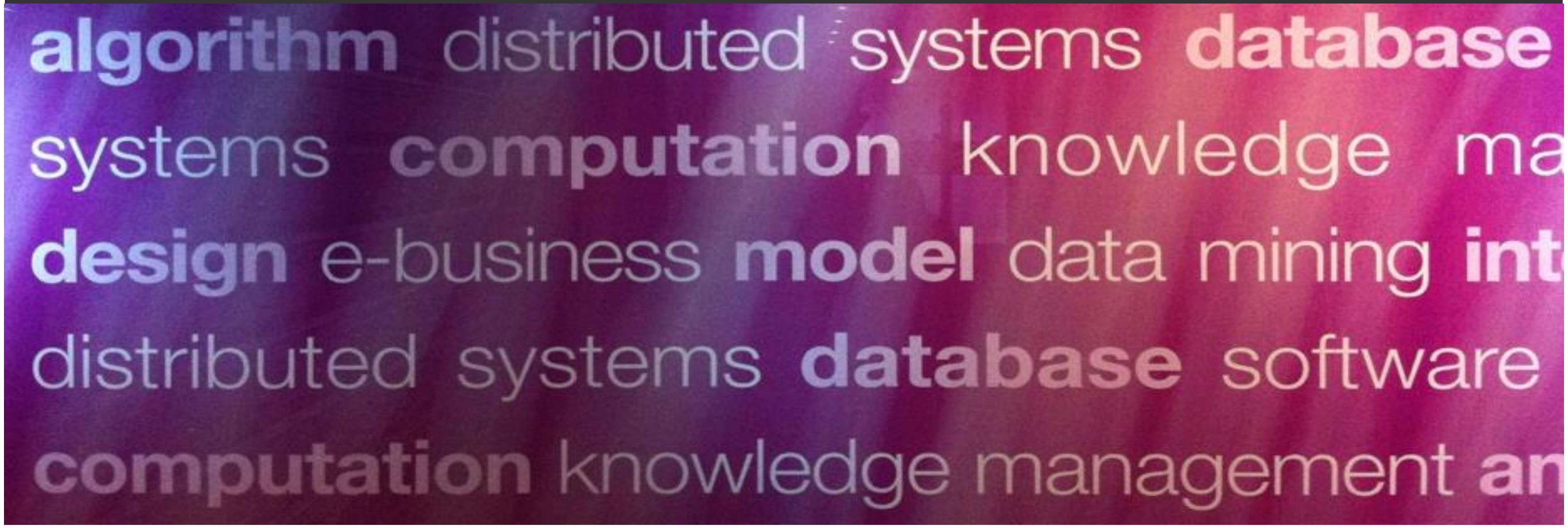
MONASH University

Information Technology

FIT3143 - LECTURE WEEK 2

PARALLEL COMPUTING ON SHARED MEMORY WITH POSIX


algorithm distributed systems database
systems computation knowledge ma
design e-business model data mining int
distributed systems database software
computation knowledge management an

Overview

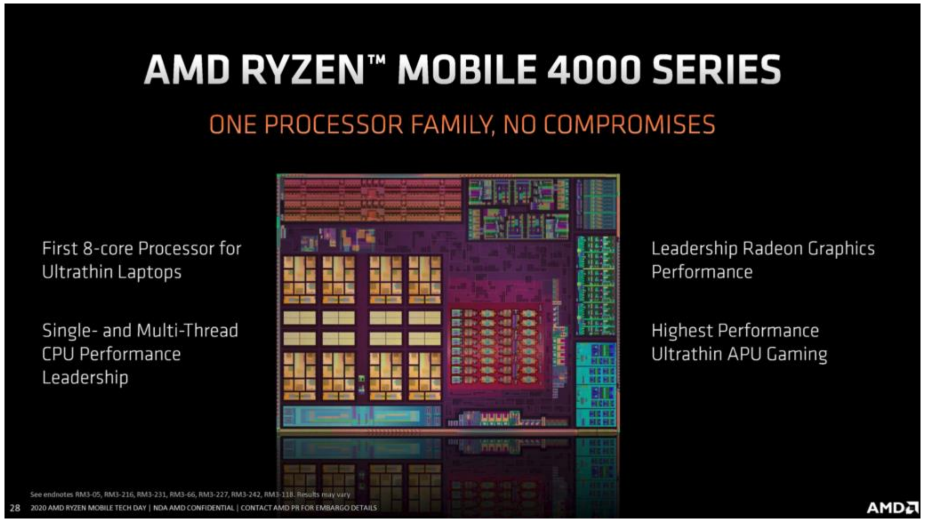
1. Shared memory architecture and constructs for specifying parallelism
2. POSIX for shared memory parallel programming

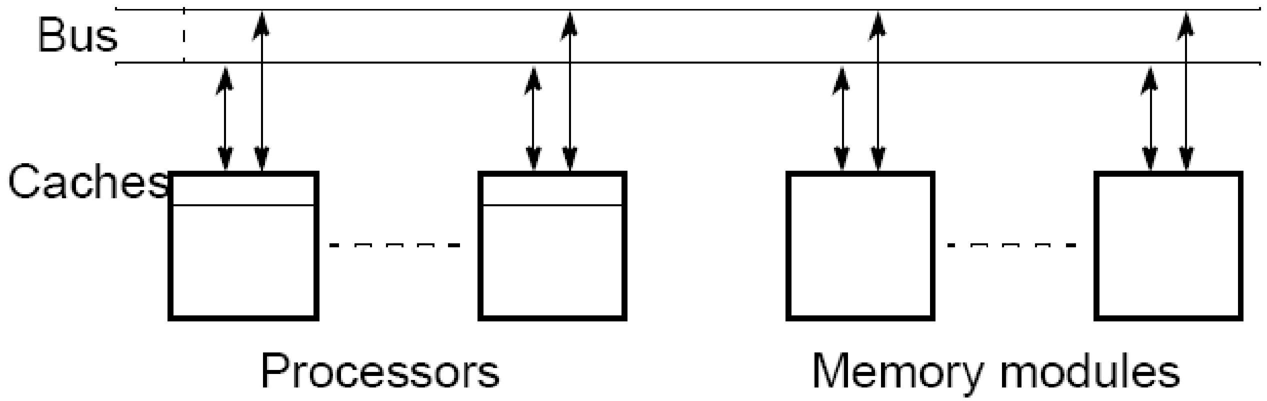
Associated learning outcomes

- Explain the fundamental principles of parallel computing architectures and algorithms (LO1)
- Design and develop parallel algorithms for various parallel computing architectures (LO3)

Shared Memory architecture

- Any memory location can be accessible by any of the processors or cores within a processor.
- A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.


AMD RYZEN™ MOBILE 4000 SERIES
ONE PROCESSOR FAMILY, NO COMPROMISES
First 8-core Processor for Ultrathin Laptops
Single- and Multi-Thread CPU Performance Leadership
Leadership Radeon Graphics Performance
Highest Performance Ultrathin APU Gaming
AMD



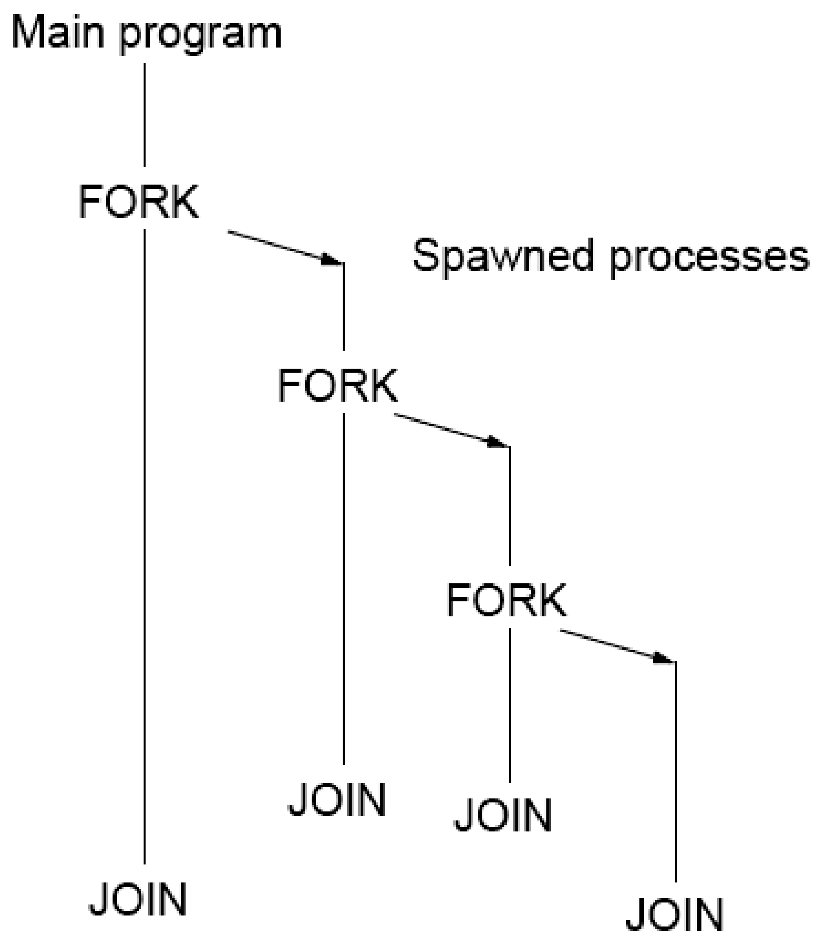
Constructs for specifying parallelism

A) Creating Concurrent Processes

- The first example of a structure to specify concurrent processes is the **FORK-JOIN** group of statements described by CONVEY (1963).
- **FORK-JOIN** constructs have been applied as extensions to FORTRAN and to the UNIX operating system.
- **FORK** statement generates one new path for a concurrent process and the concurrent processes use **JOIN** statements at their ends. When both **JOIN** statements have been reached, processing continues in a sequential fashion.
- For more concurrent processes, additional **FORK** statements are necessary.
- The **FORK-JOIN** constructs are shown nested in the figure on the next slide.

Constructs for specifying parallelism

- Each spawned process requires a **JOIN** statement at its end, which brings together the concurrent processes to a single terminating point.
- Only when all concurrent processes have completed can the subsequent statements of the main process be executed.
- A counter is used to keep record of processes not completed.



FORK- JOIN construct

Constructs for specifying parallelism

UNIX Heavyweight Processes

- The **UNIX** system call **fork()** creates a new process.
- The new process (child process) is an *exact copy* of the calling process except that it has a unique process ID.
- It has its own copy of the parent’s variables.
- They are assigned the same values as the original variables initially.
- The forked process starts execution at the point of the fork.
- On success, **fork()** returns 0 to the child process and returns the process ID of the child process to the parent process.

Constructs for specifying parallelism

Processes are “joined” with the system calls **wait()** and **exit()** defined as:

```
wait(statusp);     /*delays caller until signal received or one of its child
                   /* processes terminates or stops */
exit(status);      /*terminates a process */
```

A single child process can be created by

```
            .
pid = fork();                                /* fork */
   Code to be executed by both child and parent
if (pid == 0) exit(0); else wait(0);   /* join */
            .
            .
            .
```

Constructs for specifying parallelism

If the child is to execute different code, we could use

```
pid = fork();
if (pid == 0) {
    code to be executed by slave
} else {
    code to be executed by parent
}
if (pid == 0) exit(0); else wait(0);
        .
        .
```

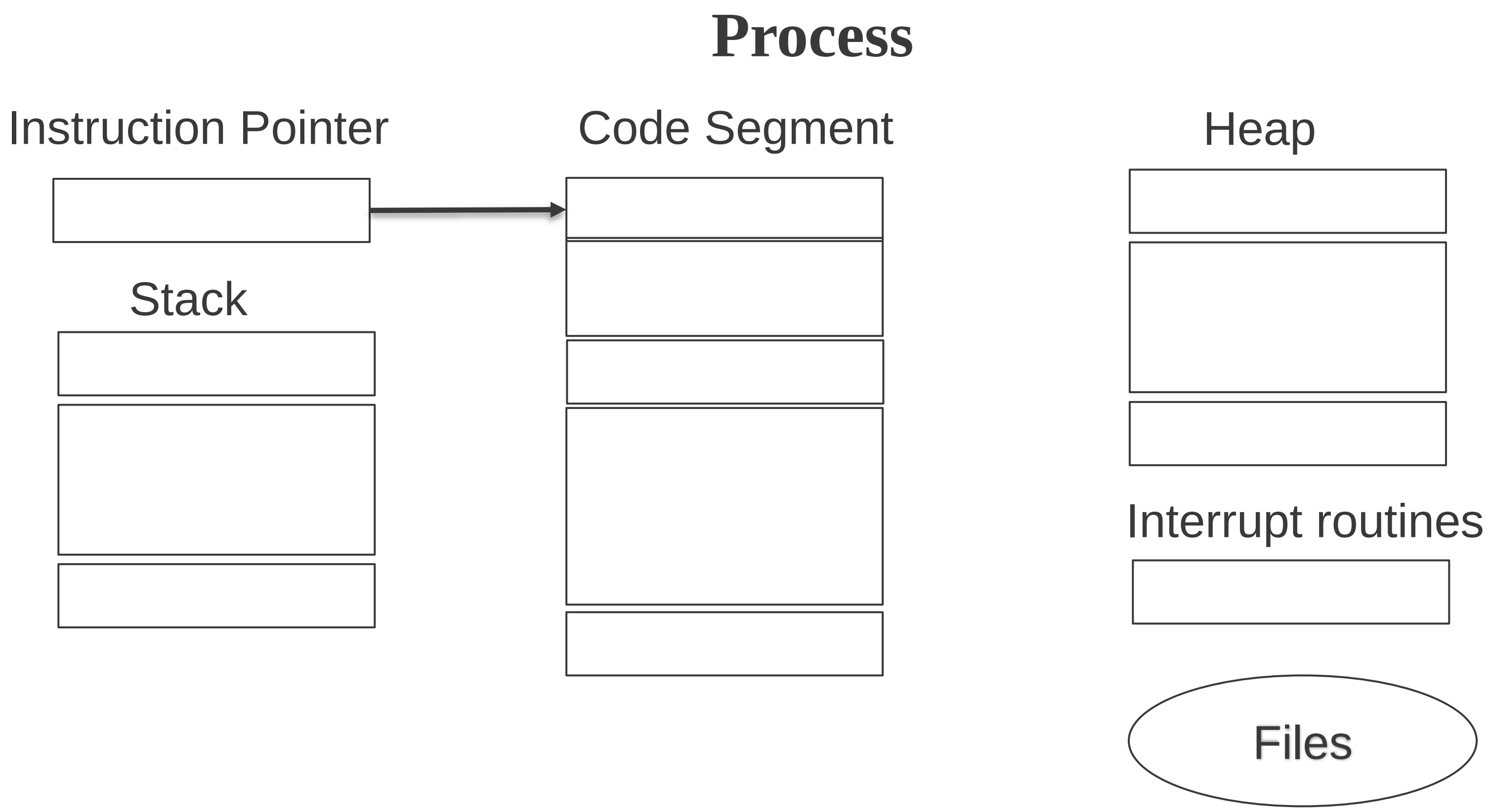
- All the variables in the original program are duplicated in each process, becoming local variables for the process.
- They are assigned the same values as the original variables initially. The forked process starts execution at the point of the fork.

Constructs for specifying parallelism

B) Creating Threads

- ❑ As shown before, the process created with UNIX fork is a “heavy weight” process. It is a completely separate program with its own variables, stack and memory allocation.
- ❑ Heavyweight processes are particularly expensive to create in time and memory space. A complete copy of the process with its own memory allocation, variables, stack etc., is created even though execution only starts from the forked position.
- ❑ A much more efficient mechanism is one in which independent concurrent routine is specified that *shares* the same memory space and global variables. This can be provided by a mechanism called THREAD or LIGHTWEIGHT PROCESS.
- ❑ The difference between processes and threads and the basic parts of a process are shown in the following slide.

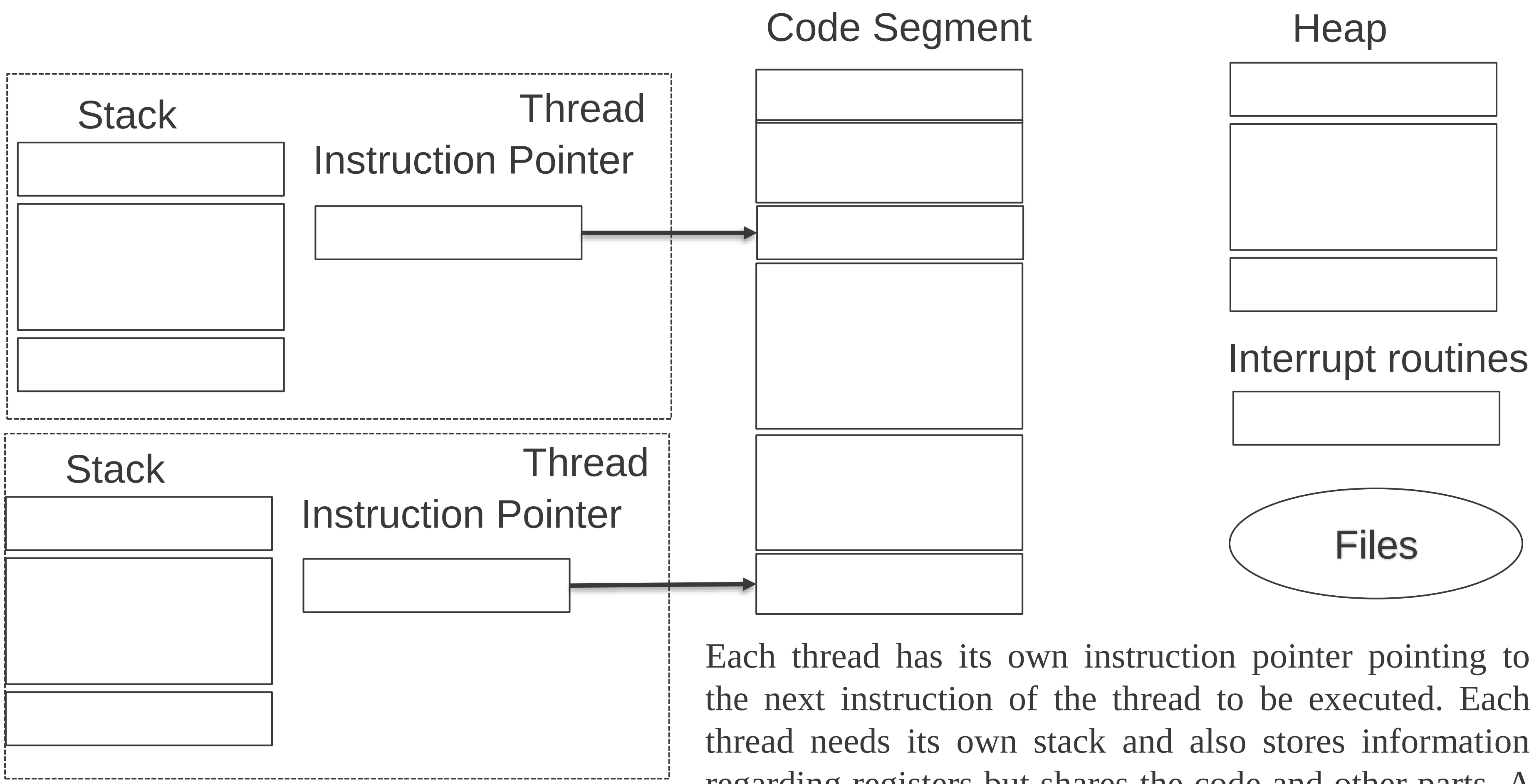
Constructs for specifying parallelism



An instruction pointer holds address of the next instruction to be executed. A stack is present for procedure calls, and a heap, system routines and files.

Constructs for specifying parallelism

Threads



Each thread has its own instruction pointer pointing to the next instruction of the thread to be executed. Each thread needs its own stack and also stores information regarding registers but shares the code and other parts. A process can have more than one thread.

Constructs for specifying parallelism

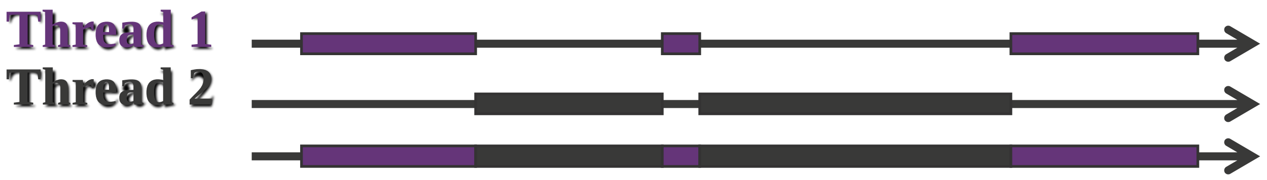
- ❑ Creation of threads is faster than creation of processes.
- ❑ A thread will immediately have access to shared global variables.
- ❑ Synchronization of threads can be done much more efficiently than synchronization of processes.
- ❑ Whenever an activity of thread is delayed or blocked, such as waiting for I/O, another thread can take over.
- ❑ Multithreading also helps alleviate the long latency of message passing; the system can switch rapidly from one thread to another while waiting for messages and provides a powerful mechanism for latency hiding.
- ❑ Examples of multithreaded operating systems include SUN Solaris and Windows NT
- ❑ A widely available standard is **PTHREADS** (IEEE Portable Operating Systems Interface, POSIX, Section 1003.1)

Threads

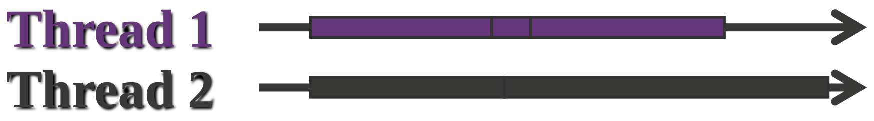
In shared address space architecture, communication is implicitly specified since some (or all) of the memory is accessible to all the processors. Consequently, programming paradigms for shared address space machines focus on constructs for expressing concurrency and synchronization along with techniques for minimizing associated overheads.

Concurrency vs. Parallelism

- Concurrency: two or more threads are <u>in progress</u> at the same time:



- Parallelism: two or more threads are executing at the same time



Thread Basics

A thread is a single stream of control in the flow of a program.

Code segment for computing the product of two dense matrices of size *n* x *n*:

```
for(row = 0; row < n; row++)
   for(column = 0; column < n; column++)
           c[row][column] = dot_product(get_row(a,row), get_col(b,col));
```

There are **n**^{2} iterations which can be executed independently.

The independent sequence of instructions is referred to as a thread leading to n^2 threads.

The threads can be scheduled concurrently on multiple processors.

```
for(row = 0; row < n; row++)
   for(column = 0; column < n; column++)
           c[row][column] = create_thread(dot_product(get_row(a,row),
                                                          get_col(b,col)));
```

Thread Basics

On a **single processor**, multithreading generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads.

This **context switching generally happens frequently enough** that the user **perceives** the threads or tasks as running at the same time.

On a **multiprocessor or multi-core system**, the threads or tasks will generally run at the same time, with each processor or core running a particular thread or task.

Advantages and Disadvantages of Threads

Software portability: Threaded applications can be developed on serial machines and on parallel machines without changes (migrating programs between diverse architectural platforms).

Latency hiding: One of the major overheads in programs is the access latency for memory access, I/O and communication. Multiple threads are executing on the same processor, thus hiding this latency. While one thread is waiting for a communication operation, other threads can utilize the CPU, thus masking associated overhead.

Advantages and Disadvantages of Threads

Scheduling and load balancing: A programmer must express concurrency in a way that minimizes overheads of remote interaction and idling. In many structured applications, the task of allocating equal work to processors is easily accomplished. In unstructured and dynamic applications (e.g. game playing and discrete optimization) this task is more difficult. Threaded APIs allow the programmer to specify a large number of concurrent tasks and support system-level dynamic mapping of tasks to processor with a view to minimizing idling overheads, thus there is no need for explicit scheduling and load balancing.

Ease of programming, widespread use: Threaded programs are easier to write than programs using message passing APIs. With widespread acceptance of POSIX thread API, development tools for POSIX are more widely available and stable.

Threads

- A number of vendors provide vendor-specific thread APIs. The IEEE specifies a standard 1003.1c-1995, POSIX API (***Pthreads***) – standard threads API supported by most vendors. Other thread APIs: *Win32 or Windows* threads*, NT threads, Solaris threads, Java threads, etc.

2) POSIX* Thread or Pthread

Thread Creation & Synchronization

POSIX Threads

What are Pthreads?

- POSIX.1c standard
- C language interface
- Threads exist within same process
- All threads are peers
  - No explicit parent-child model
  - Exception: “main thread” holds process information

pthread_create

int pthread_create(tid, attr, function, arg);

pthread_t *tid
  handle of created thread
const pthread_attr_t *attr
  attributes of thread to be created
void *(*function)(void *)
  function to be mapped to thread
void *arg
  single argument to function

- Spawn a thread running the function
- Thread handle returned via **pthread_t** structure
  - Specify **NULL** to use default attributes
- Single argument sent to function
  - If no arguments to function, specify **NULL**
- Check error codes!

EAGAIN - insufficient resources to create thread
EINVAL - invalid attribute

Example I: Thread Creation

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

void *hello (void * arg) {
            printf(“Hello Thread\n”);
            return NULL;
}

main() {
  pthread_t tid;
  pthread_create(&tid, NULL, hello, NULL);
}
```

Possible outcomes:

- ❑ **Message “Hello Thread” is printed on screen**
- ❑ **Nothing printed on screen. This outcome is more likely that previous. Main thread is the process and when the process ends, all threads are cancelled, too. Thus, if the pthread_create call returns before the O/S has had the time to set up the thread and begin execution, the thread will die a premature death when the process ends.**

Waiting for a Thread

int pthread_join(tid, val_ptr);

pthread_t tid

handle of *joinable* thread

void **val_ptr

exit value returned by joined thread

- Calling thread waits for thread with handle **tid** to terminate
  - Only one thread can be joined
  - Thread must be *joinable*
- Exit value is returned from joined thread
  - Type returned is **(void *)**
  - Use **NULL** if no return value expected

ESRCH - thread (pthread_t) not found
EINVAL - thread (pthread_t) not joinable

- **This is the better way to have one thread wait for the completion of another thread.**
- **Pthread_join will block until the thread associated with the pthread_t handle has terminated. The second parameter returns a pointer to a value from the thread being joined. This value can be "sent" from the joined thread by use of return or pthread_exit(). The type of the returned value is (void *) since this is the return type of the function that was used in the pthread_create call.**
- **pthread_join() can be used to wait for one thread to terminate. There is no single function that can join multiple threads.**

Thread States

- Pthreads threads have two states
  - *joinable* and *detached*
- Threads are joinable by default
  - Resources are kept until **pthread_join**
  - Can be reset with attributes or API call
- Detached threads cannot be joined
  - Resources can be reclaimed at termination
  - Cannot reset to be *joinable*

- **Pthread_join detaches the thread automatically, so resources can be reclaimed at that time. This would be why threads can only be joined once during the execution.**
- **Once a thread is detached, whether by attributes or API call, that thread cannot be set to be joinable.**

Example II: Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
       printf(“Hello Thread\n”);
}

main() {
  pthread_t tid[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS;
i++)
    pthread_create(&tid[i], NULL,
hello, NULL);

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

Better example of waiting for threads, in this case, multiple threads doing the same function. Notice that there must be one call for each thread needed to be “joined” after termination. Also, the joins are done in the order of the thread’s creation. Thus, if the last thread created is the first to finish, it will not be joined until the previous threads have finished.

What’s Wrong?

Problem is passing *address* of “i”; value of “i” is changing and will likely be different when thread is allowed to run than when pthread_create was called.

What is printed for myNum?

```
void *threadFunc(void *pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( “Thread number %d\n”, myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
   pthread_create(&tid[i], NULL,
threadFunc, &i);
}
```

Time	**thread0**	**thread1**	**thread2**	**index**	
0:	**created**			**0**	
1:		**created**		**1**	
2:	**get *arg**			**2**	**// value of *arg is 2**
Etc.					

Solution – "Local" Storage

```
void *threadFunc(void *pArg)
{
  int myNum = *((int*)pArg);
  printf( "Thread number %d\n", myNum);
}
. . .

// from main():
for (int i = 0; i < numThreads; i++) {
  tNum[i] = i;
  pthread_create(&tid[i], NULL, threadFunc,
&tNum[i]);
}
```

Solve the problem of passing *address* of "i" by saving current value of "i" in location that will not change. Be sure each thread gets pointer to unique element of tNum array.

Pthreads Mutex Variables

- Simple, flexible, and efficient
- Enables correct programming structures for avoiding race conditions
- New data types to declare objects
  - **pthread_mutex_t**
    - the mutex variable
  - **pthread_mutexattr_t**
    - mutex attributes
- Mutex must first be initialized before it can be used.

Mutex can only be "held" by one thread at a time.

Mutexes can be shared between processes, but only if the Pthreads system supports the functionality.

pthread_mutex_init

int pthread_mutex_init(mutex, attr);

pthread_mutex_t *mutex

mutex to be initialized

const pthread_mutexattr_t *attr

attributes to be given to mutex

ENOMEM - insufficient memory for mutex
EAGAIN - insufficient resources (other than memory)
EPERM - no privilege to perform operation

Can also use the static, default initializer: **PTHREAD_MUTEX_INITIALIZER**

```
pthread_mutex_t mtx1 = PTHREAD_MUTEX_INITIALIZER;
```

which uses default attributes

pthread_mutex_lock

int pthread_mutex_lock(mutex);

pthread_mutex_t *mutex

mutex to attempt to lock

- Attempts to lock mutex
  - If mutex is locked by another thread, calling thread is blocked
- Mutex is held by calling thread until unlocked
  - Mutex lock/unlock must be paired or deadlock occurs

EINVAL - mutex is invalid
EDEADLK - calling thread already owns mutex

pthread_mutex_unlock

int pthread_mutex_unlock(mutex);

pthread_mutex_t *mutex

mutex to be unlocked

EINVAL - mutex is invalid
EPERM - calling thread does not own mutex

Example III: Use of mutex

```
#define NUMTHREADS 4
pthread_mutex_t gMutex; // why does this have to be global?
int g_sum = 0;

void *threadFunc(void *arg)
{
  int mySum = bigComputation();
  pthread_mutex_lock( &gMutex );
    g_sum += mySum;             // threads access one at a time
  pthread_mutex_unlock( &gMutex );
}

int main() {
  pthread_t hThread[NUMTHREADS];

  pthread_mutex_init( &gMutex, NULL );
  for (int i = 0; i < NUMTHREADS; i++)
    pthread_create(&hThread[i],NULL,threadFunc,NULL);

  for (int i = 0; i < NUMTHREADS; i++)
    pthread_join(hThread[i]);
  printf ("Global sum = %f\n", g_sum);
}
```

(Point out features and function calls of example code)

Q: Why not just put bigComputation() into critical region?

A: Thread would exclude all other threads from running their own, independent calls to bigComputation. This would make the code serial.

Condition Variables

- Semaphores are conditional on the semaphore count
- Condition variable is associated with an arbitrary conditional
  - Same operations: wait and signal
- Provides mutual exclusion

- **This box is meant to call out the difference between a semaphore and a condition variable. The semaphore is conditioned on the value of the semaphore (zero or non-zero), while condition variables can be triggered on any arbitrary condition the programmer cares to write.**
- **Mutual exclusion is provided by having threads wait on the condition variable until signaled. Once signaled and woken up, if the conditional expression evaluates correctly, a thread will proceed; otherwise, the thread should be directed to return to waiting on the condition variable.**

Lost and Spurious Signals

- Signal to condition variable is not saved
  - If no thread waiting, signal is "lost"
  - Thread can be deadlocked waiting for signal that will not be sent
- Condition variable can (rarely) receive spurious signals
  - Slowed execution from predictable signals
  - Need to retest conditional expression

- **Two problems that can arise from using condition variables. Both of these are taken care of when using the algorithm on the next slide.**
- **Lost signal – condition has no memory, thus, if no thread is waiting on the condition variable, all signals on that condition variable will do nothing. If a thread "blindly" waits on a condition variable, it can be deadlocked if there is no other signal to wake it up. (Thus, the conditional expression is checked before a thread will wait.)**
- **Spurious wakeups – on some multi-processor systems, condition variable code could be slowed down if all signals were to be made predictable. (Thus, the correct algorithm requires a retest of the conditional expression after a thread is signaled after waiting on the condition variable. Spurious wakeups should put the thread back to waiting on the condition variable.)**

Condition Variable and Mutex

- Mutex is associated with condition variable
  - Protects evaluation of the conditional expression
  - Prevents race condition between signaling thread and threads waiting on condition variable

- **Condition variables are always paired with a mutex. The mutex is used to protect access any variables that are used in the conditional expression. This access will be in testing the conditional and updating variables that are involved in the conditional test. In other words, the mutex must protect the code from race conditions between a thread signaling the condition variable with a thread waiting on the condition variable.**
- **The only job of the mutex should be to protect variables used in the conditional. This will ensure proper utilization of condition variables and prevent lock contention performance problems when a mutex is overloaded by being used in other parts of the code.**

Condition Variable Algorithm

Avoids problems with lost and spurious signals

Negation of condition needed to proceed

Mutex is automatically released when thread waits

```
acquire mutex;
while (conditional is true)
   wait on condition variable;
perform critical region computation;
update conditional;
signal sleeping thread(s);
release mutex;
```

May be optional

- Conditional in while test is the negation of the condition needed to proceed in to the critical region. For example, if (x > 0) is needed to get past condition variable, test will be while (x<= 0). This is the most important part of the algorithm presented. The while test prevents…
- LOST SIGNALS since the conditional is tested before the thread waits. If the condition is false (able to proceed), the thread will not wait
- SPURIOUS WAKEUP since the thread will retest the while condition. If the condition is still true (not able to proceed), the thread will go back to waiting. [This is not prevented, but handled properly so that the code works as expected even when spurious wakeups occur.]
- As will be seen shortly, the mutex is released when the thread waits on the condition variable.
- The "update" and "signal" steps can be done external to the algorithm, dependent upon the requirements of the application. However, programmer must be sure the variables involved are updated while protected by the mutex associated with the condition variable.

Condition Variables

pthread_cond_init, pthread_cond_destroy

initialize/destroy condition variable

pthread_cond_wait

thread goes to sleep until signal of condition variable

pthread_cond_signal

signal release of condition variable

pthread_cond_broadcast

broadcast release of condition variable

Condition Variable Types

Data types used

pthread_cond_t

the condition variable

pthread_condattr_t

condition variable attributes

Before use, condition variable (and mutex) must be initialized

- **The Pthread condition variable object.**
- **New data types used to declare objects. Condition variable must first be initialized before it can be used.**
- **Condition is used to have threads wait until some condition has been met.**
- **Condition variables can be shared between processes (in shared memory), but only if the Pthreads implementation supports the functionality.**

pthread_cond_init

int pthread_cond_init(cond, attr);

pthread_cond_t *cond

condition variable to be initialized

const pthread_condattr_t *attr

attributes to be given to condition variable

ENOMEM - insufficient memory for condition variable
EAGAIN - insufficient resources (other than memory)
EBUSY - condition variable already intialized
EINVAL - attr is invalid

Can also use the static, default initializer: **PTHREAD_COND_INITIALIZER**

```
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
```

which uses default attributes

pthread_cond_wait

int pthread_cond_wait(cond, mutex);

pthread_cond_t *cond

condition variable to wait on

pthread_mutex_t *mutex

mutex to be unlocked

- Thread put to "sleep" waiting for signal on **cond**
- Mutex is unlocked
  - Allows other threads to acquire lock
  - When signal arrives, mutex will be reacquired before **pthread_cond_wait** returns

EINVAL - cond or mutex is invalid
EINVAL - different mutex for concurrent waits
EINVAL - calling thread does not own mutex

- **The calling thread will block (sleep) until such time as a pthread_cond_signal is issued that wakes up the threads. Upon going to sleep, the mutex (held because the thread has the mutex via the standard algorithm already presented) is unlocked. This will allow other threads that may be wanting to wait on the condition variable to enter the wait algorithm and for threads that need to update the variables used in the conditional expression the chance to lock the mutex to make changes.**
- **Before the pthread_cond_wait function returns (after the thread receives a signal), the mutex will be automatically reacquired (locked). This gives the thread mutually exclusive access to the conditional expression variables (if needed) and why the standard algorithm releases the lock when done.**

pthread_cond_signal

int pthread_cond_signal(cond);

pthread_cond_t *cond

condition variable to be signaled

- Signal condition variable, wake one waiting thread
- If no threads waiting, no action taken
  - Signal is not saved for future threads
- Signaling thread need not have mutex
  - May be more efficient
  - Problem may occur if thread priorities used

EINVAL - cond is invalid

If the signaling thread does not hold the associated mutex, the problem when using thread priorities would develop if a high priority thread is waiting and lower priority thread might lock the mutex (at the start of the condition variable algorithm) before the higher priority thread got the chance to reawaken and lock the mutex. By holding the mutex when signaling, in this situation, the lower priority thread will block in the attempt to lock the mutex and the higher priority thread will be given preference to acquire the mutex when it is released by the signaling thread.

pthread_cond_broadcast

int pthread_cond_broadcast(cond);

pthread_cond_t *cond

condition variable to signal

- Wake all threads waiting on condition variable
- If no threads waiting, no action taken
  - Broadcast is not saved for future threads
- Signaling thread need not have mutex

EINVAL - cond is invalid

Each thread waiting on the condition variable will be signaled and, in turn, as the mutex becomes available, return from the pthread_cond_wait call.

Summary

- Parallel computing on shared memory
  - Single address space: Any memory location can be accessible by any of the processors or cores within a processor.
  - The basic concept of threads
- POSIX
  - Thread creation
  - Thread synchronization using mutex
  - Thread signaling using condition variables