



Western
UNIVERSITY • CANADA

Chapter 4 – Threads & Concurrency

Spring 2023

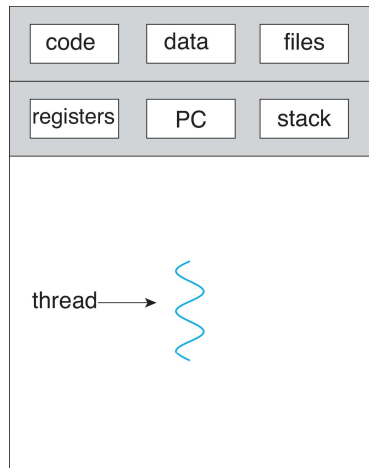
Threads & Concurrency

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Examples

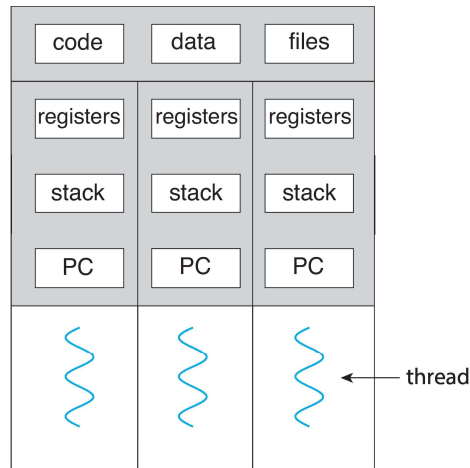
Overview

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
 - E.g. Update display, Fetch data, Spell checking, Answer a network request
- Process creation is **heavy-weight** while thread creation is **light-weight**
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

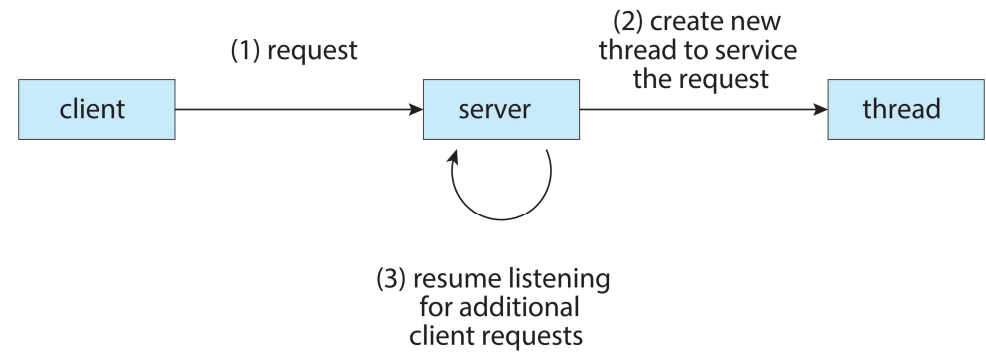
Overview



single-threaded process



multithreaded process



Overview

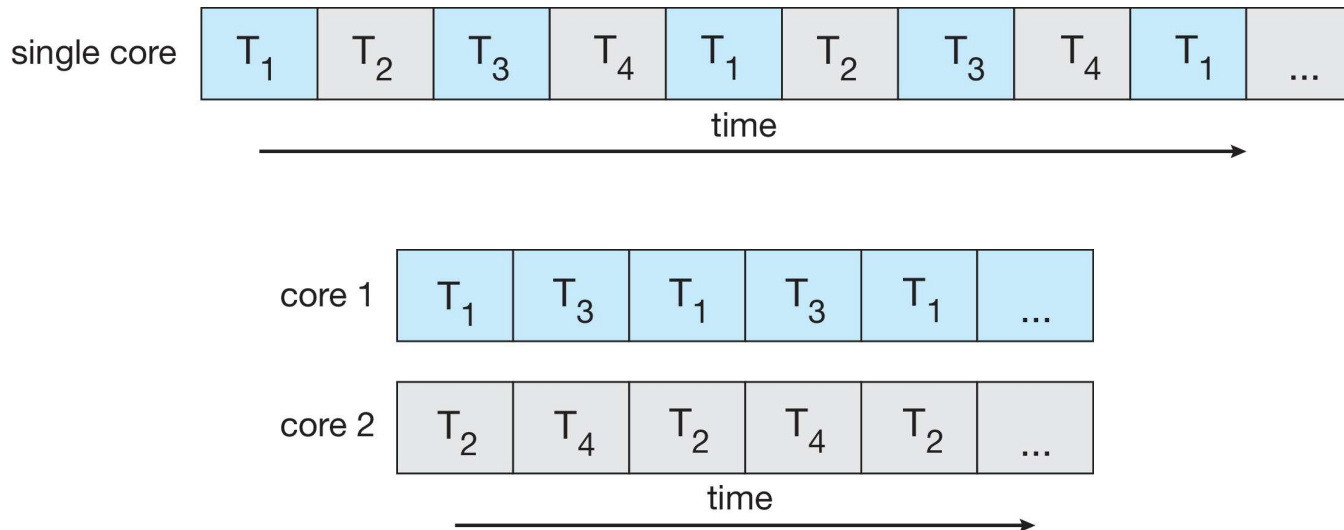
- Benefits
 - **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
 - **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
 - **Economy** – cheaper than process creation, thread switching lower overhead than context switching
 - **Scalability** – process can take advantage of multicore architectures

Multicore Programming

- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency

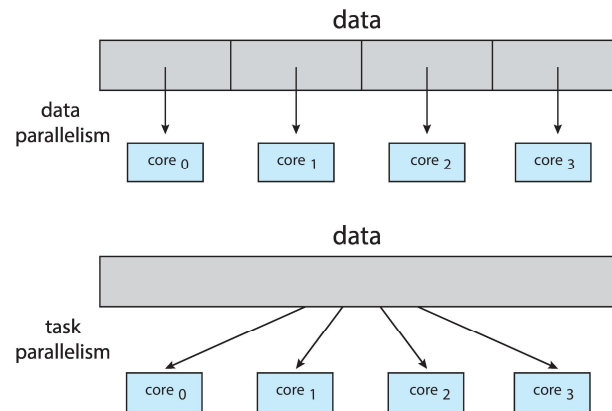
Multicore Programming

- Concurrent execution vs. Parallelism



Multicore Programming

- Types of parallelism
- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
- **Task parallelism** – distributing threads across cores, each thread performing unique operation



Multicore Programming

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
 - Dividing activities
 - Balance
 - Data splitting
 - Data dependency
 - Testing and debugging

Multicore Programming

- **Latency** – the amount of time it takes for a task to run from start to finish
- **Speed up in latency** – the calculated speedup of architecture 2 with respect to architecture 1

$$S_{\text{latency}} = \frac{L_1}{L_2}$$

Multicore Programming

- If a task can be done in 60 seconds when run in serial and 30 seconds when run in parallel, we say the task is run in "1/2 the time" or "2 times faster"
- If a task can be done in 50 seconds when run in serial and 10 seconds when run in parallel, we say the task is run in "1/5th the time" or "5 times faster"
- In general, if a task can be done in X seconds in scenario 1 and Y seconds in scenario 2, then scenario 2 accomplishes the task in $1/(X/Y)$ the time or (X/Y) times faster
- If we know what portion of a task must be run in serial and what portion of a task can be run in parallel, we can predict the speedup in latency when parallelizing the entire task

Multicore Programming

- 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

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

Multicore Programming

- Amdahl's Law
 - That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

$$\frac{1}{\frac{0.25}{1} + \frac{0.75}{2}}$$

- What about 50% parallel / 50% serial on 2 cores? A speedup of ~1.3

$$\frac{1}{\frac{0.5}{1} + \frac{0.5}{2}}$$

- What about 50% parallel / 50% serial on 1024 cores?

$$\frac{1}{\frac{0.5}{1} + \frac{0.5}{1024}}$$

Multicore Programming

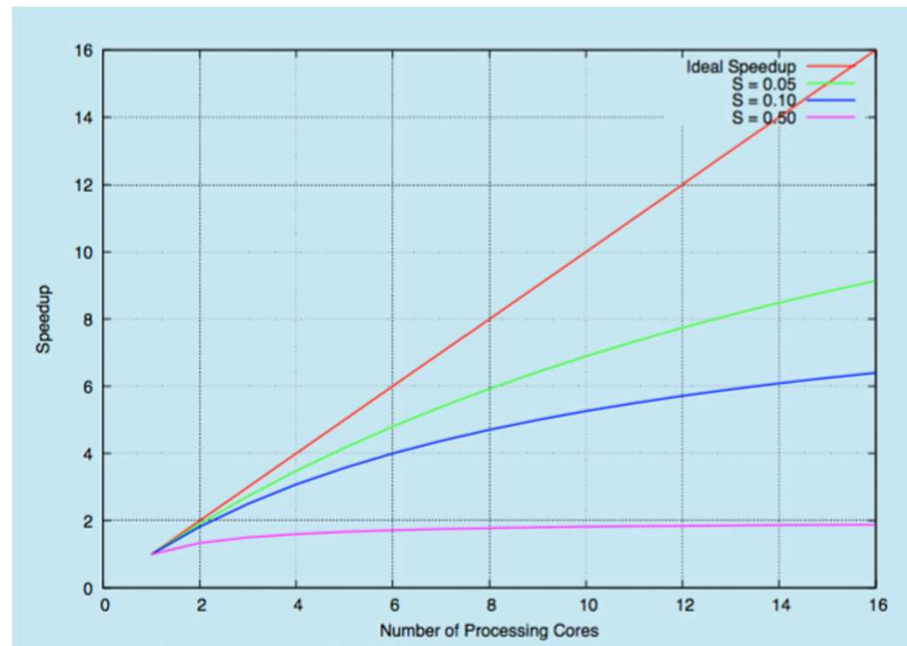
- Amdahl's Law
 - As N approaches infinity, speedup approaches $1 / S$
 - Serial portion of an application has disproportionate effect on performance gained by adding additional cores
 - As S approaches 0, speedup approaches N

Multicore Programming

- Amdahl's Law applies to all parallel tasks, not just cores
- What about a task where some parts are more parallel than others?
 - 11% must be done in serial
 - 18% can be done 5 times faster (1/5th the time)
 - 23% can be done 20 times faster (1/20th the time)
 - 48% can be done 1.6 times faster (1/1.6th the time)

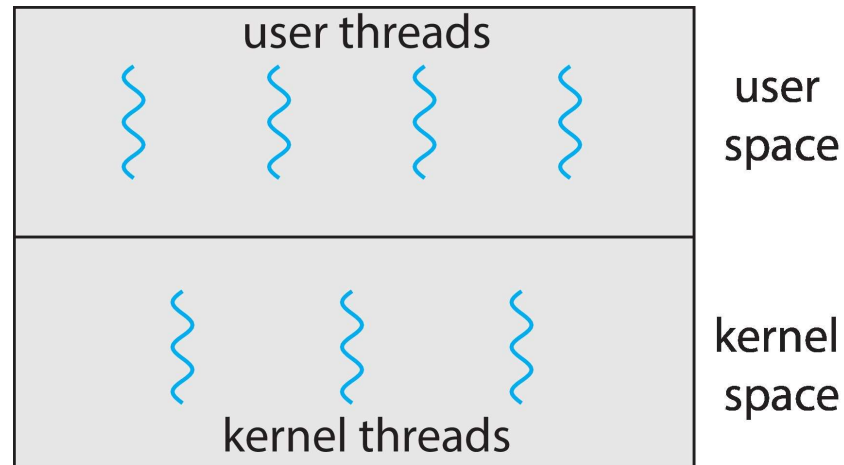
$$S_{\text{latency}} = \frac{1}{\frac{p1}{s1} + \frac{p2}{s2} + \frac{p3}{s3} + \frac{p4}{s4}} = \frac{1}{\frac{0.11}{1} + \frac{0.18}{5} + \frac{0.23}{20} + \frac{0.48}{1.6}} = 2.19.$$

Multicore Programming



Multithreading Models

- User threads - management done by user-level threads library (see the next section)
- Kernel threads - Supported by the Kernel
- Some relationship must exist between user threads and kernel threads

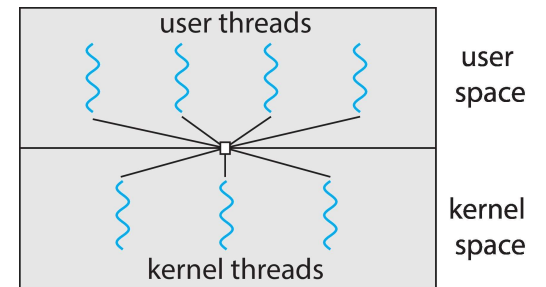
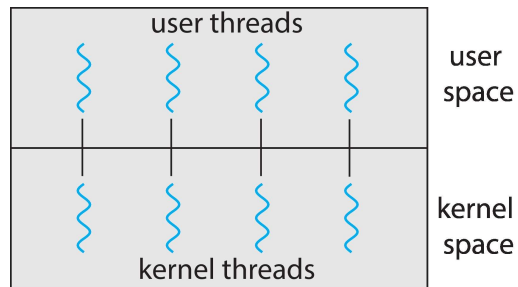
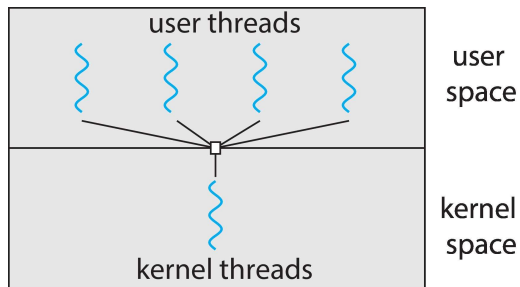


Multithreading Models

- Many-to-One
 - The developer can create as many threads as they wish, but the kernel is limiting parallelism
- One-to-One
 - The developer can create as many threads as they wish, but the kernel may limit or run out of resources
- Many-to-Many
 - The developer can create as many threads as they wish, but the kernel needs to manage threads

Multithreading Models

- Since computers have more cores these days, one-to-one is used in Linux and Windows and is the most common approach
- Many-to-One vs. One-to-One vs. Many-to-Many



Thread Libraries

- POSIX Pthreads
 - Must include `pthread.h`
 - See the manpage: `man pthreads`
 - On Linux, both `fork()` and `pthread_create()` use the system call `clone()`
 - Fork shares nothing. Threads share, for example, file-system information (`CLONE_FS`), memory space (`CLONE_VM`), signal handlers (`CLONE_SIGHAND`) and open files (`CLONE_FILES`).
- Windows threads
- Java threads

Implicit Threading

- Some compilers and run-time libraries will insert threading for developers rather than asking the developer to handle threading explicitly
- The general idea is to just "label" a task (e.g. a function) as "able to be run in parallel in a thread". The compiler or run-time library does the rest.
- Example strategies: Thread pools, Fork-join model, OpenMP, Grand Central Dispatch, Intel Thread Building Blocks

Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred

Threading Issues

- Semantics of `fork()` and `exec()` system calls
 - Does `fork()` duplicate only the calling thread or all threads?
 - On Linux, only the calling thread is duplicated. Therefore, the child process starts single threaded
 - Some UNIXes have two versions of `fork`
 - `exec()` usually works as normal – replace the running process including all threads

Threading Issues

- Signal handling
 - Signals are used in UNIX systems to notify a process that a particular event has occurred.
 - When a signal is sent to a process, its normal execution is interrupted
 - Such events can arise due to internal or external sources:
 - Internal: Manual (intentional) signals, Illegal instruction (e.g. divide by zero)
 - External: e.g. CTRL+C or `kill` command
 - Upon receipt of a signal, the process takes some action (the "handler")

Threading Issues

- Signal handling
 - A signal handler is used to process signals
 - Signal is generated by particular event
 - Signal is delivered to a process
 - Signal is handled by one of two signal handlers: default or user-defined

Threading Issues

- Signal handling
 - Important signals. See `man 7 signal`
 - SIGALRM – Timer signal. call with `alarm()` or `raise(SIGALRM)`
 - SIGINT – Interrupt from keyboard (e.g. CTRL+C)
 - SIGTERM – Termination signal (e.g. `kill <pid>`)
 - SIGKILL – CANNOT be caught. No user-defined option (e.g. `kill -9 <pid>`)

Threading Issues

- Signal handling. For single-threaded, signal delivered to process
 - Where should a signal be delivered for multi-threaded? Depends on the signal
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process (e.g. SIGKILL)
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process
- `kill(pid_t pid, int signal)` vs.
`pthread_kill(pthread_t tid, int signal)`

Threading Issues

- Signal handling
 - Every signal has a default handler that the kernel runs when handling signal
 - User-defined signal handler can override default
 - ```
#include <signal.h>
int alarmflag=0;
void alarmHandler (){
 printf("An alarm clock signal was received\n");
 alarmflag = 1;
}
```

# Threading Issues

- ```
void main() {  
    signal(SIGALRM, alarmHandler);  
    alarm(3);  
    printf("Alarm has been set\n");  
    while (alarmflag==0);  
    printf("Back from alarmHandler function\n");  
}
```

Threading Issues

- Thread cancellation of target thread
 - Terminating a thread before it has finished
 - Thread to be canceled is **target thread**
 - Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Threading Issues

- Thread cancellation of target thread
 - Use `pthread_cancel()` – but this is only a request
 - Use `pthread_setcanceltype()` to set it. Use `pthread_testcancel()` to check for cancellation requests

```
pthread_t tid;

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

. . .

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

/* wait for the thread to terminate */
pthread_join(tid, NULL);
```


Examples

- Create a thread
 - ```
int pthread_create(pthread_t* thread, pthread_attr_t* attr,
void * (start_routine), void *arg);
```

    - Returns 0 to indicate success, otherwise returns error code
    - `thread` – name of the new thread
    - `attr` – argument that specifies the attributes of the thread to be created (NULL = default attributes)
    - `start_routine` – function to use as the start of the new thread
    - `arg` – argument to pass to the new thread routine

# Examples

- Waiting for a thread to complete
  - `int pthread_join(pthread_t thread, void **retval);`
    - Returns 0 to indicate success, otherwise returns error code
    - `thread` – name of the new thread
    - `retval` – copy the return value of `pthread_exit()` into the address of `retval`
  - If you do not wait for a thread to complete, the parent process may get cleaned up before the thread can do its work

# Examples

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

```
void *do_work() {
    printf("Hello, World!...I am a thread\n");
}

int main(int argc, char **argv) {
    pthread_t worker_thread;
    if (!pthread_create(&worker_thread, NULL, do_work, NULL)) {
        printf("Error while creating thread\n");
    }
    pthread_join(worker_thread, NULL);
}
```

↑ name of the new thread.

↑ if join is not here,

Examples

- `//3 threads`
`#include <stdio.h>`
`#include <unistd.h>`
`#include <pthread.h>`

`...`
`int main(int argc, char **argv) {`
 `pthread_t worker_thread[3];`
 `for (int i = 0; i<3; i++){`
 `if (!pthread_create(&worker_thread[i], NULL, do_work, NULL)) {`
 `printf("Error while creating thread\n");`
 `}`
 `}`
 `for (int i = 0; i<3; i++){`
 `if (!pthread_join(worker_thread[i], NULL)){`
 `printf("Error joining with thread");`
 `}`
 `}`
`}`

Examples

- Passing data to threads
 - Typically, you want to create threads that work independently, so passing data back and forth is unusual
 - However, there are a few ways to copy data between the parent and its threads or between threads themselves
 - Global variables (Remember, threads, **unlike** forked processes, can share global variables. This can be useful but be careful about updating them. Solutions in chapter 6) *if we're using forking, this cannot be used.*
 - Arguments to pthread_create, pthread_exit, pthread_join
 - We could use a pipe as an argument

Examples

- `//Pass in a message`
`#include <stdio.h>`
`#include <unistd.h>`
`#include <pthread.h>`

```
void *thread_prints_msg(void *msg) {  
    printf("From thread_prints_msg: %s\n", (char *) msg);  
}
```

```
int main(int argc, char **argv){  
    pthread_t thread_1;  
    printf("From main: Going to create Thread...\n");  
    pthread_create(&thread_1, NULL, thread_prints_msg, "Hello, World!");  
    pthread_join(thread_1, NULL);  
    printf("thread terminates...\n");  
    return 0;  
}
```

*the return type have to
be the same as accept type.*

casting the input as str.

*"Hello, World!"
passing str*

Examples

- //3 threads. Pass data in only

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

```
void *do_work(void * args) {
    int threadNumber = *(int *) args;
    printf("Hello, World!...I am thread %d\n", threadNumber);
}
```

```
int main(int argc, char **argv) {
    pthread_t worker_thread[3];
    int threadNumber[3];
    for (int i = 0; i<3; i++){
        threadNumber[i] = i; //Each thread needs to use its own variable
        if (!pthread_create(&worker_thread[i], NULL, do_work, &threadNumber[i])) {
            printf("Error while creating thread\n");
        }
    }
}
```

...

this place is not updated in future for loop

Examples

- `//15 threads. Pass data in and get data out`
`#include <stdio.h>`
`#include <unistd.h>`
`#include <pthread.h>`

`#define MAX_NUMBER 15`
`void *do_work(void * args) {`
 `int *square = (int *)args;`
 `*square *= *square;`
`}`

Examples

```
• int main(int argc, char **argv) {  
    pthread_t worker_thread[MAX_NUMBER];  
    int squareNumber[MAX_NUMBER];  
    for (int i = 0; i<MAX_NUMBER; i++){  
        squareNumber[i] = i;  
        if (!pthread_create(&worker_thread[i], NULL, do_work, &squareNumber[i])) {  
            printf("Error while creating thread\n");  
        }  
    }  
    for (int i = 0; i<MAX_NUMBER; i++){  
        if (!pthread_join(worker_thread[i], NULL)){  
            printf("Error joining with thread");  
        }  
        printf("%d squared is %d\n", i, squareNumber[i]);  
    }  
}
```

Examples

- ```
//Use a pipe
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void * dowork(void* args){
 int *port = (int *) args;
 char s[4];
 read(port[0],&s,sizeof(s));
 printf("Thread finds %s\n", s);
}
```

# Examples

```
• int main(int argc, char **argv) {
 int port[2];
 if (pipe(port) < 0){
 perror("pipe error");
 exit(1);
 }
 char s[4] = "ABC"; //Note: sizeof(s) == 4
 write(port[1],s,sizeof(s)); // write string
 ...
 pthread_create(&tid, NULL, dowork, port);
 ...
}
```



Western  
UNIVERSITY • CANADA