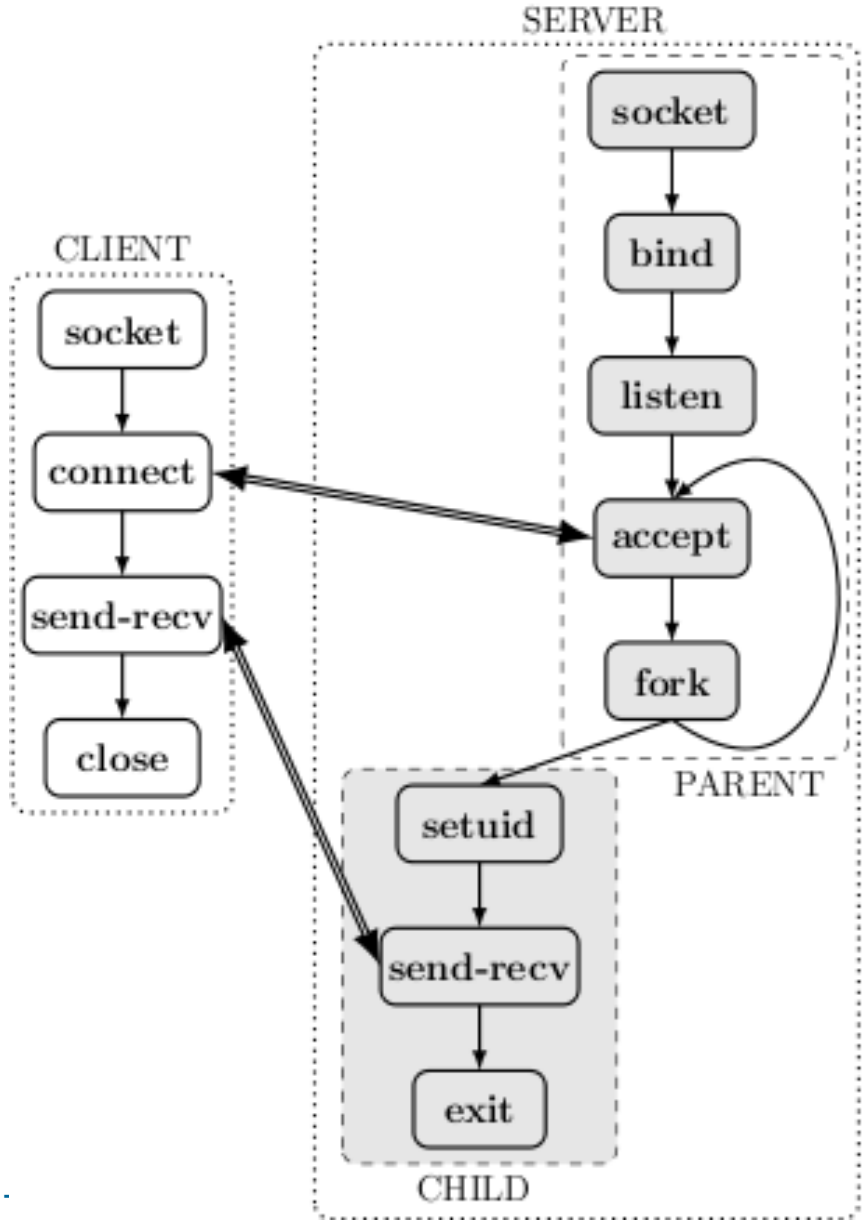
- ***fork()*** system call allows one process, ***the parent***, to create a new process, ***the child***

- ***the child process*** is a copy of the address space of ***the parent*** (code, static data, heap, stack)

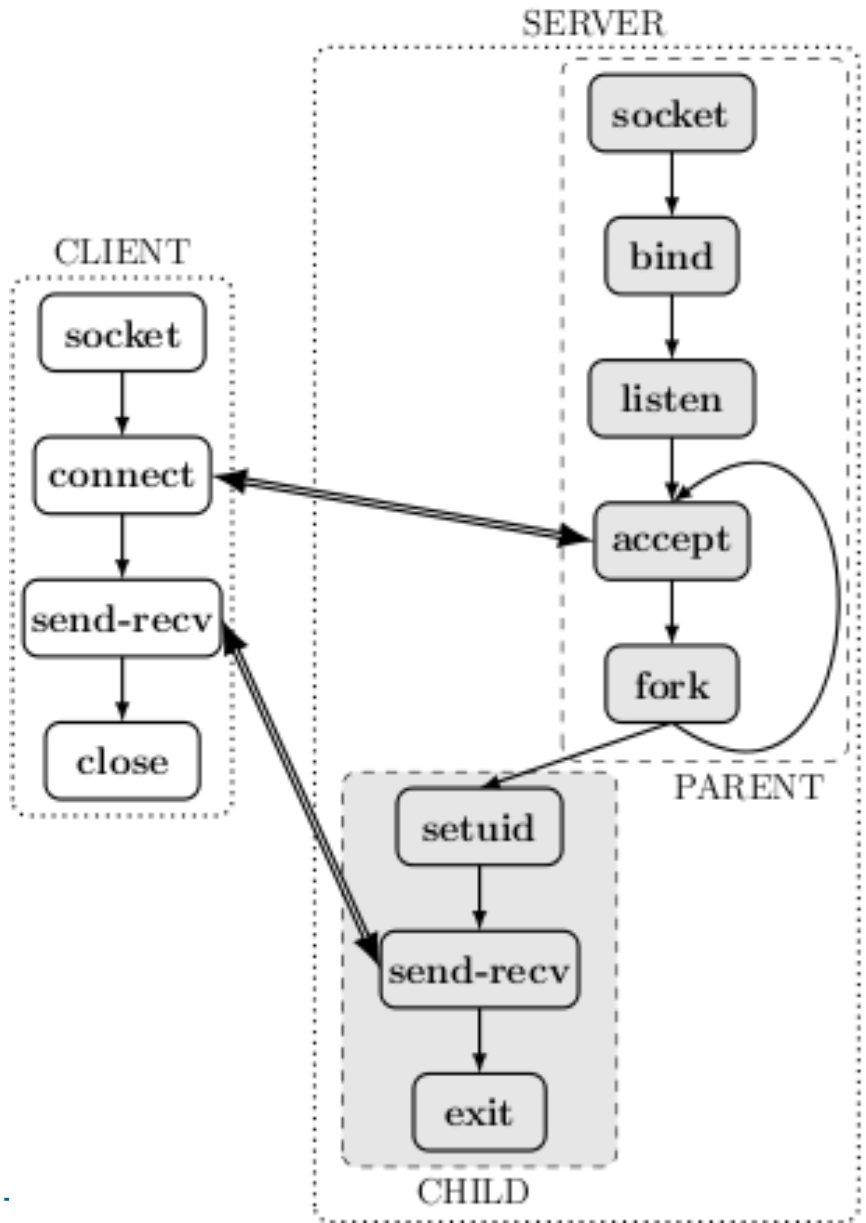
```
pid =fork();

If (pid < 0) {
    //error
}
else if (pid == 0) {
    // child process
}
else {
    // parent process
}
```

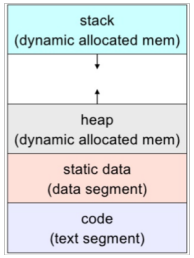


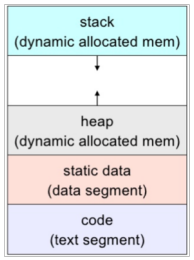
Web server with fork() system call



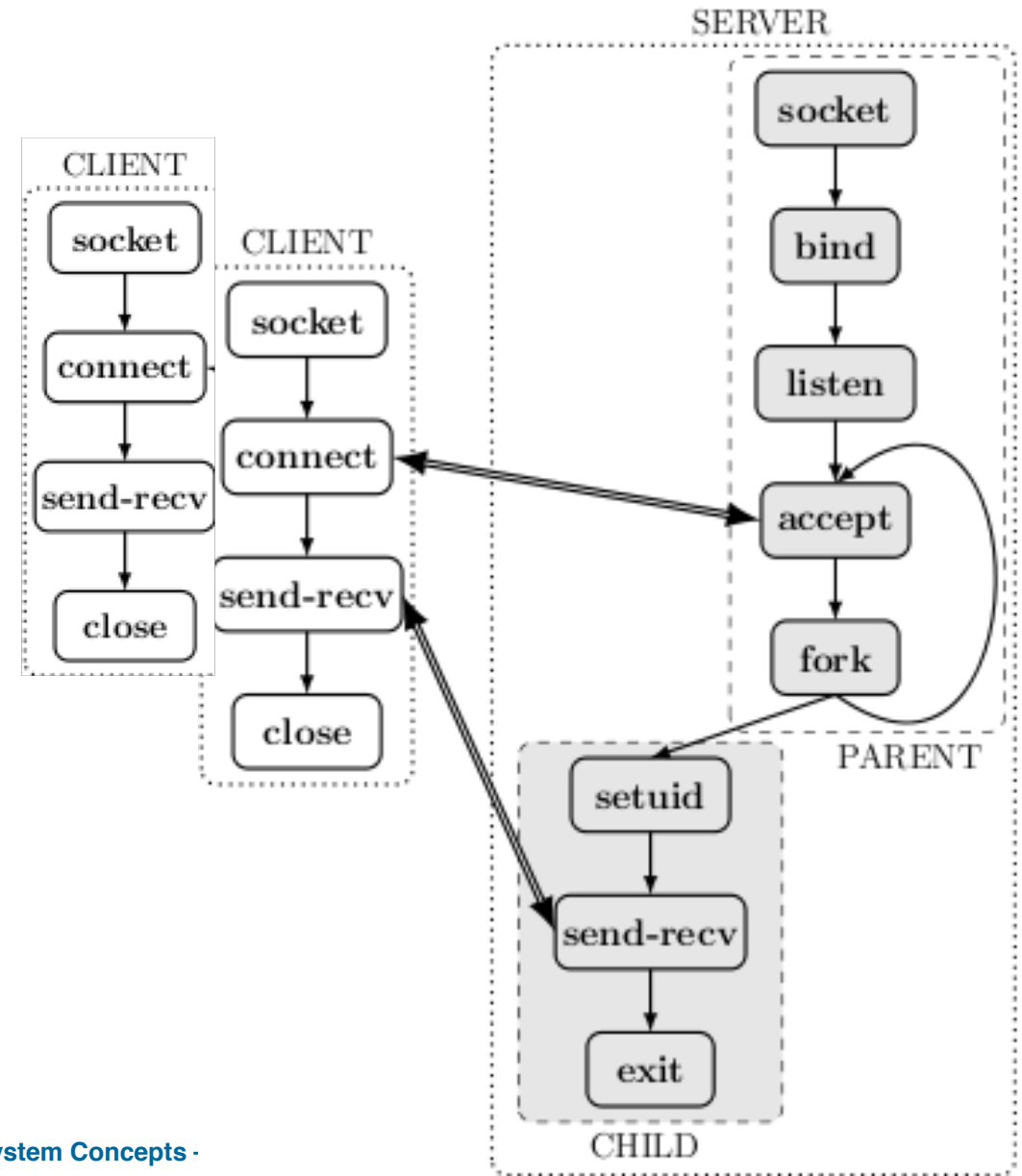
Web server with fork() system call

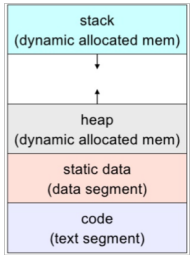


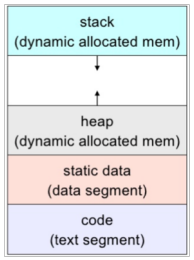


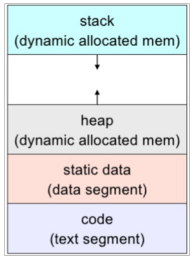


Web server with fork() system call

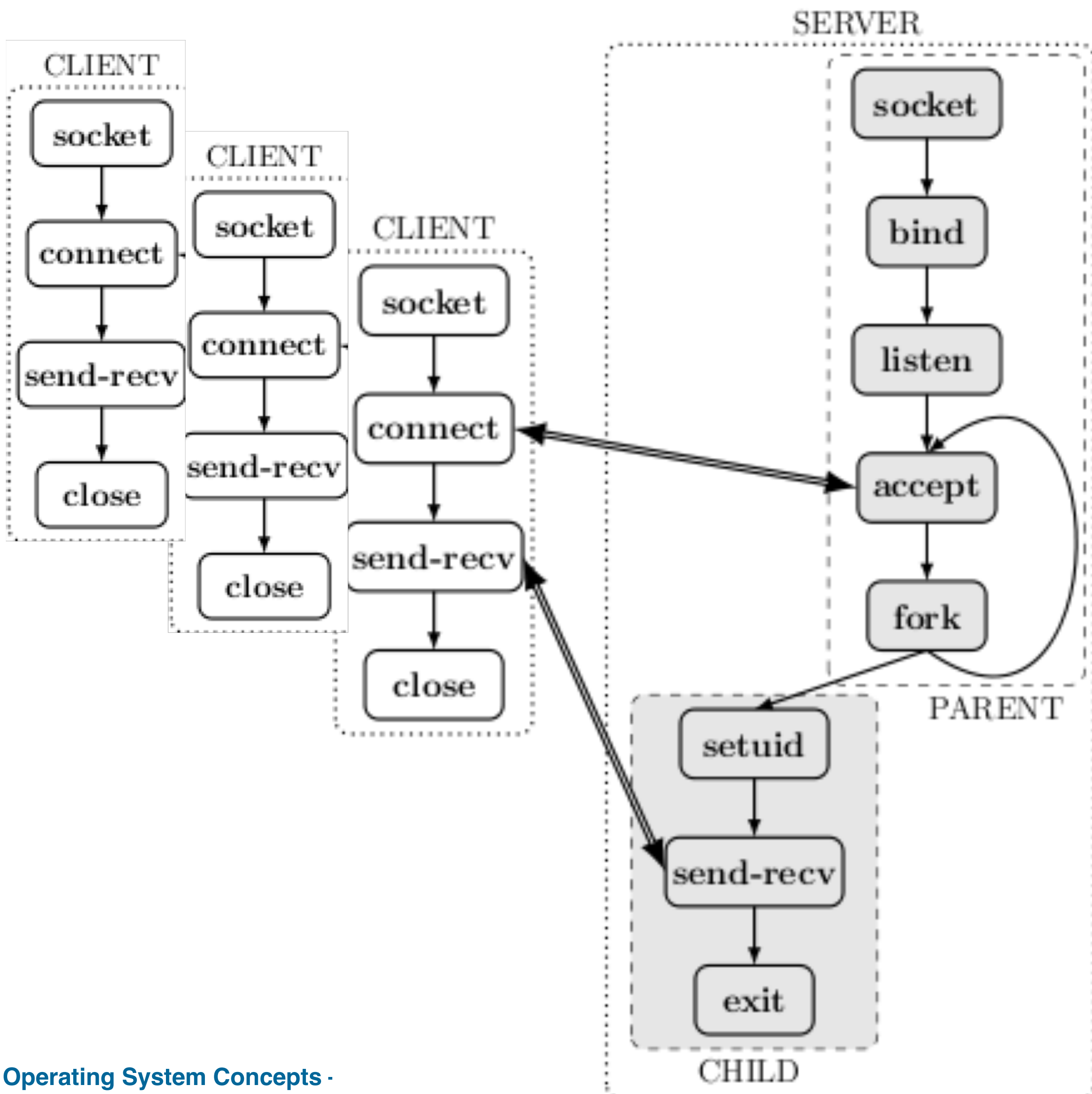


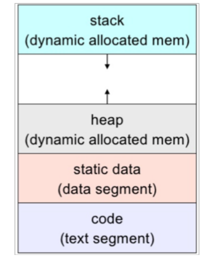


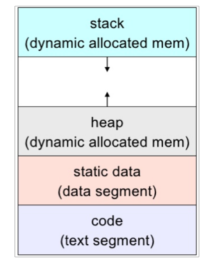


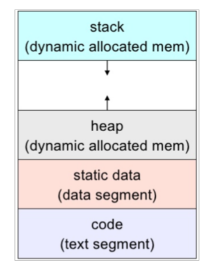


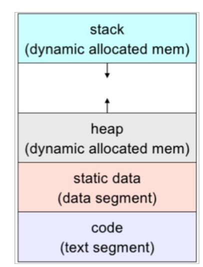
Web server with fork() system call











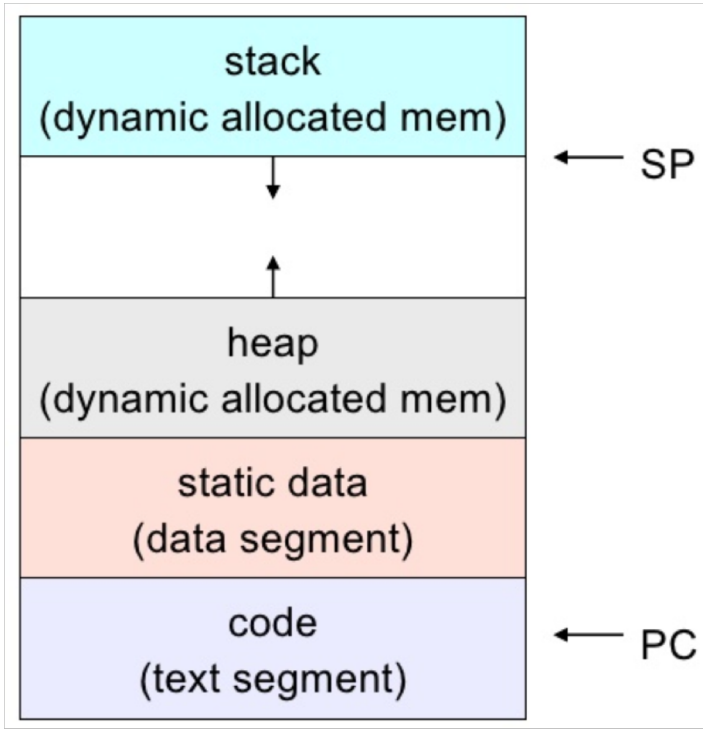
Processes overhead

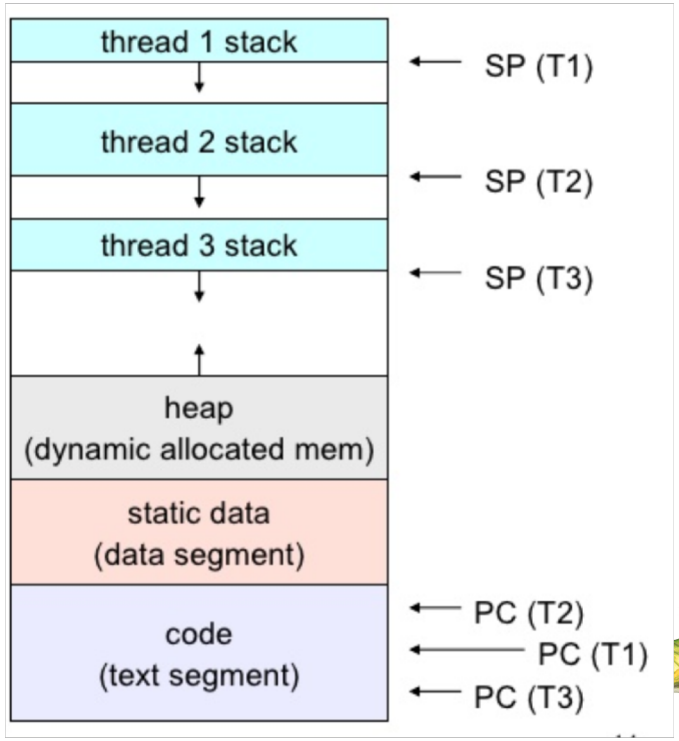
- Imagine a web server handling 1000 requests at the same time
  - 1000 child processes
  - All processes run the same code
  - All processes access the same data
  - All processes use the same resources

- What's the point in having 1000 child processes? Inefficient use of resources!
  - process creation is heavy-weight
  - memory:
    - 1000 copies of the same address space,
    - 1000 PCBs (process control blocks)
  - time:
    - copying 1000 times the same address space is time consuming!

Threads

- A **thread** is the smallest unit of processing that can be performed in an OS (a basic unit of CPU utilization).
- A traditional process performs one task at a time, i.e. has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.

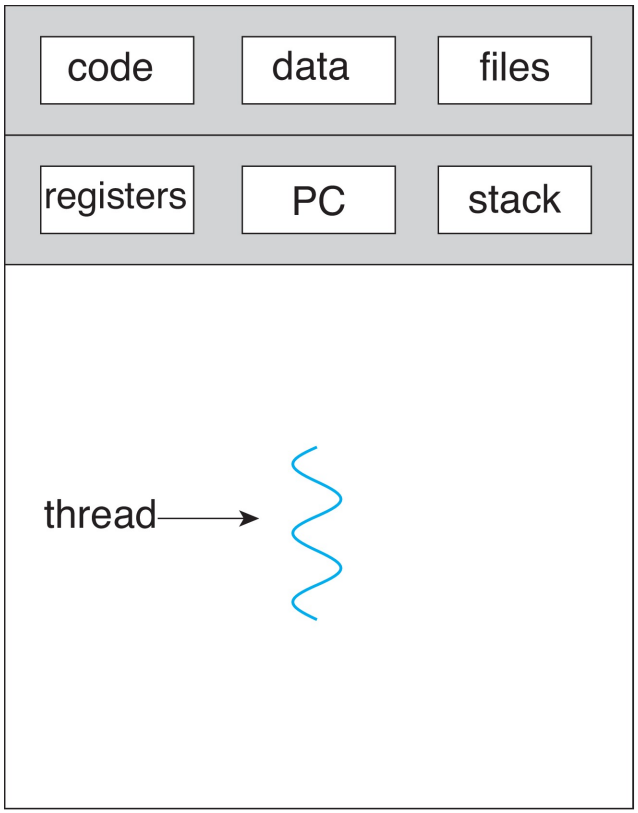




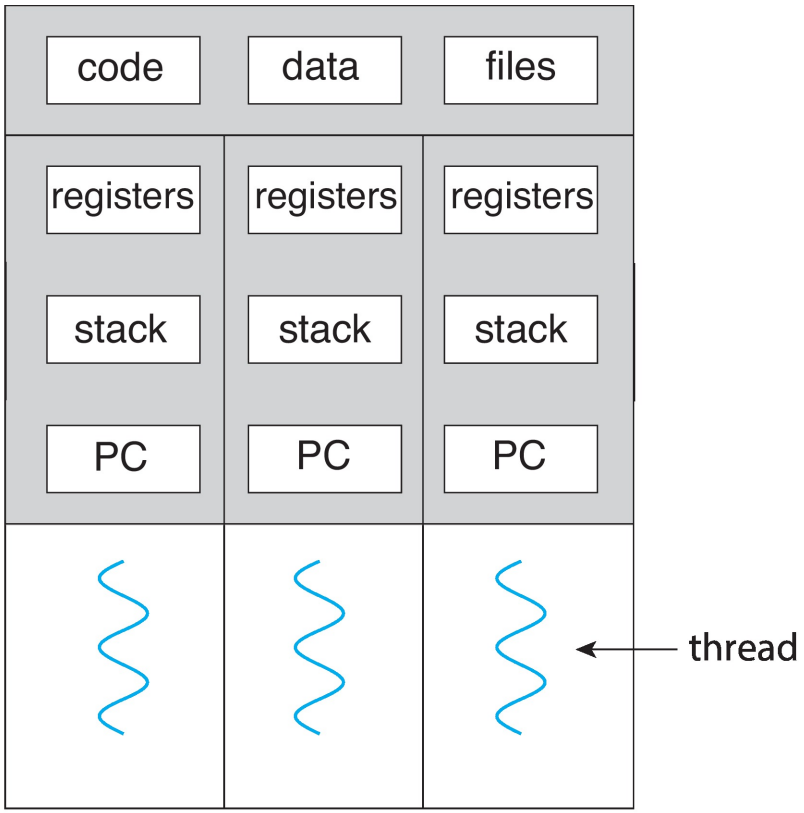
Thread components

- A thread is a flow of execution through the process code with its own:
  - thread ID
  - program counter (PC)
  - stack
  - register set
- Each thread belongs to exactly one process; cannot exist outside a process
- A thread shares with other threads belonging to the same process:
  - its code section
  - data section
  - operating-system resources
    - open files and signals
    - I/O devices

Single and Multithreaded Processes



single-threaded process



multithreaded process

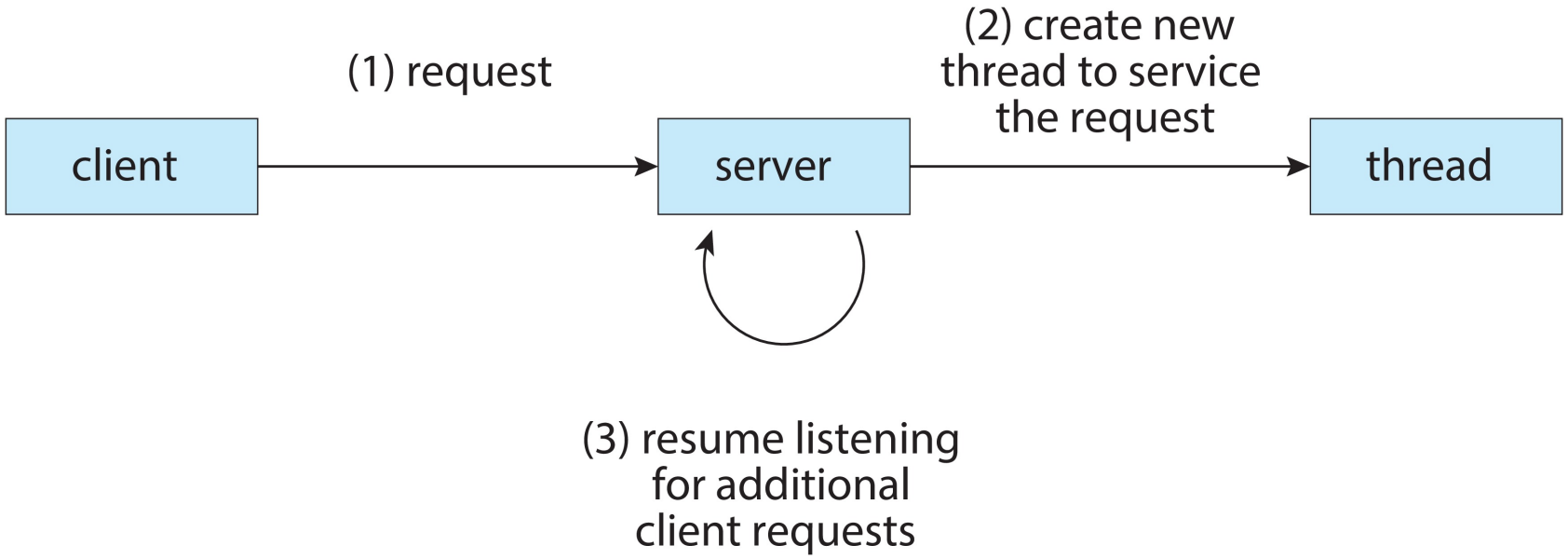
Processes and Threads

- Most modern OS's support both:
  - processes (which define the address space)
  - threads (which define a sequential execution stream within a process)
- Process creation is heavy-weight while thread creation is light-weight
- Process switching needs interaction with OS while thread switching doesn't.
- A thread runs in a single process (address space)
  - One process can have multiple threads and they can read, write, change another thread's data.
- All threads in one process share system resources
  - creating threads is quick (ca. 10 times quicker than processes)
  - ending threads is quick
  - switching threads within one process is quick
  - inter-thread communication is quick and easy (shared memory)

Benefits

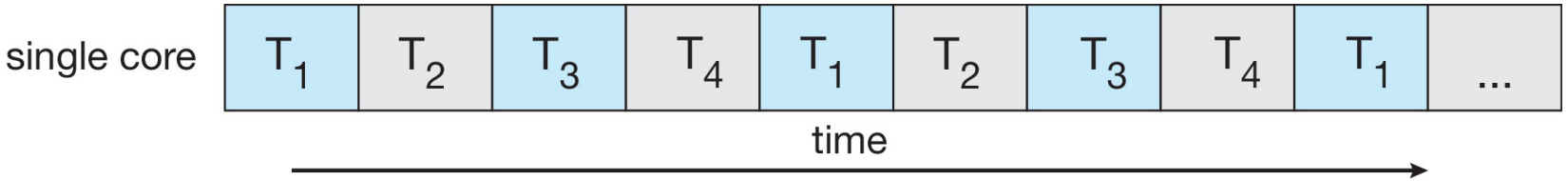
- **Economy**
  - thread creation is cheaper (time, memory) than process creation
  - thread switching has lower overhead than context switching
  - better utilization of multicore (multiprocessor) architecture
- **Resource Sharing**
  - threads share the same address space
  - no need to implement Inter-Process Communications
- **Responsiveness –** interactive applications can be performing two tasks at the same time (important for user interfaces)
- **Scalability –** multithreading can take advantage of multicore architectures

Multithreaded Server Architecture

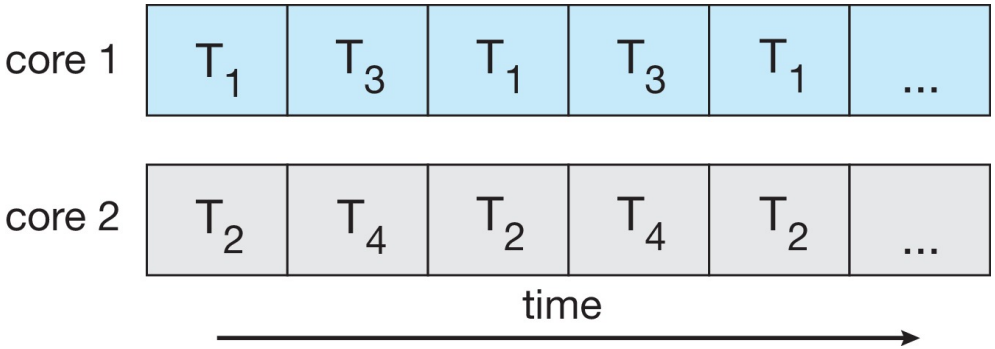


Concurrency vs. Parallelism

- **Concurrent execution on single-core system** (more than one task can make progress when a task scheduler provides concurrency)

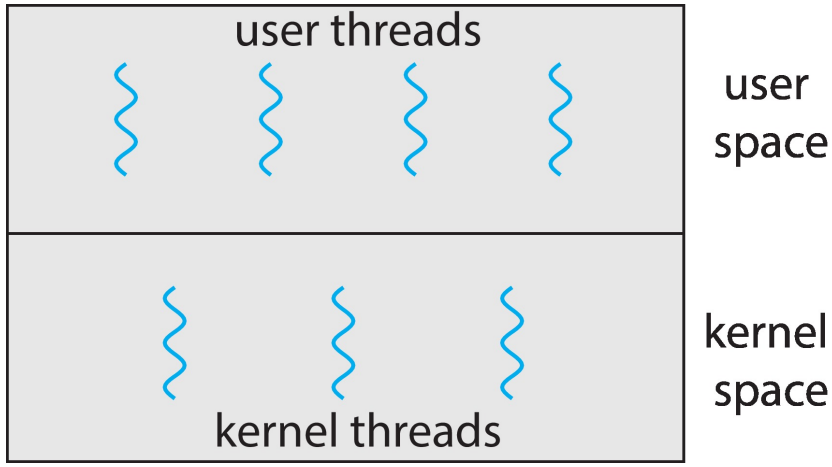


- **Parallelism on a multi-core system -** a system can perform more than one task simultaneously



Thread types

- **User threads** – managed by user-level threads library
  - Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads
- **Kernel threads** - supported by the Kernel
  - Examples:
    - Windows
    - Linux
    - Mac OS X
    - iOS
    - Android

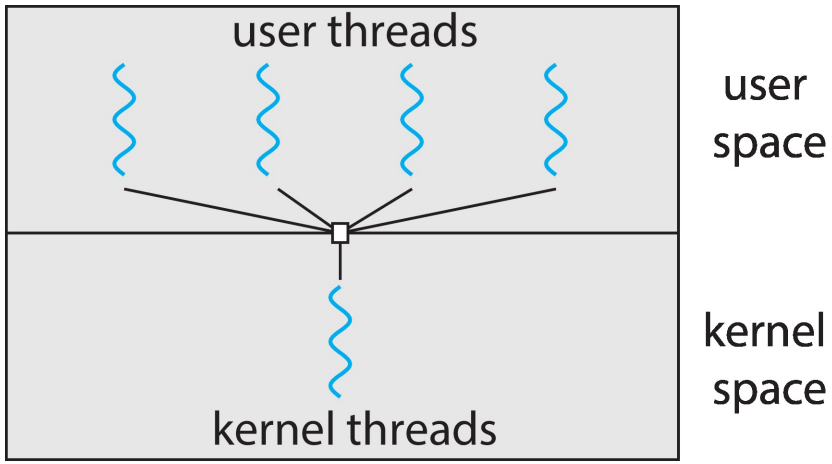


Multithreading Models

- Operating Systems provide various types of mapping between user-level threads and kernel-level threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

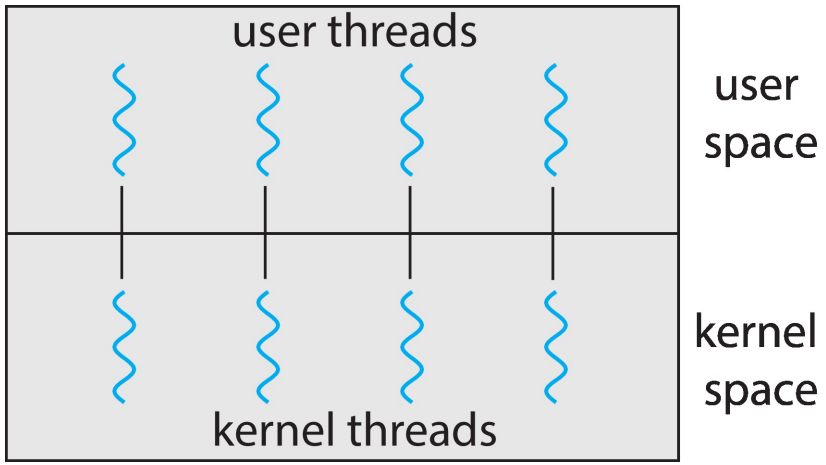
Many-to-One

- Many user-level threads mapped to single kernel thread
  - If single kernel thread blocks all the other cannot run
- Multiple threads cannot run in parallel on muticore systems becauseof single kernel thread
- Few systems currently use this model
  - **Solaris Green Threads**
  - **GNU Portable Threads**



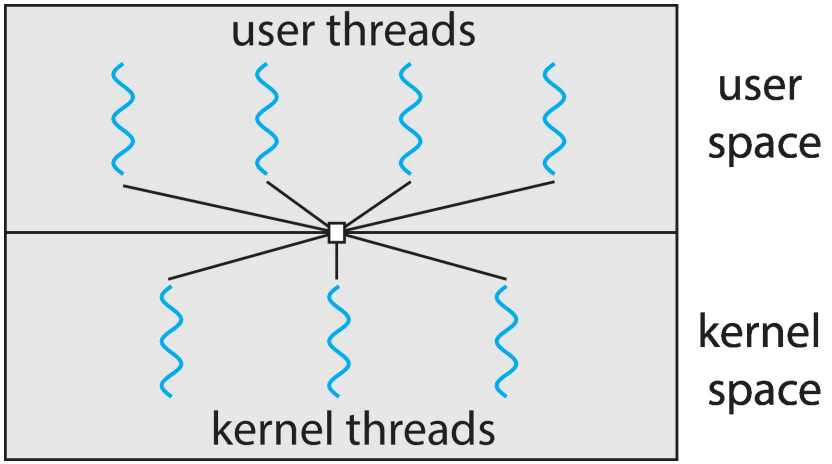
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



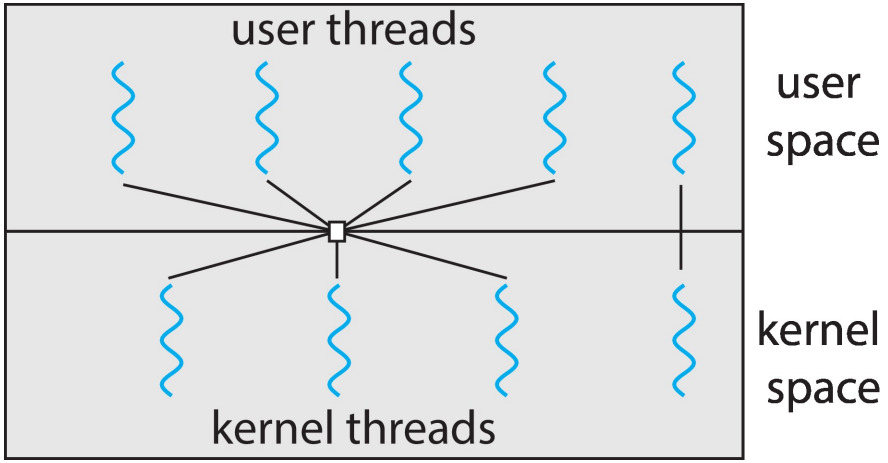
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Implementation

- Two primary ways of implementing
  - Threads can be implemented as part of the OS
    - Kernel-level library supported by the OS
  - Threads can be implemented in user space
    - user-space libraries

Pthreads

- May be provided either as user-level or kernel-level
- 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 (Linux & Mac OS X)

Pthreads Example

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

Pthreads Example (cont)

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

THREAD PROGRAMMING

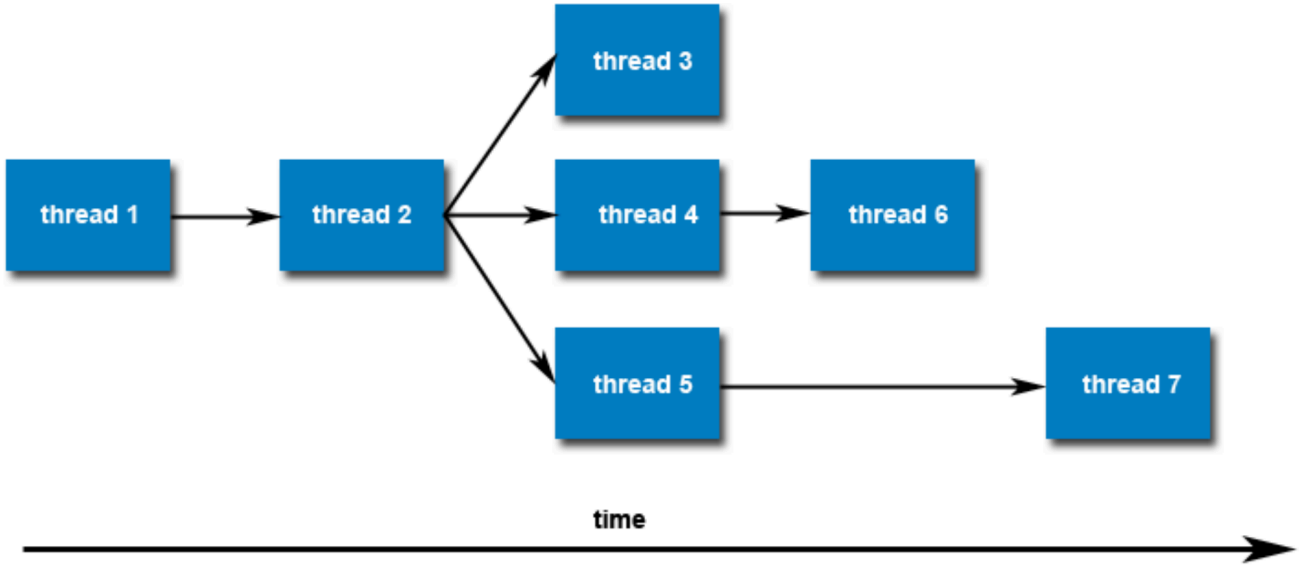
POSIX Threads (Pthreads) API

- Portable Operating System Interface for UNIX
  - POSIX for thread programming, specified by the IEEE POSIX 1003.1c standard
  - C Language
  - Supported by most operating systems: Linux, Mac OS X, Windows (partially), etc.
- The Pthreads API contains over 60 subroutines in three major classes:
  - Thread management: creating, terminating, joining
  - Mutexes: provides exclusive access to code segments/variables with the use of locks (mutual exclusion)
  - Condition variables: provides synchronization and communication between threads that share a mutex

Creating threads

pthread_create (thread_id, attr, function, arg)

- Creates a child thread which starts running the specified function.
- Once created, threads may create other child threads.



Creating threads (cont.)

int pthread_create(pthread_t *thread_id, pthread_attr_t *attr,
void *(*function)(void *), void *arg);

- Function ***pthread_create()*** returns “0” when successful or error integer number otherwise
- Arguments:
  - ***pthread_t *thread*:** address of a threadID
  - ***pthread_attr_t *attr*:** attributes that can be applied to this thread (can be NULL)
  - ***void *(*function)(void *):*** the actual function this thread executes
  - ***void *arg*:** pointer to arguments to be passed to the actual function (can be NULL)

Terminating threads

void pthread_exit(void *status);

- Function ***pthread_exit()*** terminates the thread
- The ***status*** value is returned to the master thread if the master thread is waiting (by calling ***pthread_join()***) for thread termination

- Ways for a thread to terminate:
  - The thread function calls the ***return*** statement
  - The main() process calls ***return*** or ***exit()***
  - The thread function calls ***pthread_exit()***
  - The thread is canceled by another thread via the ***pthread_cancel()*** function

Compling Pthreds:

$ gcc <source_code.c> -o <executable> -lpthread

Simplest thread code

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

void * function(void * arg){
   printf("Hello world, this is child thread!\n");

   //terminate thread
   printf("Child thread is exiting\n");
    pthread_exit(NULL);
}

int main(){
   printf("Hello world, this is master thread!\n");
   pthread_t t_id;

   //create  thread
    int r = pthread_create(&t_id, NULL, function, NULL);
   if (r == 0)
    printf("Child thread created, t_id = %d\n", (int)t_id);
   else
    printf("Thread error nr: %d\n", r);

   printf("Master thread is exiting\n");
return 0;
}
```

OUTPUT, version 1

```
Hello world, this is master thread!
Child thread created, t_id = 174768128
Hello world, this is child thread!
Child thread is exiting
Master thread is exiting
```

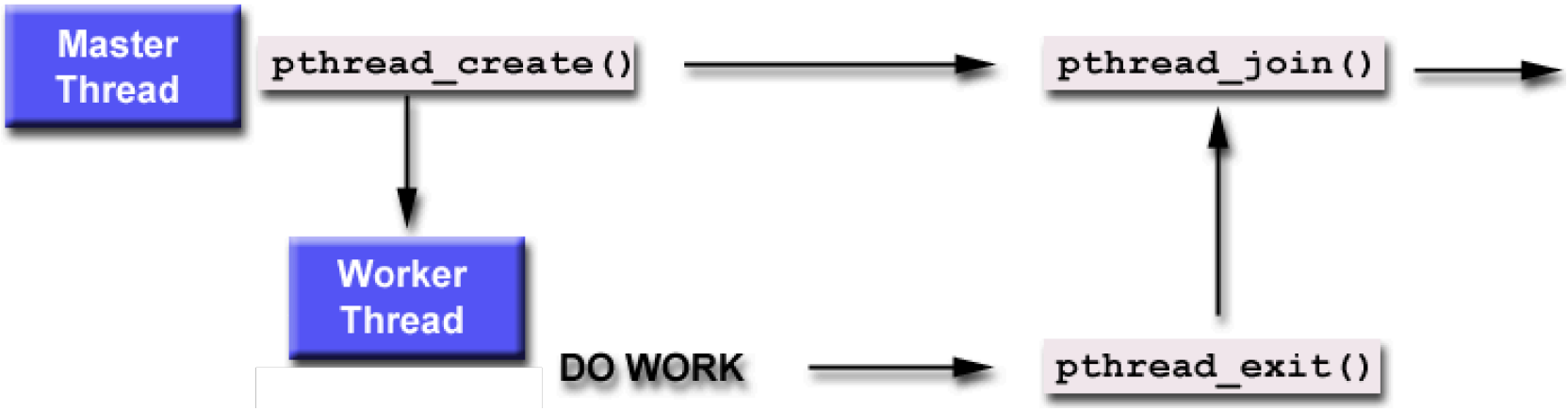
OUTPUT, version 2

```
Hello world, this is master thread!
Child thread created, t_id = 73080832
Master thread is exiting
```

Blocking the master thread until the child thread terminates.

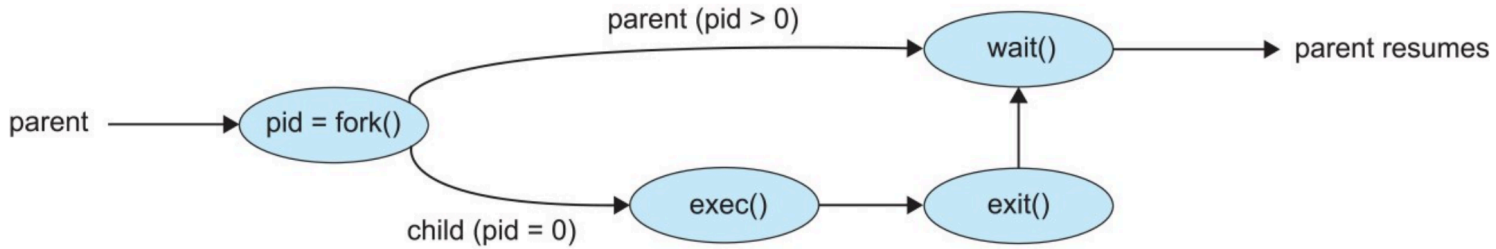
int pthread_join(pthread_t thread, void **status);

- blocks the calling (master) thread
- the calling thread waits for successful termination of the thread specified as the first argument
- an optional *status value is received from the terminating thread's call

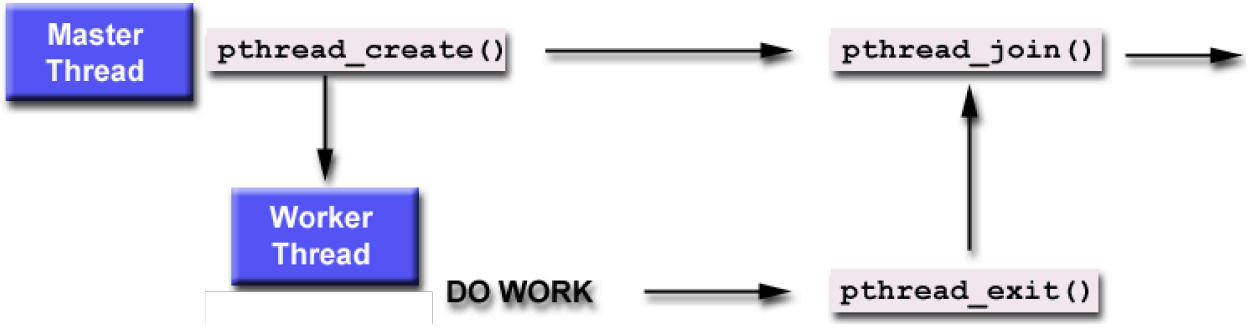


wait() system call & *pthread_join()* function

- The parent process may wait for termination of a child process by using the ***wait()*** system call.



- The parent (master) thread may wait for termination of a child thread by using the ***pthread_join()*** function.



Simplest thread code with *pthread_join()*

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


void * function(void * arg){
   printf("Hello world, this is child thread!\n");

   //terminate thread
   printf("Child thread is exiting\n");
   pthread_exit(NULL);
}

int main(){
   printf("Hello world, this is master thread!\n");
   pthread_t t_id;

   //create  thread
    int r = pthread_create(&t_id, NULL, function, NULL);
   if (r == 0)
      printf("Child thread created, t_id = %d\n", (int)t_id);
   else
      printf("Thread error nr: %d\n", r);

    pthread_join(t_id, NULL);
   printf("Master thread is exiting\n");
return 0;
}
```

OUTPUT

```
Hello world, this is master thread!
Child thread created, t_id = 149229568
Hello world, this is child thread!
Child thread is exiting
Master thread is exiting
```

Multi-threaded process (2 threads)

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


void * f1(void * arg){
   for (int i=0; i<1000; i++){
      printf("1");
      fflush(stdout);

    }
   pthread_exit(NULL);
}

void * f2(void * arg){
   for (int i=0; i<1000; i++){
      printf("2");
      fflush(stdout);

    }
   pthread_exit(NULL);
}

int main(){
   pthread_t t_id1, t_id2;
   int r1 = pthread_create(&t_id1, NULL, f1, NULL);
   int r2 = pthread_create(&t_id2, NULL, f2, NULL);
   pthread_join(t_id1, NULL);
   pthread_join(t_id2, NULL);
return 0;
}
```

OUTPUT

```
22221111222222222222222222222222222222
22222222222222222222212111111111111111
11111111121111111211112222221111112222222
22222222222222222222111111111111111111111
22222222222222222222222221122211112222222
21111111111111112222222222221212222222222
22222111121111111222222222222222222222222
22222111111111111111111111111111111111111
12211111111112222222222222222221222222222
22222222222222211111111111111111112222222
12221222211222222222222222222222211111111
11111111111211111211111222222222222222211
11111111111111111111222222222222222222222
22222222222222222222222222221111111122222
11112222111111111111111112222222222222221
11222222222222222111111111121111111111111
```

Multi-threaded process (2 threads)

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

void * f1(void * arg){
   for (int i=0; i<10; i++){
      printf("1");
      fflush(stdout);
      sleep(1)
    }
   pthread_exit(NULL);
}

void * f2(void * arg){
   for (int i=0; i<10; i++){
      printf("2");
      fflush(stdout);
      sleep(1);
    }
   pthread_exit(NULL);
}

int main(){
   pthread_t t_id1, t_id2;
   int r1 = pthread_create(&t_id1, NULL, f1, NULL);
   int r2 = pthread_create(&t_id2, NULL, f2, NULL);
   pthread_join(t_id1, NULL);
   pthread_join(t_id2, NULL);
return 0;
}
```

OUTPUT

```
$ time ./thread4
12212121212112121221
real 0m10.034s
user 0m0.002s
sys 0m0.005s
```

Single-threaded process (2 threads)

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

void f1(){
   for (int i=0; i<10; i++){
      printf("1");
      fflush(stdout);
      sleep(1);
   }
}

void f2(){
   for (int i=0; i<10; i++){
      printf("2");
      fflush(stdout);
      sleep(1);
   }
}

int main(){
   f1();
   f2();
return 0;
}
```

OUTPUT

```
$ time ./thread5
1111111112222222222
real 0m20.053s
user 0m0.002s
sys 0m0.004s
```

Passing arguments to the thread function

- Passing an integer

```
int myInteger = 10;
int * ptr = &myInteger;
pthread_create(&tid, NULL, fn, (void *) ptr);
```

- Reading an integer in the thread

```
void *fn (void *ptr){
   int * x = (int *)ptr;
   printf("I received the value: %d", *x);
   (…)
```

- Passing a string

```
char message[25] = {"hello mr thread"};

pthread_create(&t_id, NULL, fn, message);
```

- Reading string in the thread

```
void *function (void *ptr){

   printf("I received a message: %s", ptr);
   (…)
```

Thread code with integer passing

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

void * function(void * ptr){
   printf("Hello world, this is child thread!\n");
   int *x = (int*) ptr;
   printf("I received an integer: %d\n", *x);
   pthread_exit(NULL);
}

int main(){
   printf("Hello world, this is master thread!\n");
   pthread_t t_id;

   //a variable x stores the integer value
   int x = 10;

   // a pointer to the address of x will be passed
   int * ptr = &x;

   int r = pthread_create(&t_id, NULL, function, (void *)ptr);
   pthread_join(t_id, NULL);
   printf("Master thread is exiting\n");
return 0;
}
```

OUTPUT

```
Hello world, this is master thread!
Hello world, this is child thread!
I received an integer: 10
Master thread is exiting
```

Thread code with string passing

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

void * function(void * ptr){
   printf("Hello world, this is child thread!\n");

   printf("I got a message: %s\n", ptr);
   pthread_exit(NULL);
}

int main(){
   printf("Hello world, this is master thread!\n");
   pthread_t t_id;

   //a string "hello mr thread" will be sent
   char message[25] = {"hello mr thread"};

   //create  thread
   int r = pthread_create(&t_id, NULL, function, message);

   pthread_join(t_id, NULL);
   printf("Master thread is exiting\n");
return 0;
}
```

OUTPUT

```
Hello world, this is master thread!
Hello world, this is child thread!
I got a message: hello mr thread
Master thread is exiting
```

Multi-threaded process with integer passing

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

#define NUM_THREADS 5

void *perform_work(void *arguments){
  int index = *((int *)arguments);
  int sleep_time = 1 + rand() % NUM_THREADS;
  printf("THREAD %d: Started.\n", index);
  printf("THREAD %d: Will be sleeping for %d seconds.\n", index, sleep_time);
  sleep(sleep_time);
  printf("THREAD %d: Ended.\n", index);

}
```

Multi-threaded process with integer passing (cont.)

```
int main(void) {
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS], i, result_code;

  //create all threads one by one
  for (i = 0; i < NUM_THREADS; i++) {
    printf("IN MAIN: Creating thread %d.\n", i);
    threads[i] = i;
    thread_args[i] = i;
    result_code = pthread_create(&threads[i], NULL, perform_work, &thread_args[i]);
    assert(!result_code);
  }
  printf("IN MAIN: All threads are created.\n");

  //wait for each thread to complete
  for (i = 0; i < NUM_THREADS; i++) {
    result_code = pthread_join(threads[i], NULL);
    assert(!result_code);
    printf("IN MAIN: Thread %d has ended.\n", i);
  }
  printf("MAIN program has ended.\n");
  return 0;
}
```

Multi-threaded process with integer passing (output)

```
IN MAIN: Creating thread 0.
IN MAIN: Creating thread 1.
THREAD 0: Started.
THREAD 1: Started.
IN MAIN: Creating thread 2.
THREAD 1: Will be sleeping for 5 seconds.
THREAD 0: Will be sleeping for 3 seconds.
THREAD 2: Started.
THREAD 2: Will be sleeping for 4 seconds.
IN MAIN: Creating thread 3.
IN MAIN: Creating thread 4.
THREAD 3: Started.
THREAD 3: Will be sleeping for 4 seconds.
THREAD 4: Started.
THREAD 4: Will be sleeping for 1 seconds.
IN MAIN: All threads are created.
THREAD 4: Ended.
THREAD 0: Ended.
IN MAIN: Thread 0 has ended.
THREAD 2: Ended.
THREAD 3: Ended.
THREAD 1: Ended.
IN MAIN: Thread 1 has ended.
IN MAIN: Thread 2 has ended.
IN MAIN: Thread 3 has ended.
IN MAIN: Thread 4 has ended.
MAIN program has ended.
```

Unrestricted access to global variables!

```
#include (…)

long balance = 0; //global variable

void * inc(void * arg){
   for (long i=0; i<100000000; i++)
      balance++;
   pthread_exit(NULL);
}

void * dec(void * arg){
   for (long i=0; i<100000000; i++)
      balance--;
   pthread_exit(NULL);
}
int main(){
   pthread_t t_id1, t_id2;
   pthread_create(&t_id1, NULL, inc, NULL);
   pthread_create(&t_id2, NULL, dec, NULL);
   pthread_join(t_id1, NULL);
   pthread_join(t_id2, NULL);
   printf("Value of balance is: %ld\n", balance);
return 0;
}
```

UNDESIRED OUTPUTS:

```
Value of balance is: -986105

Value of balance is: -771273

Value of balance is: 2822774

Value of balance is: 2791508
```

Tools to restrict access to global variables

- Mutexes: provides exclusive access to code segments/variables with the use of locks (mutual exclusion)

- Condition variables: provides synchronization and communication between threads that share a mutex

Thank you.