SHARED MEMORY PARALLEL COMPUTING

COMP4300/8300 PARALLEL SYSTEMS

PROF. JOHN TAYLOR

APRIL 2024

Australian National University

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Logistics

- Personal attendance to lectures highly encouraged
- Lecture material uploaded on Wattle and Parallel Systems website before the live lecture
- Careful with usage of Gadi resources
- Assignment 1 due this week on Thu, April XXth @midnight

  Assignment 2 *released* on Fri, April XXst; *due* on Fri, May 26th, 5 PM. Results of MSE released on Wattle today
- Feedback (also through SELT, opening soon): *We are here to support you learning!*

Logistics

- COMP4300 is a fast-paced 4000-level course: **It assumes you are mature and independent programmers**.
- The course introduces the basics of the semantics of the programming models (Pthreads, OpenMP, CUDA).
- **You are left with the task and the responsibility of their further exploration and practice to master these programming models**.
- This is particularly important for the second half of the course and a main difference with COMP3320.

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

References

- Chapter 12 from Computer Systems A Programmer's Perspective, Third Edition, Randal E. Bryant and David R. O'Hallaron, Pearson Education Heg USA, ISBN 9781292101767.

- Programming with POSIX Threads, David R. Butenhof, Addison-Wesley Professional, ISBN-13 : 978-0201633924.

PARALLEL COMPUTERS & PROGRAMMING MODELS, PTHREADS

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

USING MULTIPLE CORES

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

How can we avoid the flaws of process-based concurrency?

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Parallel Programming with Threads

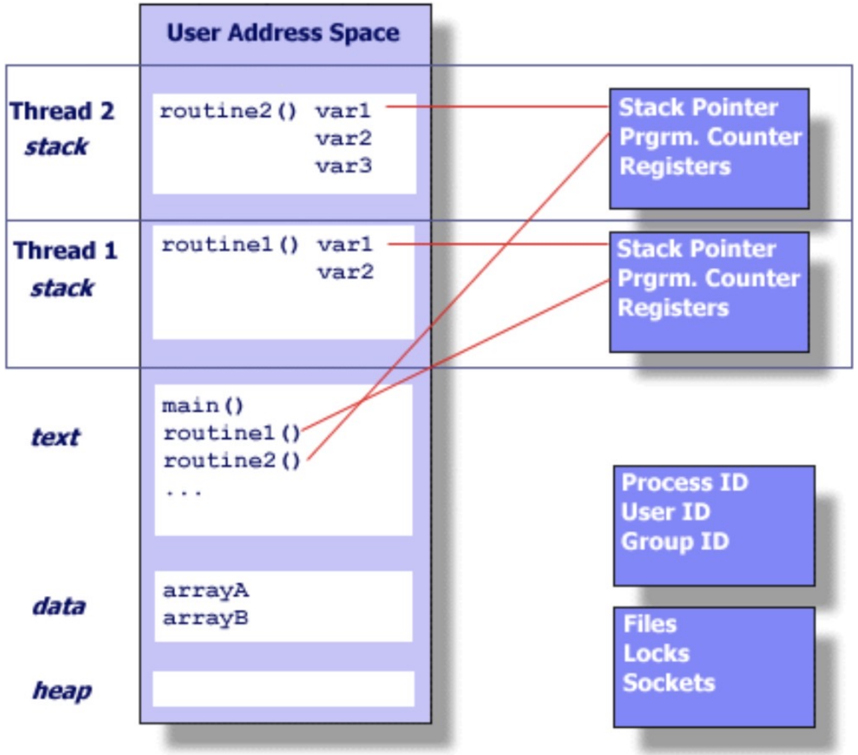
A *thread* is a logical flow that runs within the context of a process.

- So far we have discussed programs that consisted of a single thread per process.
- Multiple "independent" threads can be added to an existing process rather than starting a new process.
- Threads are scheduled by the OS and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.

Understanding Pthreads concepts helps developers grasp fundamental parallel programming principles

Parallel Programming with Threads

- A thread is a logical flow that runs within the context of a process.
- So far we discussed programs that consisted of a single thread per process.
- Multiple "independent" threads can be added to an existing process rather than starting a new process.
- Threads are able to be scheduled by the OS and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.

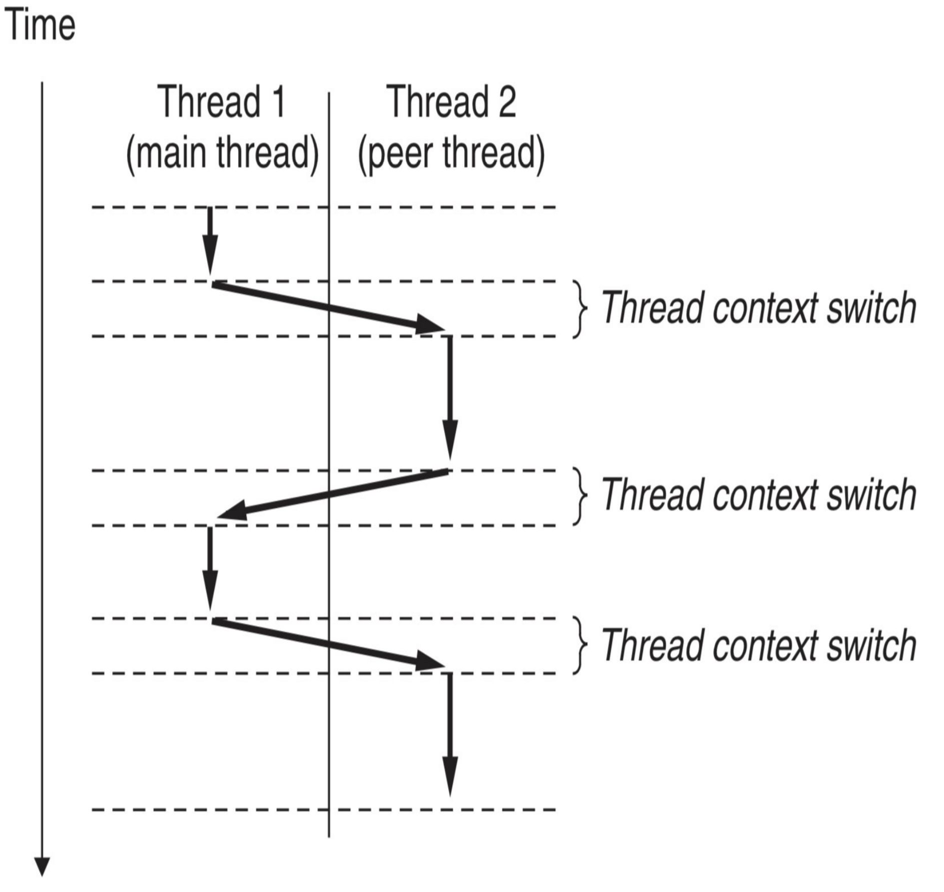


THREADS WITHIN A UNIX PROCESS

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Thread Execution Model

- Each process is created with a single thread called the main thread.
- The main thread can create a peer thread, and the two run concurrently.
- If they are scheduled on the same CPU/core, the OS at some point will pass control from the main to the peer thread via a context switch.
- If they are scheduled by the OS on different cores they will run in parallel.
- Threads associated with a process form a pool of peers, where each one can kill or wait for any other to terminate.
- Each peer can read and write the same shared data.



POSIX Threads (Pthreads)

- Historically, hardware vendors have implemented their own proprietary versions of threads, making it difficult for programmers to develop portable threaded applications.
- POSIX Threads provide a standard interface, specified by the IEEE 1003.1 (1995) standard, for manipulating threads from C programs.
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with an #include <pthread.h> header file and a thread library (this may be part of another library, such as libc, in some implementations).
- The specification contains about 60 functions that allow programs to create, kill, and reap threads, to share data safely with peer threads, and to notify peers about changes in the system state.

Compiler / Platform	Compiler Command	Description
INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, Blue Gene	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++

POSIX Threads (Pthreads)

A Hello world example

```c
#include <pthread.h>
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

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

Pthread_create: This function creates a new thread:

- The first argument references the pthread_t thread variable.
- The second argument can specify attributes (NULL for default).
- The third argument is the function the thread will run.
- The fourth argument can pass data to the thread's function (NULL here).

Pthread_join: This function in the main thread waits for the created thread to terminate.

This prevents the main program from exiting before the thread has printed its output.

POSIX Threads (Pthreads)

Creating Threads

- Creates a new thread and runs the thread routine `f` in the context of the new thread and with an input argument of `arg`.
- The thread routine `f` takes as input a single generic pointer and returns a generic pointer. If you want to pass multiple arguments to a thread routine, then you should put the arguments into a structure and pass a pointer to the structure.
- Similarly, if you want the thread routine to return multiple arguments, you can return a pointer to a structure.
- When `pthread create` returns, argument `tid` contains the ID of the newly created thread, which can also be determined using

```
#include <pthread.h>
typedef void *(func)(void *);

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

Returns: 0 if OK, nonzero on error

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns: thread ID of caller

POSIX Threads (Pthreads)

Terminating Threads

- *Implicit* termination when its top-level thread routine returns.
- *Explicit* termination using `pthread exit`. Explicit termination of main thread and will wait for all other peers to terminate.
- A peer thread calls `pthread cancel` with the ID of the current thread.
- A peer thread calls `exit` terminating the process and all its threads.

```
#include <pthread.h>

void pthread_exit(void *thread_return);
```

Returns: 0 if OK, nonzero on error

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Returns: 0 if OK, nonzero on error

POSIX Threads (Pthreads)

Returning and reaping

- Current thread waits for thread `tid` to terminate, blocking until it does so.
- Assigns the generic `(void *)` pointer returned by the thread routine to the location pointed to by `thread return`.
- *Reaps* any memory resources (e.g. stack) held by the terminated thread. Reaping memory resources involves managing and reclaiming memory that is no longer needed by a program

```
#include <pthread.h>

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

Returns: 0 if OK, nonzero on error

POSIX Threads (Pthreads)

Detaching threads

- A thread is either *joinable* or *detachable*.
- *Joinable*: its memory resources (such as the stack) are not freed until it is reaped by another thread.
- A detached thread cannot be reaped by other threads. Its memory resources are freed automatically by the system when it terminates.
- To avoid memory leaks, each joinable thread should either be explicitly reaped by another thread, or detached by a call to `pthread detach`

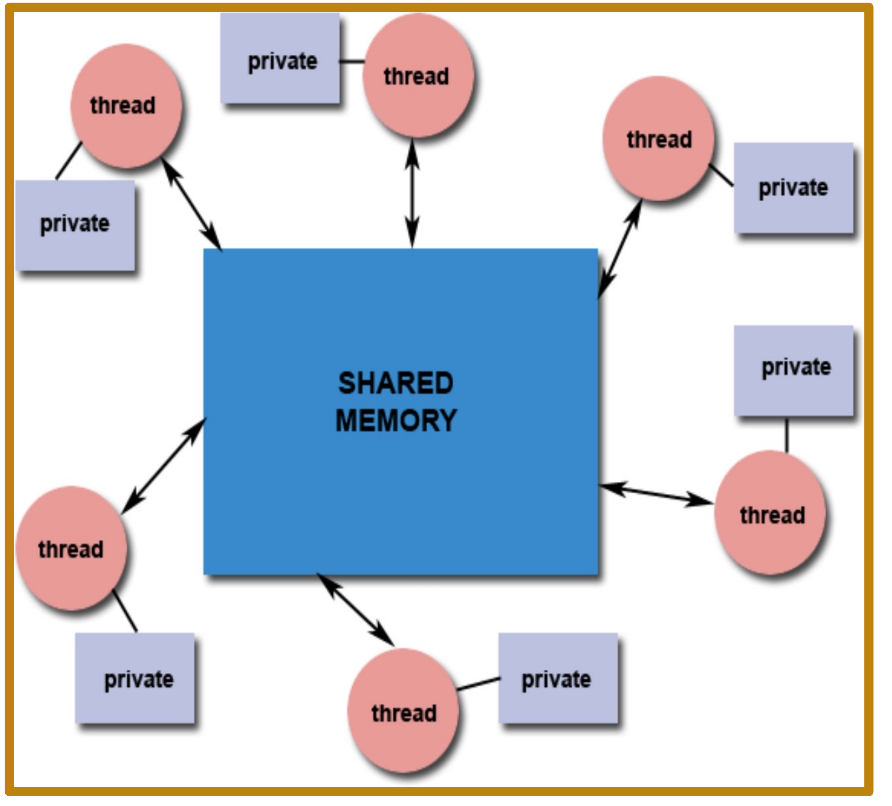
```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, nonzero on error

Threads can detach themselves by calling pthread detach with an argument of `pthread_self`.

Pthreads Memory Model

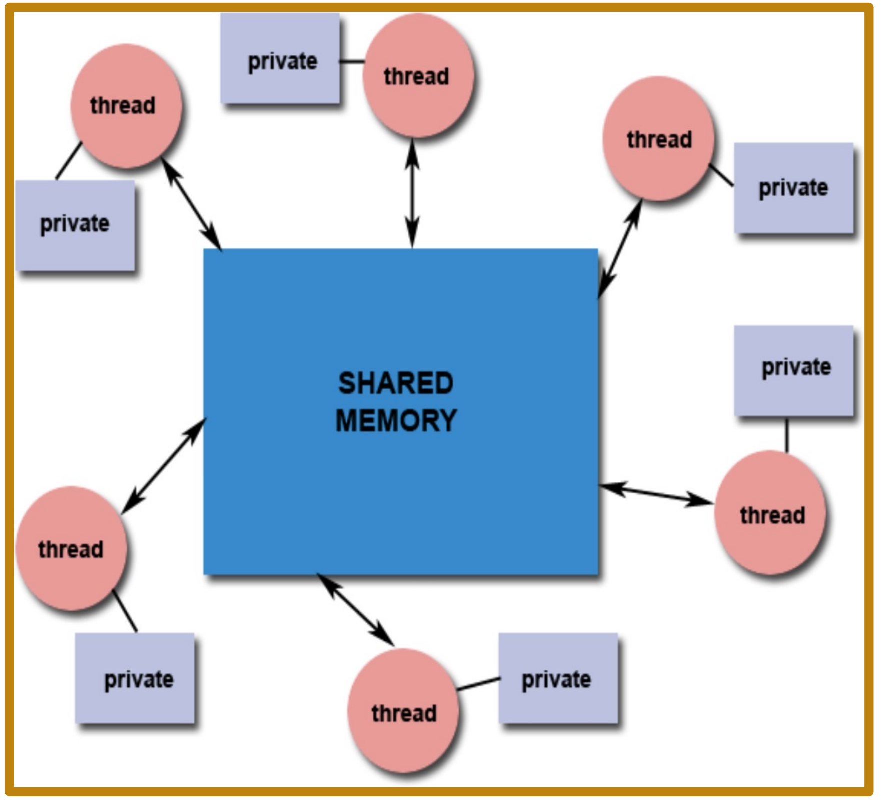
- The key advantage of threads is the efficiency and ease with which they can share data
- The key disadvanatge is the potential for complicated bugs that do not appear in serial code.
- To write correct threaded programs a clear understanding of the sharing mechanisms and risks is required



Pthreads Memory Model

- **Thread context**: TID, stack, stack pointer (SP), program counter (PC), condition codes (CC), registers values.

- **Shared**: process virtual address space → code, read/write data, heap, shared libraries, open files.



Pthreads Memory Model

- Registers, Condition Codes (CC), Program Counter (PC), the Stack Pointer (SP) are private. Thread stack is usually accessed independently by each thread. However, thread stacks are not protected!
- If a thread accesses a stack pointer to another thread's stack, it can read and write to it!

```
#include "csapp.h"
#define N 2
void *thread(void *vargp);

char **ptr;  /* Global variable */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

Mapping Variables to Memory

- ***global* variables**: declared outside functions, only one copy in virtual memory area, readable/writable by any thread.
- ***local automatic* variables**: declared inside functions, each thread's stack contains its own instance of it. E.g. tid and myid.
- ***local static* variables**: declared inside functions with the static attribute. There is only one copy in virtual memory area, readable/writable by any thread.
- ***shared* variables**: a variable is shared only, and only if, one of its instances is referenced by more than one thread. E.g. cnt.

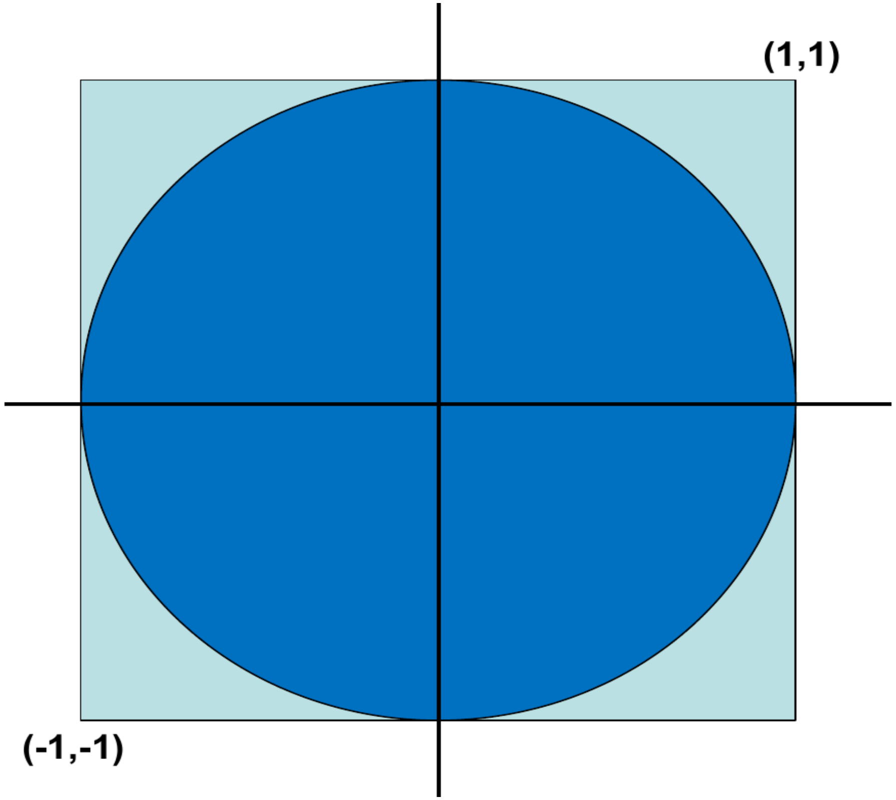
```c
#include "csapp.h"
#define N 2
void *thread(void *vargp);

char **ptr;  /* Global variable */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Pthreads

Example: Computing Pi

- The ratio of area of circle *radius=1* to the square is $\pi/4$
- Generate random numbers for x and y within the domain of square ie range [-1,1] for each axis
- identify those that are distance less than 1 from origin
- The ratio of points in circle to the total points is $\pi/4$



Pthreads

Example: Computing Pi

- The ratio of area of circle *radius=1* to the square is $\pi/4$
- Generate random numbers for x and y within the domain of square ie range [-1,1] for each axis
- identify those that are distance less than 1 from origin
- The ratio of points in circle to the total points is $\pi/4$

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

#define NUM_THREADS 4
#define NUM_THROWS 1000000

double total_points_inside_circle = 0;

int main() {
  pthread_t threads[NUM_THREADS];
  int thread_ids[NUM_THREADS];

  srand(time(NULL));

  for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i;
    pthread_create(&threads[i], NULL, computePI, (void*)&thread_ids[i]);
  }

  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }

  double pi_estimate = 4.0 * total_points_inside_circle / NUM_THROWS;
  printf("Estimated value of π: %lf\n", pi_estimate);

  return 0;
}
```

TEQSA PROVIDER ID: PRV12002 (AUSTRALIAN UNIVERSITY) CRICOS PROVIDER CODE: 00120C

Pthreads

Example: Computing Pi

- The ratio of area of circle *radius=1* to the square is $\pi/4$
- Generate random numbers for x and y within the domain of square ie range [-1,1] for each axis
- identify those that are distance less than 1 from origin
- The ratio of points in circle to the total points is $\pi/4$

```
double total_points_inside_circle = 0;

void* computePI(void* arg) {
  int myNum = *(int*)arg;
  double x, y;

  for (int i = myNum; i < NUM_THROWS; i += NUM_THREADS)
{
    x = (double)rand() / RAND_MAX * 2.0 - 1.0;
    y = (double)rand() / RAND_MAX * 2.0 - 1.0;

    if (x * x + y * y <= 1.0) {
      total_points_inside_circle++;
    }
  }

  pthread_exit(NULL);
}
```