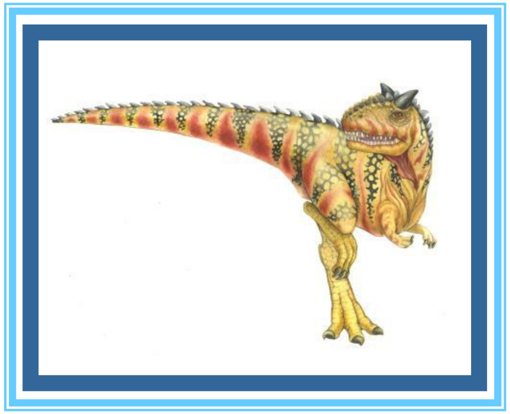
7: Threads & Concurrency



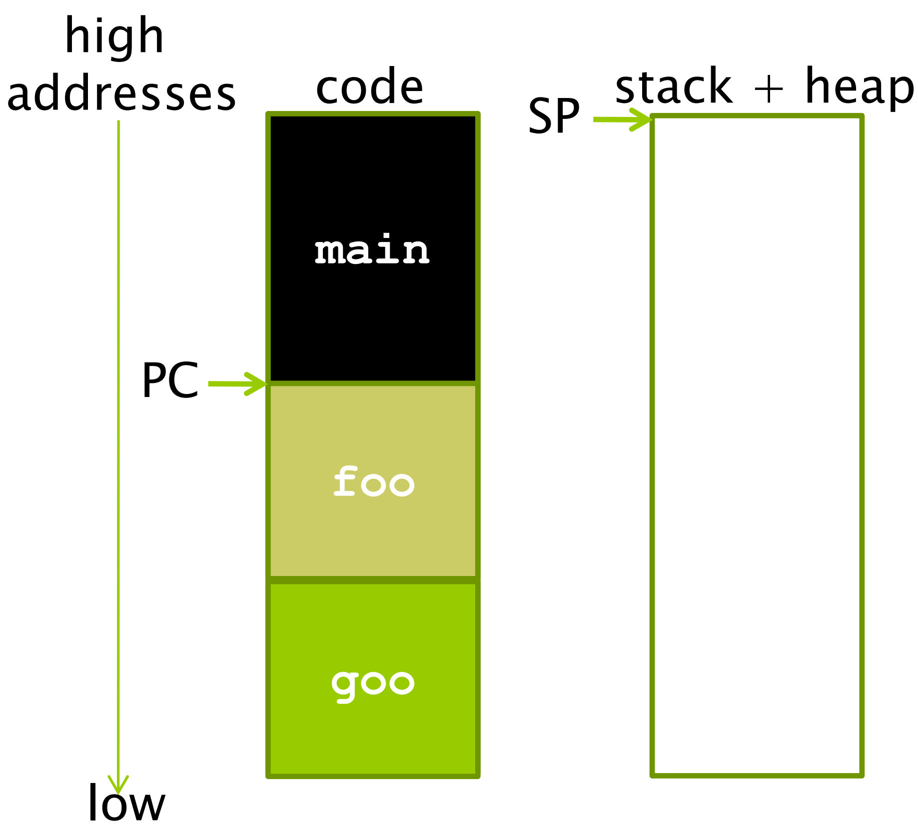
7: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Example: Windows Threads

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Illustrate implicit threading such as OpenMP
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs

Stack Pointer and Program Counter



Consider a code with the following functions:

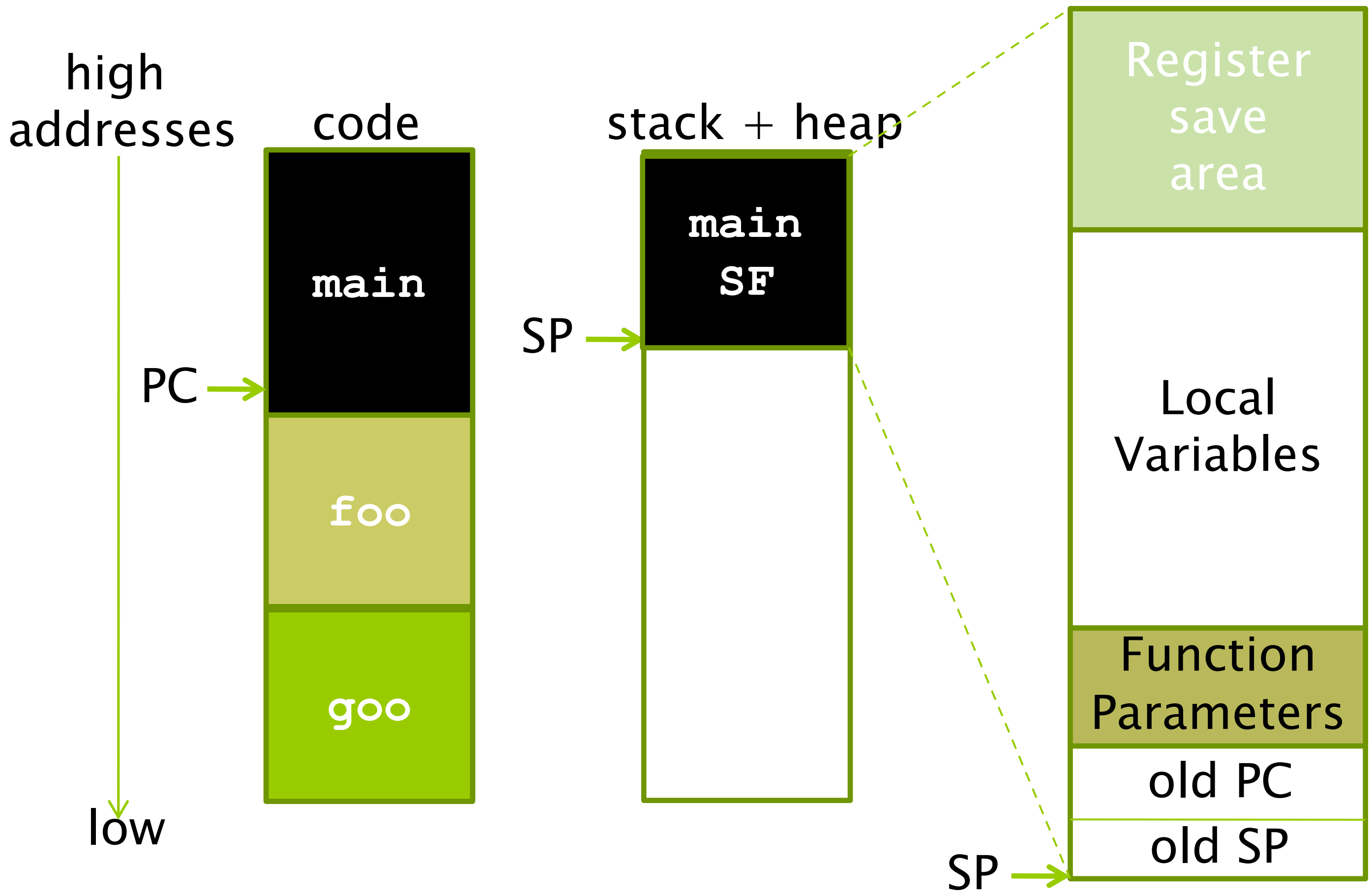
```
int main(int argc,
           char**argv);
char *foo(int, int,
  int);
int goo(double, int);
```

Assume that the functions are called:

`main->foo->goo`

PC pointing to main
Stack is empty

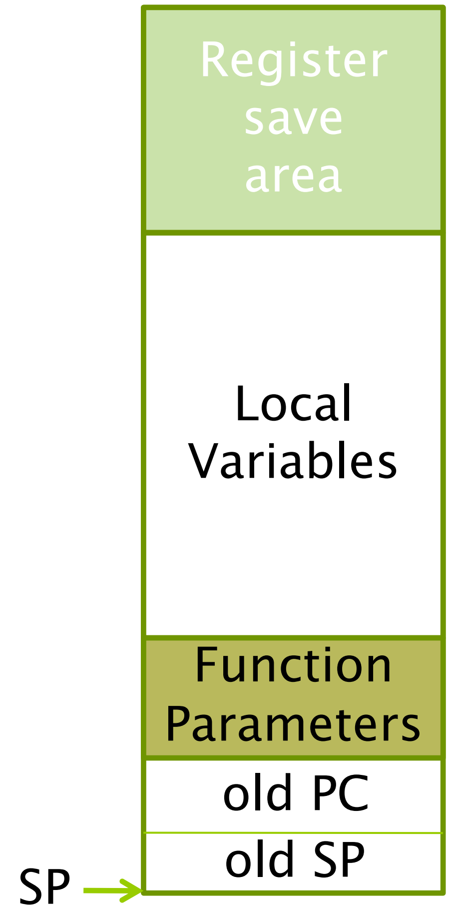
Stack Frame



high
addresses
low
code
main
foo
goo
PC
stack + heap
main
SF
SP
Register
save
area
Local
Variables
Function
Parameters
old PC
old SP
SP

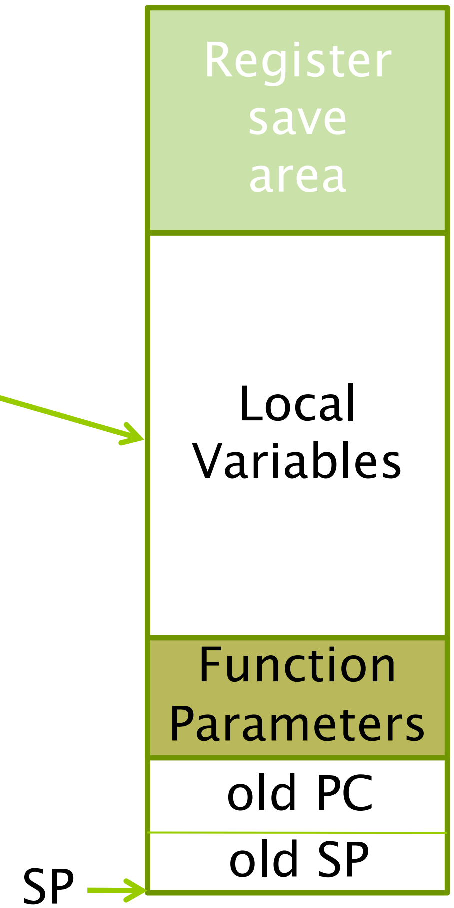
Stack Frame Organization - 1

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```



Stack Frame Organization - II

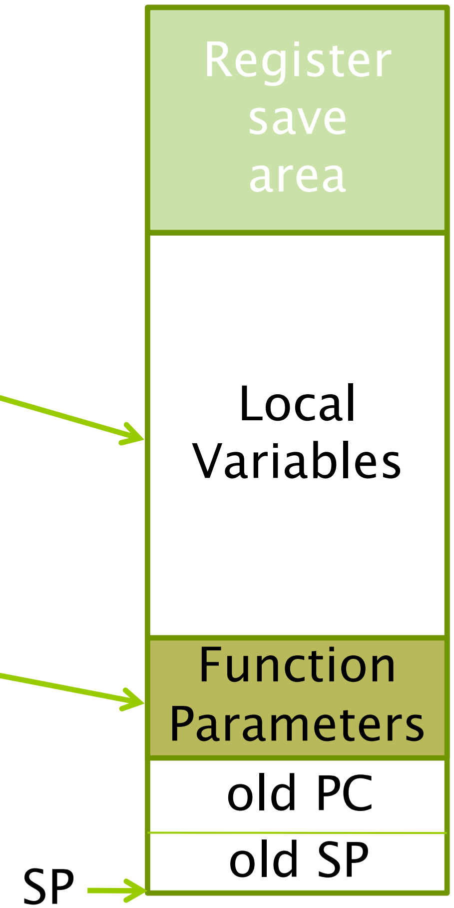
```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```



a and d could be in registers

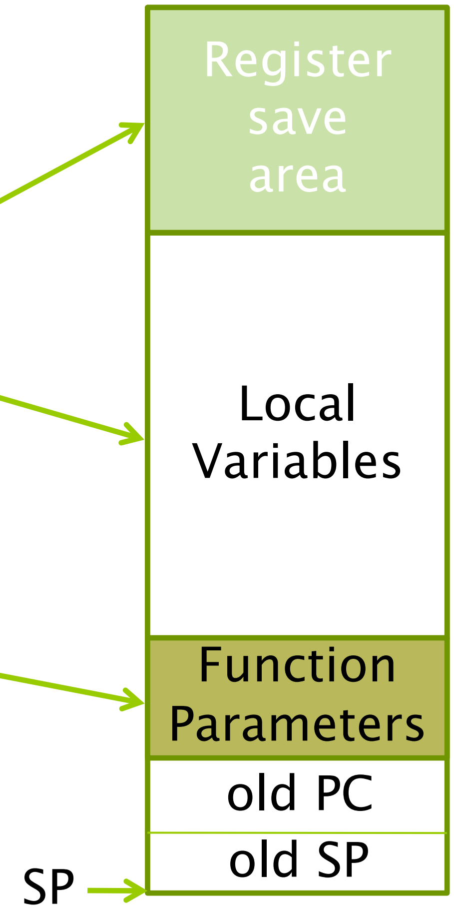
Stack Frame Organization - III

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```

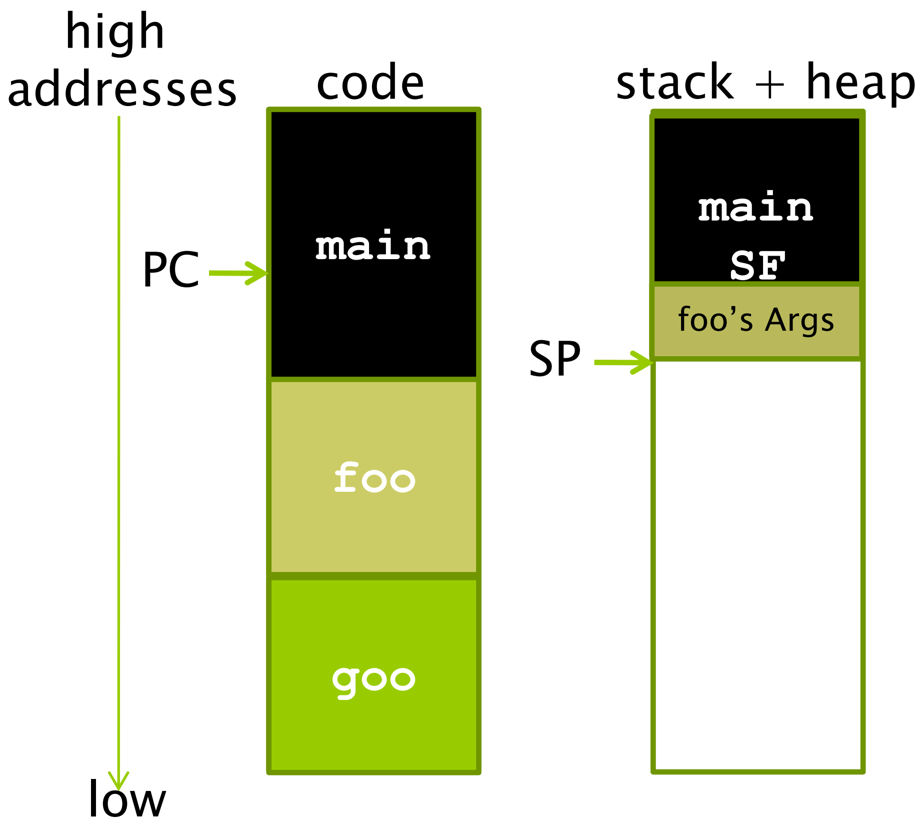


Stack Frame Organization - IV

```
char *foo(int x, int y , int z)
{
    int a;
    char array[500];
    double d;
    …

    a = x+y+goo(d,z);
    …
}
```

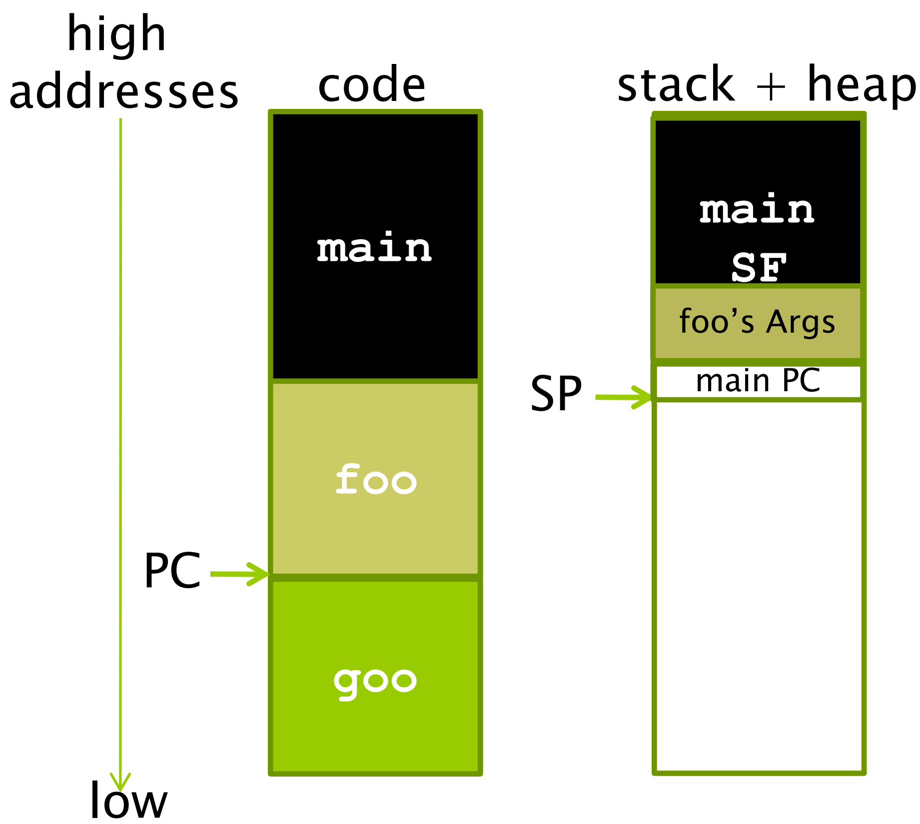


Function Call and SF Creation - I



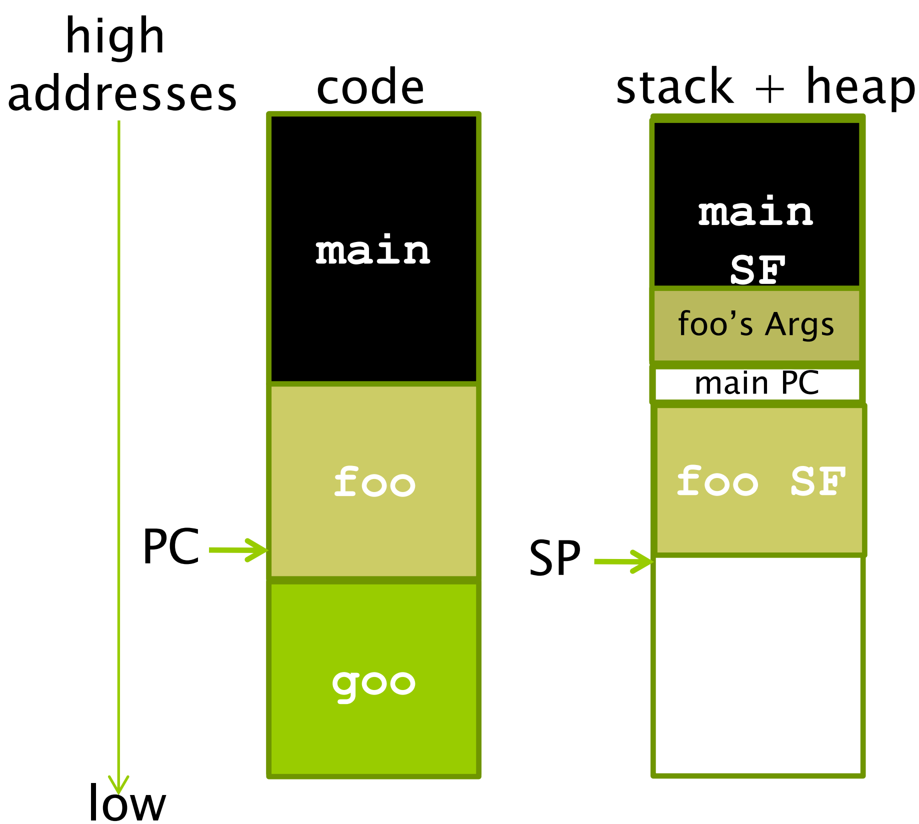
1. Push foo's Args

Function Call and SF Creation - II



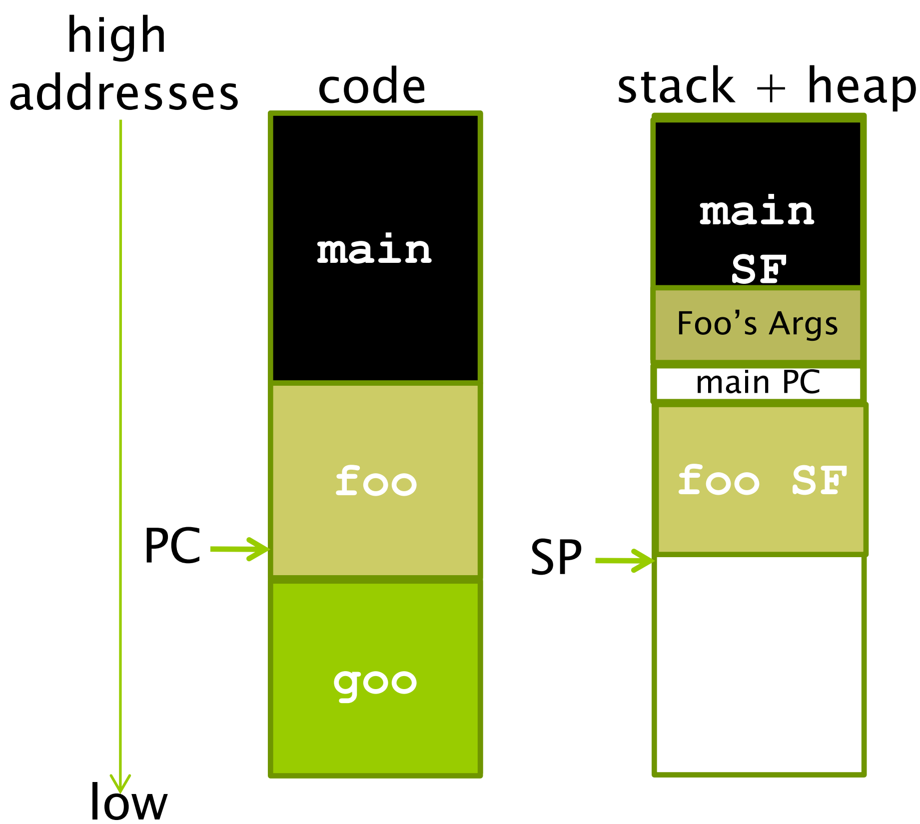
1. Push foo's Args
2. Call `foo`

Function Call and SF Creation - III



1. Push foo's Args
2. Call `foo`
3. Save `main` SP and decrement SP

Function Call and SF Creation - IV



1. Push foo’s Args
2. Call `foo`
3. Save `main` SP and decrement SP

Done in software
i.e., done by instructions generated by the compiler!

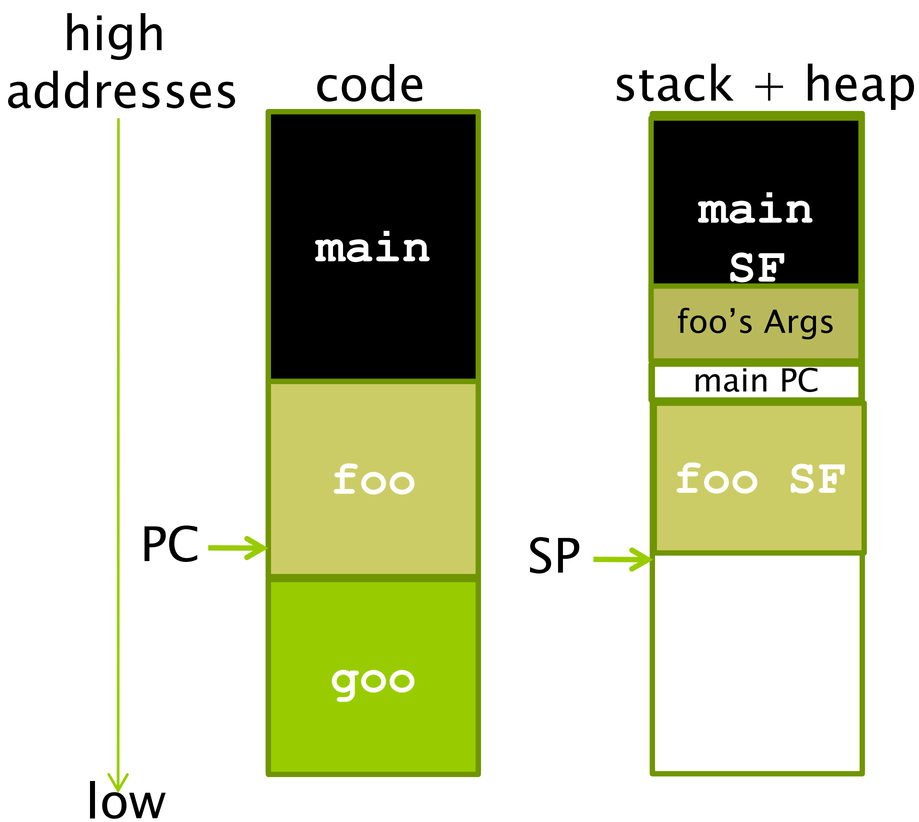
Q & A

- Does each function use the same stack frame size?
  - No. Depends on the size of local variables
- How and when is the size of stack frame determined?
  - Compiler determines by looking at the code. Compile-time.
- How is the stack frame allocated during run-time?
  - Decrementing the stack pointer

Q & A

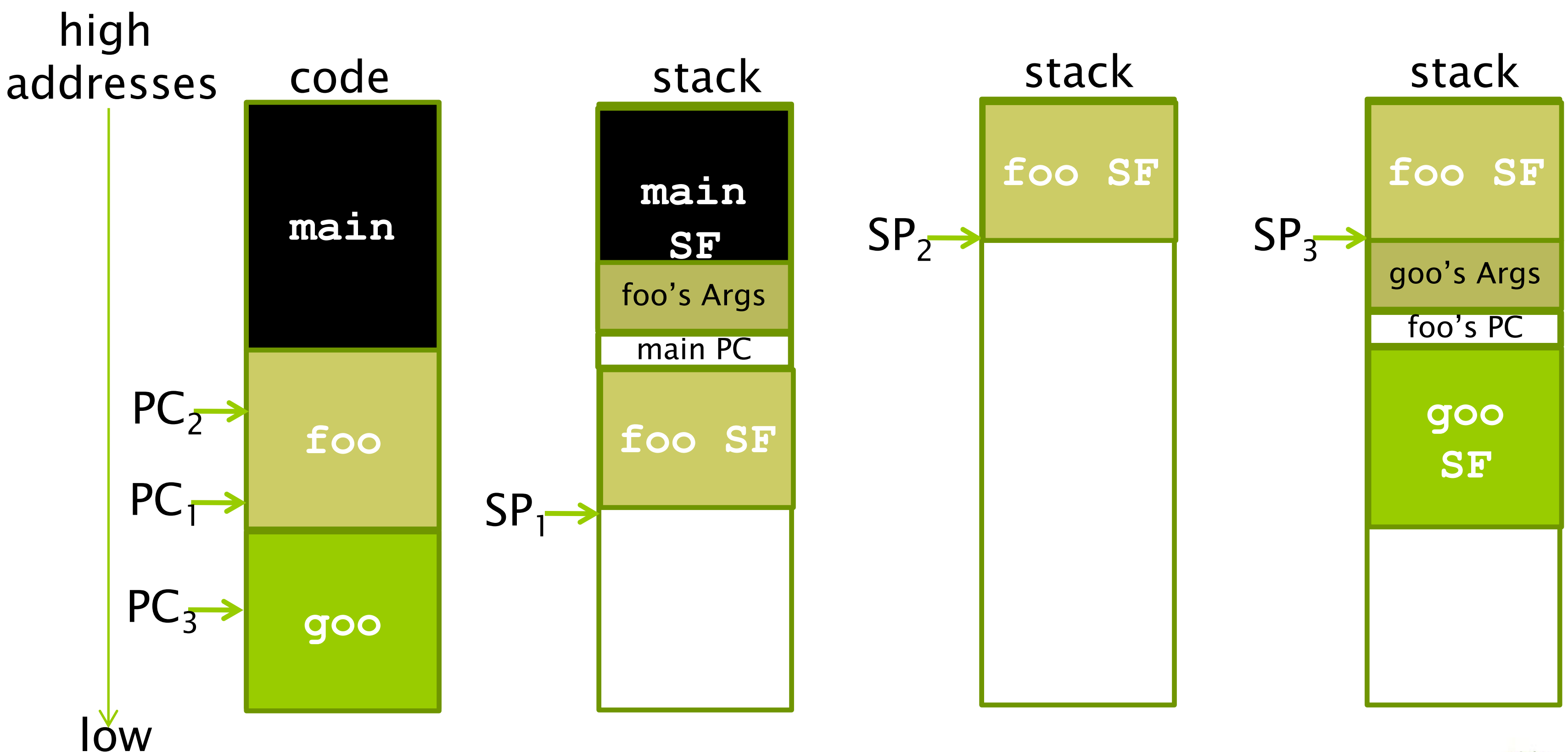
- What is the stack pointer?
  - A value stored in stack pointer register (%esp) pointing to the beginning of the stack frame
- What is a program counter?
  - A value stored in program counter register (%eip) pointing to a point in the text

Single-thread process



- Thread of control
  - PC
  - SP
  - Stack

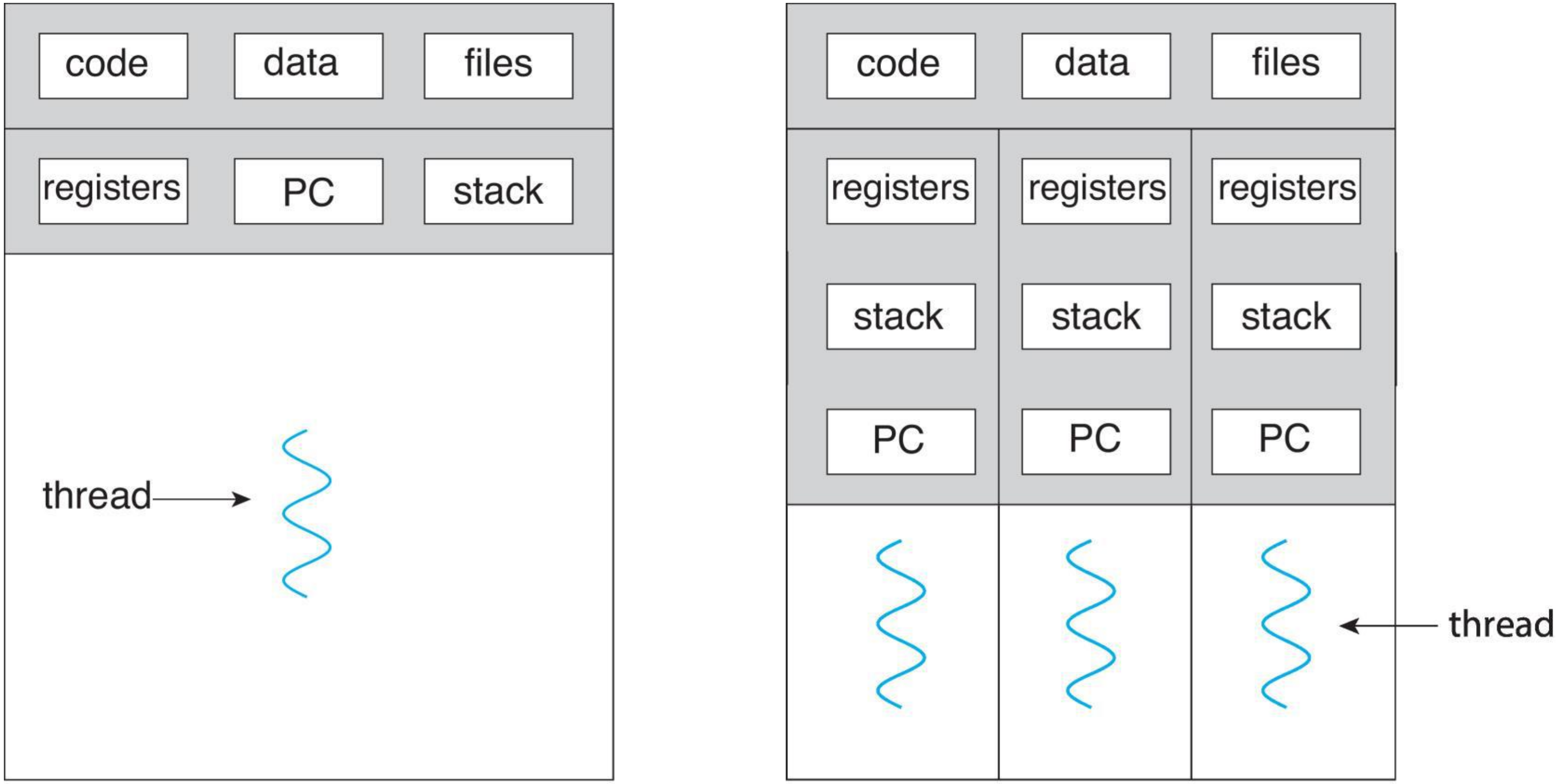
Multi-thread process



Motivation

- Multiple tasks with the application can be implemented by separate threads (**Scalability, Responsiveness)**
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight (**economy**)
- Threads run within the application/process (**Resource Sharing**)
- Can **simplify code**, increase **efficiency**
- Most modern applications are **multithreaded**
- **Kernels** are generally **multithreaded**

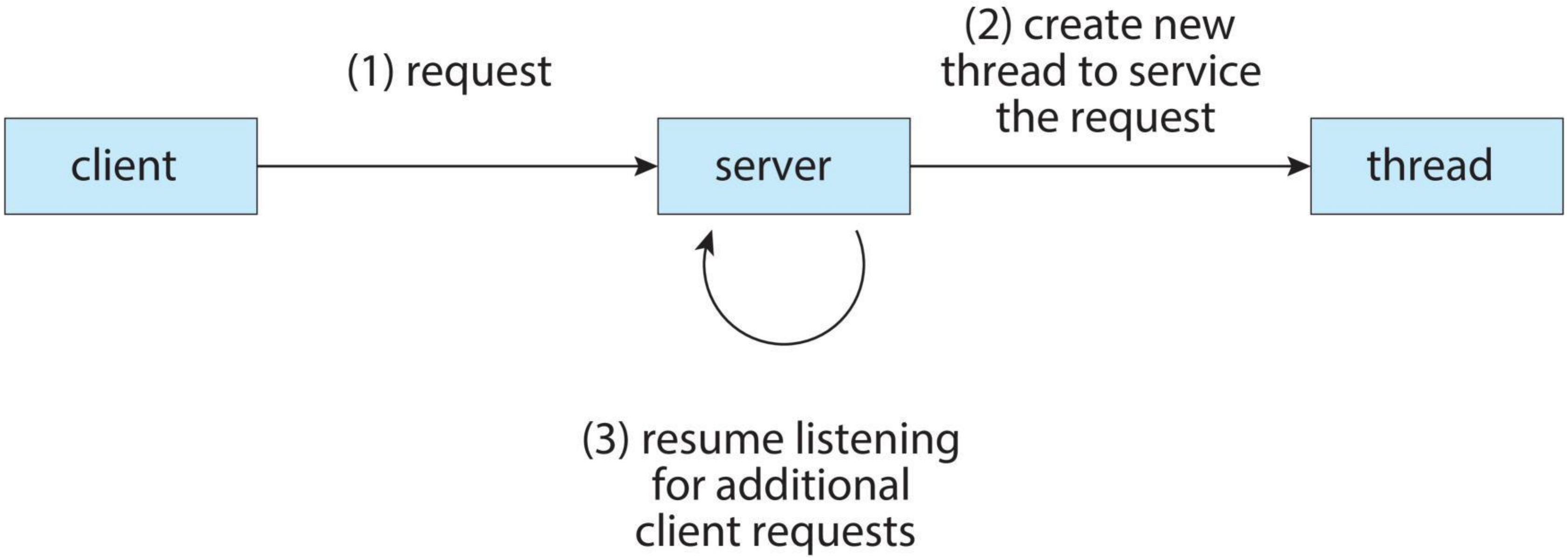
Single and Multithreaded Processes



single-threaded process

multithreaded process

Multithreaded Server Architecture



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing – threads share resources** of process, **easier** than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multicore architectures, with one or two threads per core
- But:
  - More difficult to program with threads (a single process can now do multiple things at the same time).
  - New categories of bug are possible (**synchronization** is then required between threads: Chapter 6).

What is a thread

- A thread is the smallest sequence of programmed instructions within a process that can be managed independently by an **OS scheduler**.
- The implementation of threads and processes differs between operating systems, but in most cases **a thread is a component of a process**.
- Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.
- The threads of a process share its executable **code** and the values of its **variables** (code section, data section, OS resources) at any given time.

Threads vs. processes 1/2

- **Threads differ from processes in that:**

processes	threads
typically independent	subsets of a process
more state information	share process state and resources
Separate address spaces	Same address space
interact through **ipc models**: (shared memory/message passing)	variables
Slower context switching	Faster context switching

Threads vs. processes 2/2

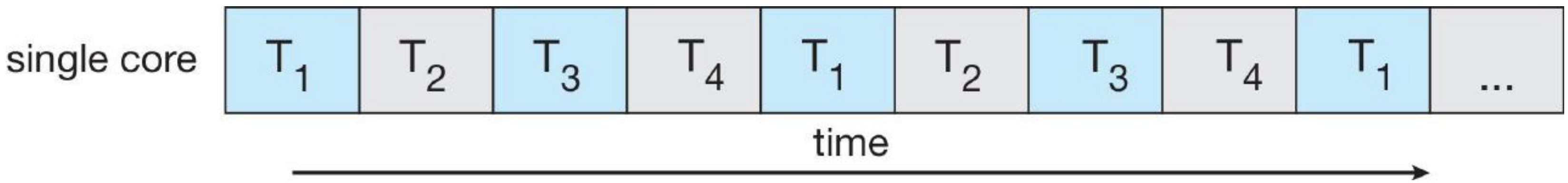
- **Similarities (like process)**
  - Threads share CPU and only one thread active (running) at a time.
  - Threads within a processes execute sequentially.
  - Thread can create children.
  - If one thread is blocked, another thread can run.
- **Differences (unlike process)**
  - Threads are not independent of one another.
  - All threads can access every address in the process
  - Threads are designed to assist one another. (Processes might or might not assist one another.)

Multicore Programming

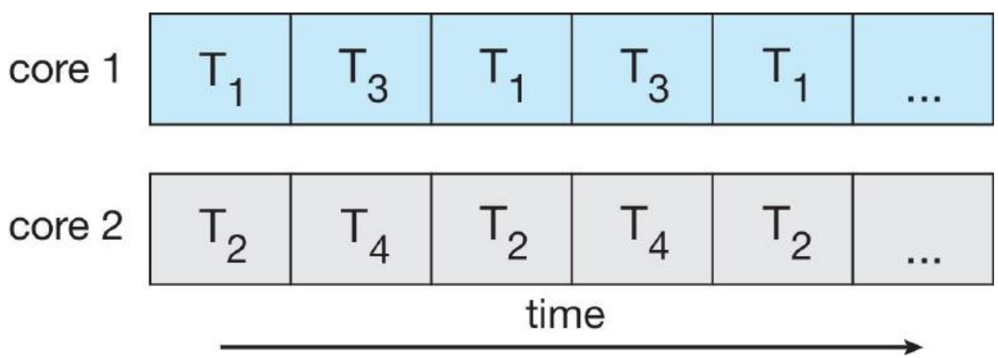
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Load Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task **simultaneously**
  - Multiple processors / cores are required
- ***Concurrency*** supports more than one task **making progress**
  - Single processor / core, CPU scheduler providing concurrency by doing context switches
- Parallelism implies concurrency, but concurrency does not imply parallelism.

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



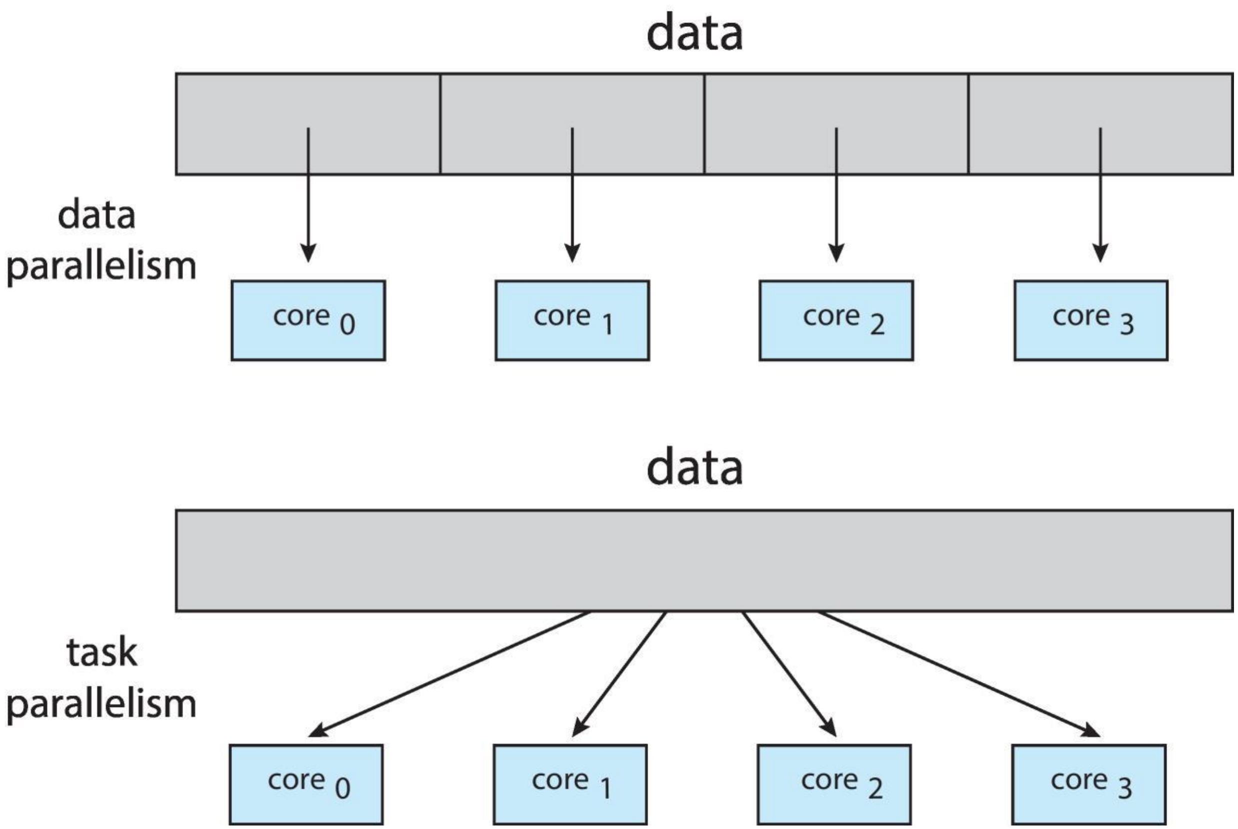
Concurrency is a property of a program where two or more tasks can be in progress simultaneously.

Parallelism is a run-time property where two or more tasks are being executed simultaneously.

Multicore Programming

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, **same operation** on each
    - Example: when doing image processing, two cores can each process half of the image
  - **Task parallelism** – distributing threads across cores, each thread performing **unique operation**
    - Example: when doing sound processing, the sound data can move through each core in sequence, with each core doing a different kind of sound processing (filtering, echo, etc.)

Data and Task Parallelism



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

https://en.wikipedia.org/wiki/Amdahl%27s_law

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Proof:

S is serial portion, P is parallel portion of program.

S and P are portions: percentages of total running time for the program on one core. So S + P = 100% = 1

Running time on one core: $R_1 = S + P = 1$

Running time on N cores: $R_N \geqslant S + P/N = S + (1 - S)/N$

$\geqslant$, not =, because of extra communication required between threads.

Speedup $= R_1 / R_N \leqslant 1 / (S + (1 - S)/N)$

Amdahl's Law

- Example: if the application is 75% parallel and 25% serial, moving from 1 to 2 cores:

  $$0.25 + 0.75/2 = 0.625$$

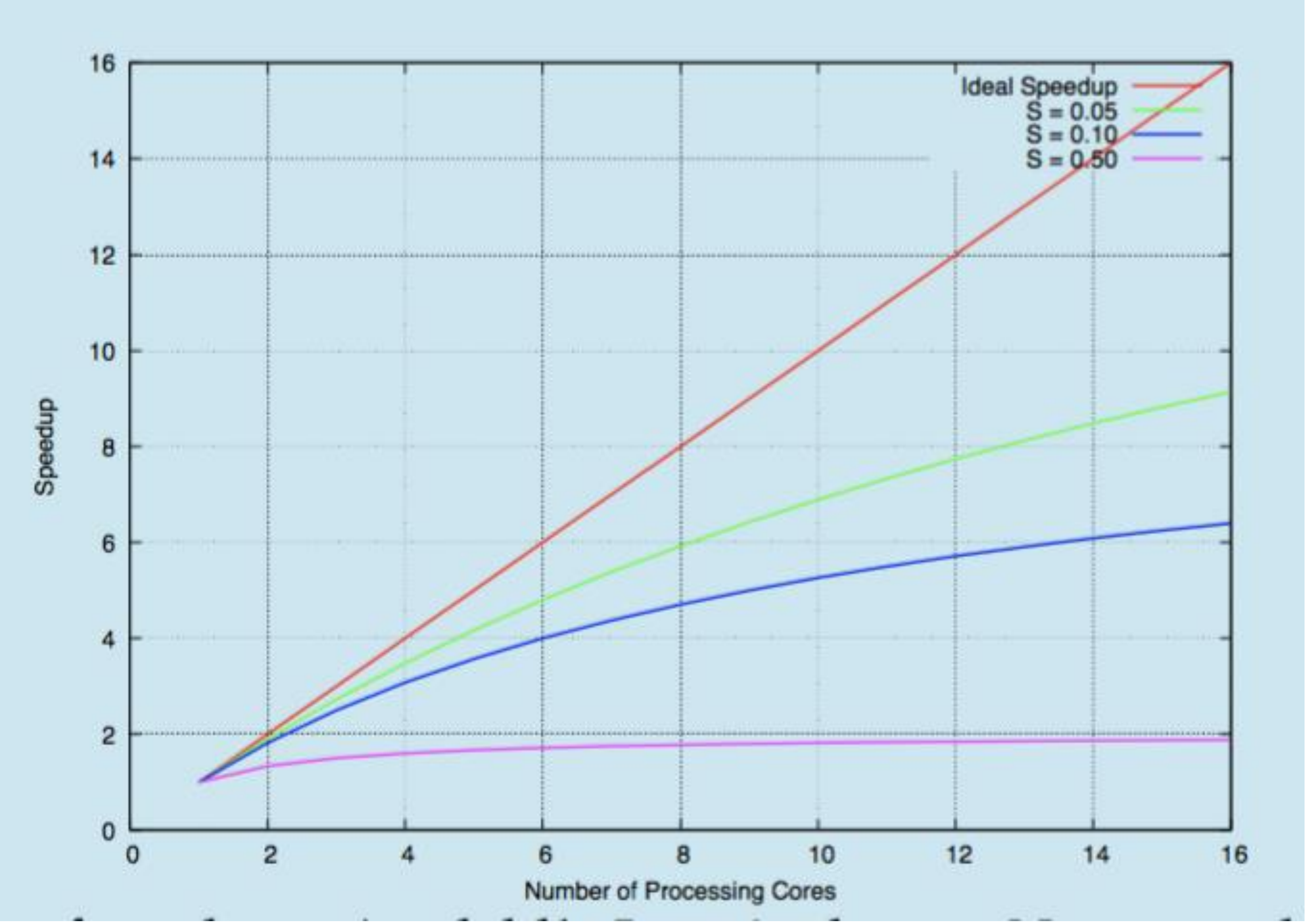
  results in a maximum speedup of 1/0.625 = 1.6 times.

- As *N* approaches infinity, the maximum speedup approaches 1 / *S*

  Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

- But does the law take into account contemporary multicore systems?

Amdahl's Law



Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

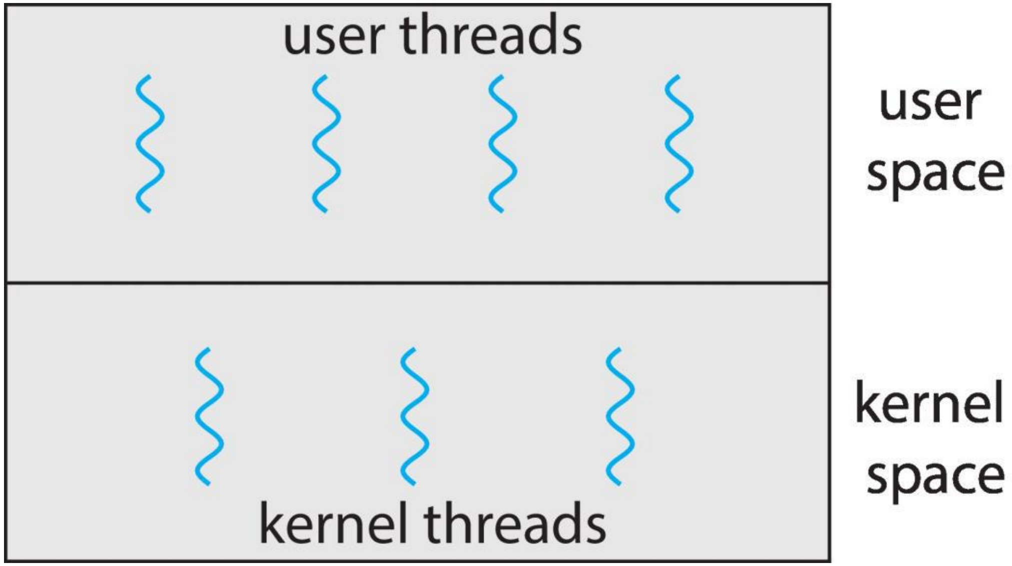
User Threads and Kernel Threads

- **User threads** - **management (thread creation, thread scheduling, etc.) done by user-level threads library.**
  - Advantages:
    - No need for OS support so works even on very old or very simple OS that does not have system calls for thread management.
    - No system call required so thread management is fast: only need a library function call.
  - Disadvantage:
    - When the kernel schedules processes, a process with only one thread gets as much CPU time as a process with many threads.
    - All the thread scheduling inside a process must be done at user level (not done by kernel) so each thread must be nice and cooperate with the other threads in the process and regularly give CPU time to the other threads. This makes the program more complicated to write.

User Threads and Kernel Threads

- **Kernel threads** - **supported by the kernel.**
  - Advantages:
    - Kernel knows how many threads each process contains so it can give more CPU time to the processes with many threads.
    - No need for threads to cooperate for scheduling (thread scheduling done automatically by kernel) so user program simpler to write.
  - Disadvantages:
    - Every thread management operation requires a system call so slower compared to user-level threads.
    - Kernel's PCB data structures more complex because the kernel needs to keep track of both **processes and threads** inside processes.
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads

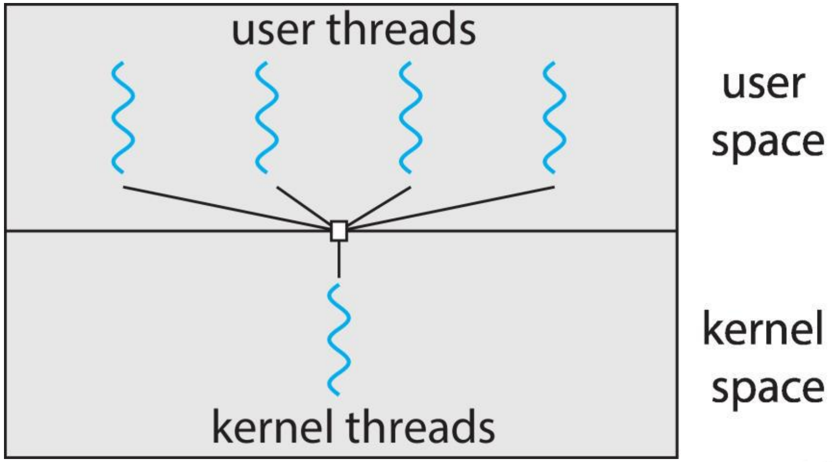


Multithreading Models

- If threads are available both at user level and kernel level, then some user threads are normally associated with some kernel threads.
- Several models of association between user threads and kernel threads are possible:
  - Many-to-One
  - One-to-One
  - Many-to-Many

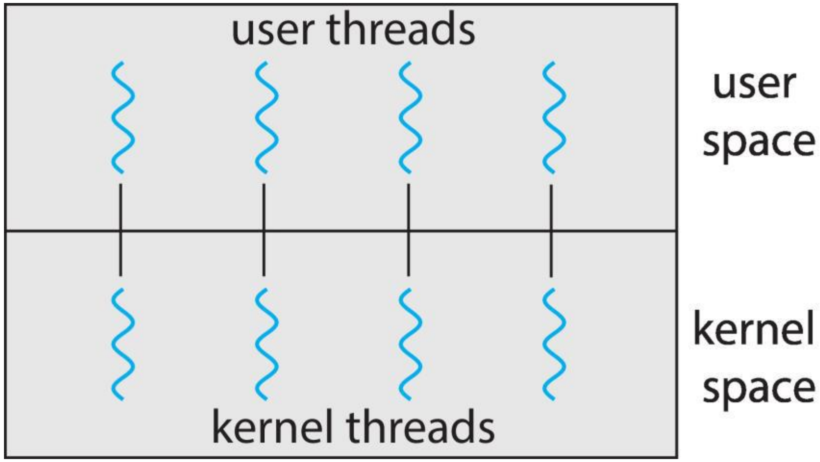
Many-to-One

- Many user-level threads mapped to single kernel thread.
- One thread blocking (waiting for something) causes all threads to block (because their common kernel thread is blocked).
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time.
- Few systems currently use this model.
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



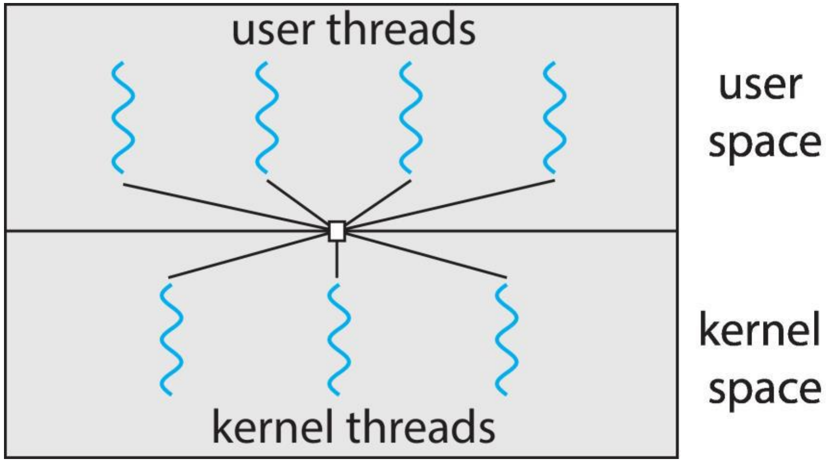
One-to-One

- 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.
- Example: Windows with the ***ThreadFiber* package**.
- Otherwise not very common.



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported by kernel, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on kernel threads (themselves scheduled by kernel)
  - Known as **process-contention scope** (**PCS**) since scheduling competition is between user-level threads within the same process
  - Typically done via priority set by programmer
- **Kernel thread** scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all kernel-level threads within all processes in the system

Process Contention Scope

- **Process Contention Scope** is one of the two basic ways of scheduling threads.
  - Process local scheduling (known as Process Contention Scope)
  - System global scheduling (known as System Contention Scope).
- PCS scheduling means that all of the scheduling mechanism for the thread is local to the process—the thread's library has full control over which thread will be scheduled on an LWP. This also implies the use of either the Many- to-One or Many-to-Many model.
- PCS: done by the threads library. The library chooses which thread will be put on which LWP.
- **SCS:** used by the kernel to decide which kernel-level thread to schedule onto a CPU, wherein all threads (as opposed to only user-level threads, as in the PCS) in the system compete for the CPU. This also implies the use of **One- to-One model**.

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_**PROCESS** schedules threads using **PCS scheduling**
  - PTHREAD_SCOPE_**SYSTEM** schedules threads using **SCS scheduling**
- Can be limited by OS – **Linux and Mac OS X** only allow PTHREAD_SCOPE_**SYSTEM**

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely **in user space** (user threads only)
  - **OS-level library** supported by the **kernel** (user threads mapped to kernel threads, with one-to-one model for example).

- Three primary thread libraries:
  1. POSIX **Pthreads**
  2. Windows threads
  3. Java threads

Pthreads

- May be provided either as **user-level or kernel-level**
- **A POSIX standard** (IEEE 1003.1c) **API** for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is **up to developers of the library**
- Common in UNIX operating systems (Linux & Mac OS X)

Pthreads Example (cont)

thrd-posix.c

gcc –o thrd-posix thrd-posix.c –lpthread
./thrd-posix

```
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[]) {
      pthread_t tid; /* the thread identifier */
      pthread_attr_t attr; /* set of attributes for the thread */
      int n;
      if (argc != 2) {
            fprintf(stderr,"usage: a.out <integer value>\n");
            return -1;
      }
      n = atoi(argv[1]);
      if (n < 0) {
            fprintf(stderr,"Argument %d must be non-negative\n",n);
            return -1;
      }
      /* get the default attributes */
      pthread_attr_init(&attr);
      /* create the thread */
      pthread_create(&tid,&attr,runner,argv[1]);
      /* now wait for the thread to exit */
      pthread_join(tid,NULL);
      printf("sum = %d\n",sum);
}
```

Pthreads Example

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

    if (upper > 0) {
    for (i = 1; i <= upper; i++)
        sum += i;
    }

    pthread_exit(0);
}
```

Pthreads Code for Joining 4 Threads

thrd-demo.c

gcc –o thrd-demo thrd-demo.c –lpthread
./thrd-demo

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4
void* threadfunc(void *r) {
        printf("This is a pthread %d.\n", *(int*)r);
        pthread_exit(0);
}

int main(void) {
    pthread_t workers[NUM_THREADS];
    int i, ret;
    int data[NUM_THREAD] = {1,2,3,4};

    for (i = 0; i < NUM_THREADS; ++i){
        ret = pthread_create(&workers[i], NULL, threadfunc, (void*)&data[i]);
        if (ret != 0){
                printf("Create pthread %d error!\n“, i);
                return 1;
        }
    }
    printf("This is the main process.\n");

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

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by **compilers** and **run-time libraries** rather than programmers
- Take **OpenMP as** an example **http://www.openmp.org/**
    1. Fork-Join
    2. Grand Central Dispatch[1]
    3. Intel Threading Building Blocks (TBB)[2]
    4. Thread Pools: Create a number of threads **in advance** in a pool where they await work

[1]a technology developed by Apple Inc. to optimize application support for systems with multi-core processors and other **symmetric multiprocessing** systems. It is an implementation of task parallelism based on the **thread pool pattern**.

[2]Threading **Building Blocks** (**TBB**) is a C++ template library developed by Intel for parallel programming on multi-core processors. Using TBB, a computation is broken down into tasks that can run in parallel. The library manages and schedules threads to execute these tasks.

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

#pragma omp parallel

Create as many threads as there are cores

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
      printf("I am a parallel region\n");
  }

  /* sequential code */

  return 0;
}
```

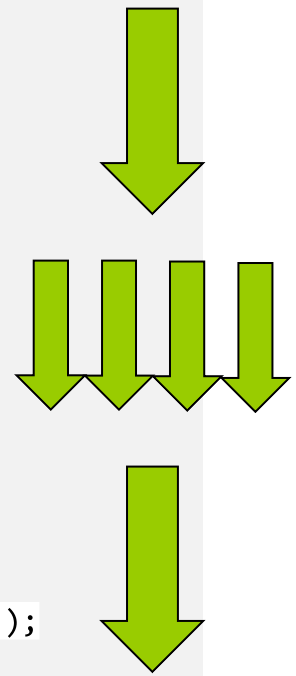
Example: run the for loop in parallel:

```c
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

OpenMP

openMP_demo.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(int argc, char *argv[]) {
      int a[1000], b[1000], c[1000];
      /* sequential code */
      srand((unsigned)time(NULL));
      for (int i = 0; i < 1000; i++) {
            a[i] = rand() * 100 /RAND_MAX;
            b[i] = rand() * 100 / RAND_MAX;
      }
      /* parallel code */
      #pragma omp parallel for
      for (int i = 0; i < 1000; i++) {
            c[i] = a[i] + b[i];
      }

      /* sequential code */
      for (int i = 0; i < 100; i++) {
            for (int j = 0; j < 10; j++) {
                  int idx = 10 * i + j;
                  printf("%d+%d=%d\n", a[idx], b[idx], c[idx]);
            }
      }
      return 0;
}
```



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread-local storage

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()` usually works as normal** – replace the running process including all threads

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- **Similar to `static` data**
  - TLS is unique to each thread

Linux declare a TLS variable:

__thread int number;

Thread-Local Storage

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

__thread int var = 5;

void* worker1(void* arg);
void* worker2(void* arg);

int main(){
   pthread_t pid1,pid2;

   pthread_create(&pid1,NULL,worker1,NULL);
   pthread_create(&pid2,NULL,worker2,NULL);

   pthread_join(pid1,NULL);
   pthread_join(pid2,NULL);

   return 0;
}

void* worker1(void* arg){
            var++;
            printf("work1: %d\n“,var);
}

void* worker2(void* arg){
            sleep(1); //sleep for 1s
            var += 2;
            printf("work2: %d\n",var);
}
```

TLC_demo.c

gcc –o TLC_demo TLC_demo.c –lpthread
./TLC_demo

What is the output?

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  1. **Asynchronous cancellation** terminates the target thread immediately
  2. **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- **Pthread** code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation **depends on thread state**

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled **through signals**

pthread_kill(pthread_t tid, int signal)

Example: Windows Threads

- Windows API – primary API for Windows applications
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - **Register set** representing state of processor
  - **Separate user and kernel stacks** for when thread runs in user mode or kernel mode
  - **Private data storage area** used by run-time libraries and dynamic link libraries (DLLs)
- The **register set, stacks, and private storage area** are known as the **context** of the thread

Windows Multithreaded C Program

```c
#include <stdio.h>
#include <windows.h>

DWORD sum; // Data shared by all the threads.

// The function executed by the new thread.
DWORD WINAPI runner(LPVOID param) {
    // The new thread is running inside the same process as the main
    // thread, and both threads share the sum variable in memory.
    DWORD Upper = *(DWORD *) param;
    sum = 0;
    for(int i = 0; i <= Upper; i++) {
        sum += i;
        //printf("new thread: sum is: %d\n", sum);
    }
    return 0; // New thread ends.
}
```

Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[]) {
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    // do some basic error checking
    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "an integer >= 0 is required \n");
        return -1;
    }
    // create the thread
    ThreadHandle = CreateThread(NULL, 0, runner, &Param, 0, &ThreadId);
    if (ThreadHandle != NULL) {
        WaitForSingleObject(ThreadHandle, INFINITE);
        CloseHandle(ThreadHandle);
        printf("sum = %d\n",sum);
    }
}
```

Summary

- What is a thread? A thread is a flow of control within a process.
- The benefits of multithreading:

increased responsiveness to the user, resource sharing within the process, economy, and scalability factors, more efficient use of multiple processing cores.

- User-level threads vs kernel threads
- Three models relate user and kernel threads: many-to – one, **one-to-one**, many-to-many.
- Most modern operating systems provide kernel support for threads.
- Thread libraries provide API for creating and managing threads.
- Implicit threading: thread pools, Fork-Join, OpenMP, and Grand Central Dispatch, TBB.
- Issues:
  - the semantics of the fork() and exec() system calls.
  - signal handling, thread cancellation, thread-local storage, and scheduler activations.

End of Lecture 7