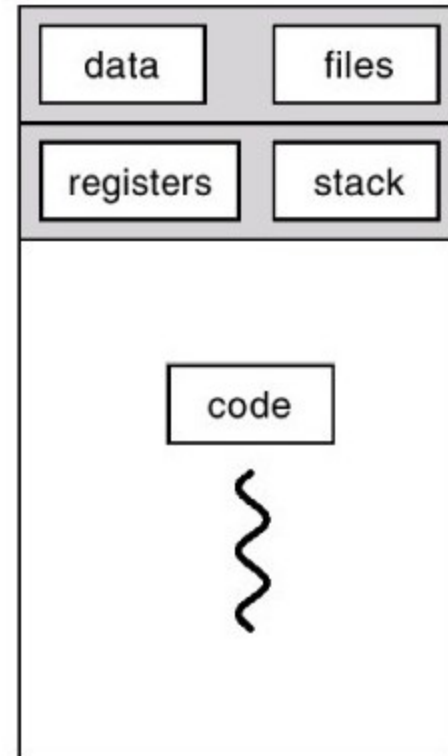# Threads
(light weight processes)

Chester Rebeiro

IIT Madras

Processes

- Separate streams of execution
- Each process isolated from the other
- Process state contains
  - Process ID
  - Environment
  - Working directory.
  - Program instructions
  - Registers
  - Stack
  - Heap
  - File descriptors
- Created by the OS using fork
  - Significant overheads

Threads

- Separate streams of execution within a single process

- Threads in a process <u>not</u> isolated from each other

- Each thread state (thread control block) contains
  - Registers (including EIP, ESP)
  - stack
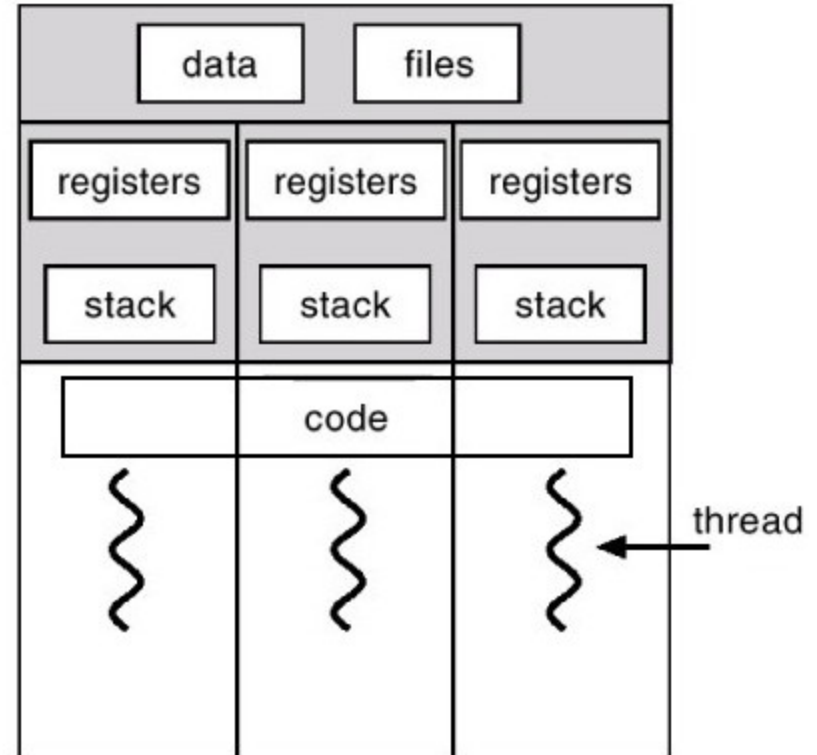
Why threads?

- Lightweight

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Cost of creating 50,000 processes / threads
(https://computing.llnl.gov/tutorials/pthreads/)

- Efficient communication between entities
- Efficient context switching

4

Threads vs Processes

- A thread has no data segment or heap

- A thread cannot live on its own. It needs to be attached to a process

- There can be more than one thread in a process. Each thread has its own stack

- If a thread dies, its stack is reclaimed

- A process has code, heap, stack, other segments

- A process has at-least one thread.

- Threads within a process share the same I/O, code, files.

- If a process dies, all threads die.

Based on Junfeng Yang's lecture slides
http://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l08-thread.pdf

pthread library

- Create a thread in a process

  int pthread_create(pthread_t *thread,

  const pthread_attr_t *attr,

  void *(*start_routine) (void *),

  void *arg);

Thread identifier (TID) much like

Pointer to a function, which starts execution in a different thread

Arguments to the function

- Destroying a thread

  void pthread_exit(void *retval);

Exit value of the thread

pthread library contd.

- Join : Wait for a specific thread to complete

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

TID of the thread to wait for

Exit status of the thread

what is the difference with wait()?

Example

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

void *thread_fn(void *arg){
  long id = (long) arg;
  printf("Starting thread %ld\n", id);
  sleep(5);
  printf("Exiting thread %ld\n", id);
  return NULL;
}

int main(){
  pthread_t t1, t2;

  pthread_create(&t1, NULL, thread_fn, (void *)1);
  pthread_create(&t2, NULL, thread_fn, (void *)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Exiting main\n");
  return 0;
}
```

How many threads are there in this program? 3

Note. You need to link the pthread library

```
chester@optiplex:~/shared/OS/programs$ gcc threads.c -lpthread
chester@optiplex:~/shared/OS/programs$ ./a.out
Starting thread 1
Starting thread 2
Exiting thread 1
Exiting thread 2
Exiting main
```

Other thread libraries

- Windows threads
- Boost (in C++)
- LinuxThreads
- etc.

Who manages threads?

- Two strategies
  - User threads
    - Thread management done by user level thread library. Kernel knows nothing about the threads.
  - Kernel threads
    - Threads directly supported by the kernel.
    - Known as light weight processes.

User level threads

- ## Advantages:
  - Fast (really lightweight)

    (no system call to manage threads. The thread library does everything).
  - Can be implemented on an OS that does not support threading.
  - Switching is fast. No, switch from user to protected mode.

- ## Disadvantages:
  - Scheduling can be an issue. (Consider, one thread that is blocked on an IO and another runnable.)
  - Lack of coordination between kernel and threads. (A process with 1000 threads competes for a timeslice with a process having just 1 thread.)
 - Requires non-blocking system calls. (If one thread invokes a system call, all threads need to wait)
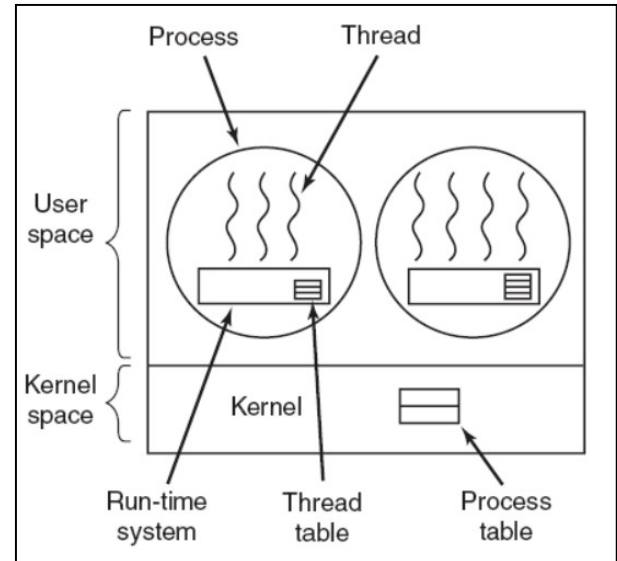
Kernel level threads

- ## Advantages:
  - Scheduler can decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel-level threads are especially good for applications that frequently block.

- ## Disadvantages:
  - The kernel-level threads are slow (they involve kernel invocations.)
  - Overheads in the kernel. (Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads.)
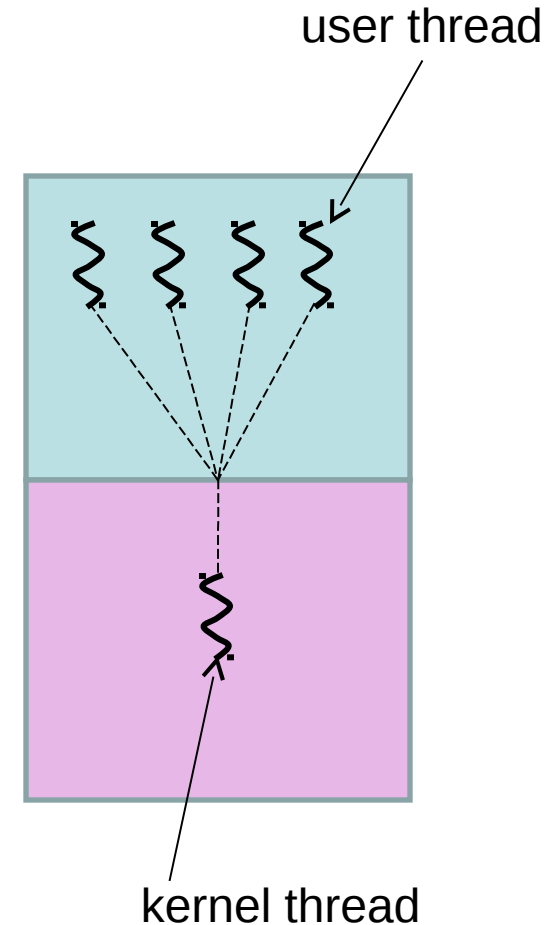
Process   Thread

Kernel

Process table   Thread table
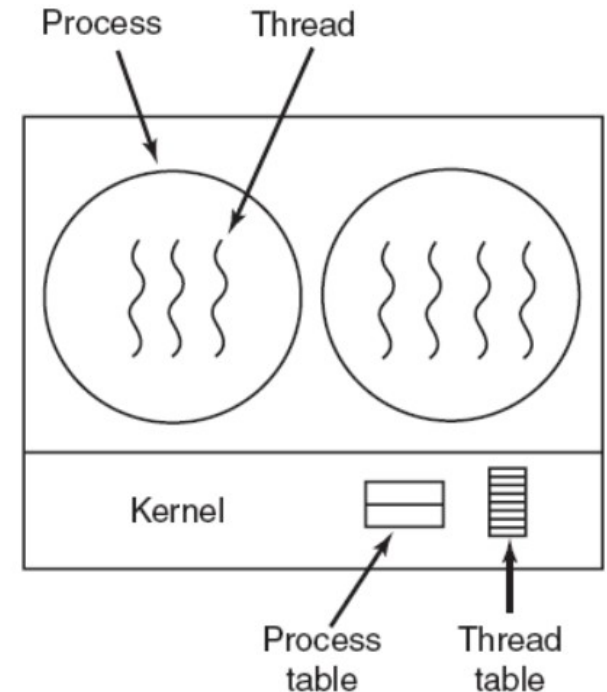
Thread Models

- Many-to-one model

- one-to-one model

- Many-to-many model

Many-to-one model

- Many user level threads map to a single kernel thread

- Pros:
  - Fast. No system calls to manage threads.
  - No mode change for switching threads

- Cons:
  - No parallel execution of threads. All threads block when one has a system call.
 - Not suited for multi-processor systems.

user thread

kernel thread

One-to-one model

- Each user thread associated with one kernel thread.
- Pros.
  - Better suited for multiprocessor environments.
  - When one thread blocks, the other threads can continue to execute.
- Cons.
  - Expensive. Kernel is involved.

user thread
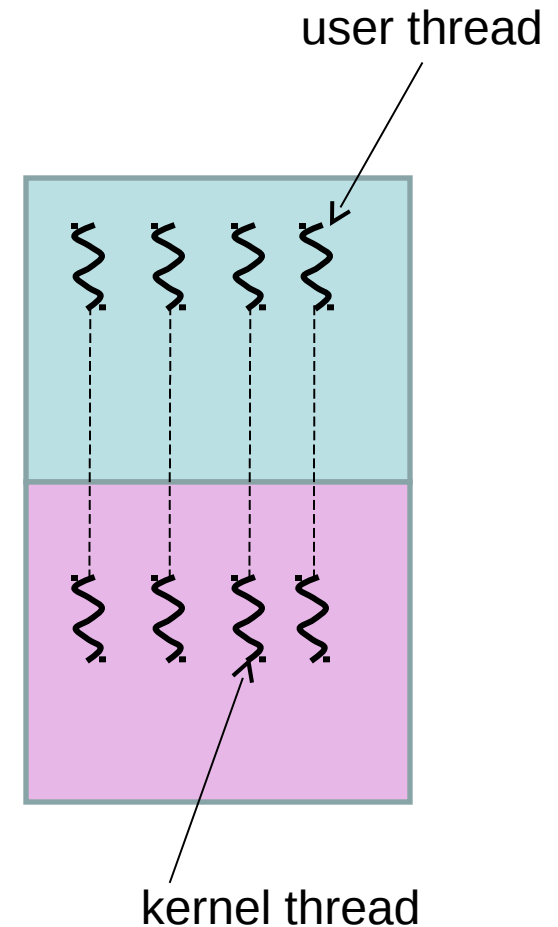
kernel thread

Many-to-Many model
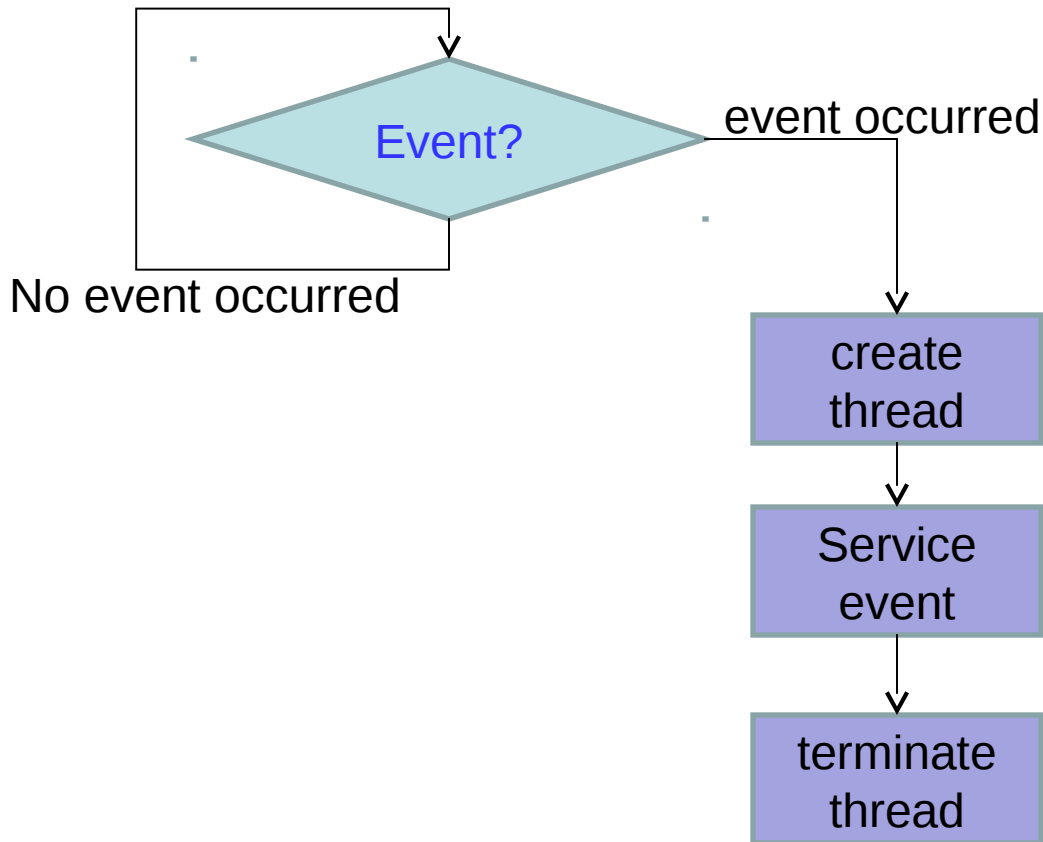
- Many user threads mapped to many kernel threads
  - Supported by some unix and windows versions
- Pros: flexible
  - OS creates kernel threads as required
  - Process creates user threads as needed
- Cons: Complex
  - Double management
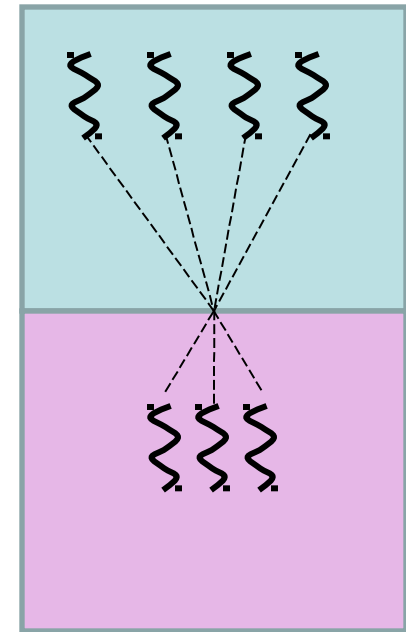
Threading issues

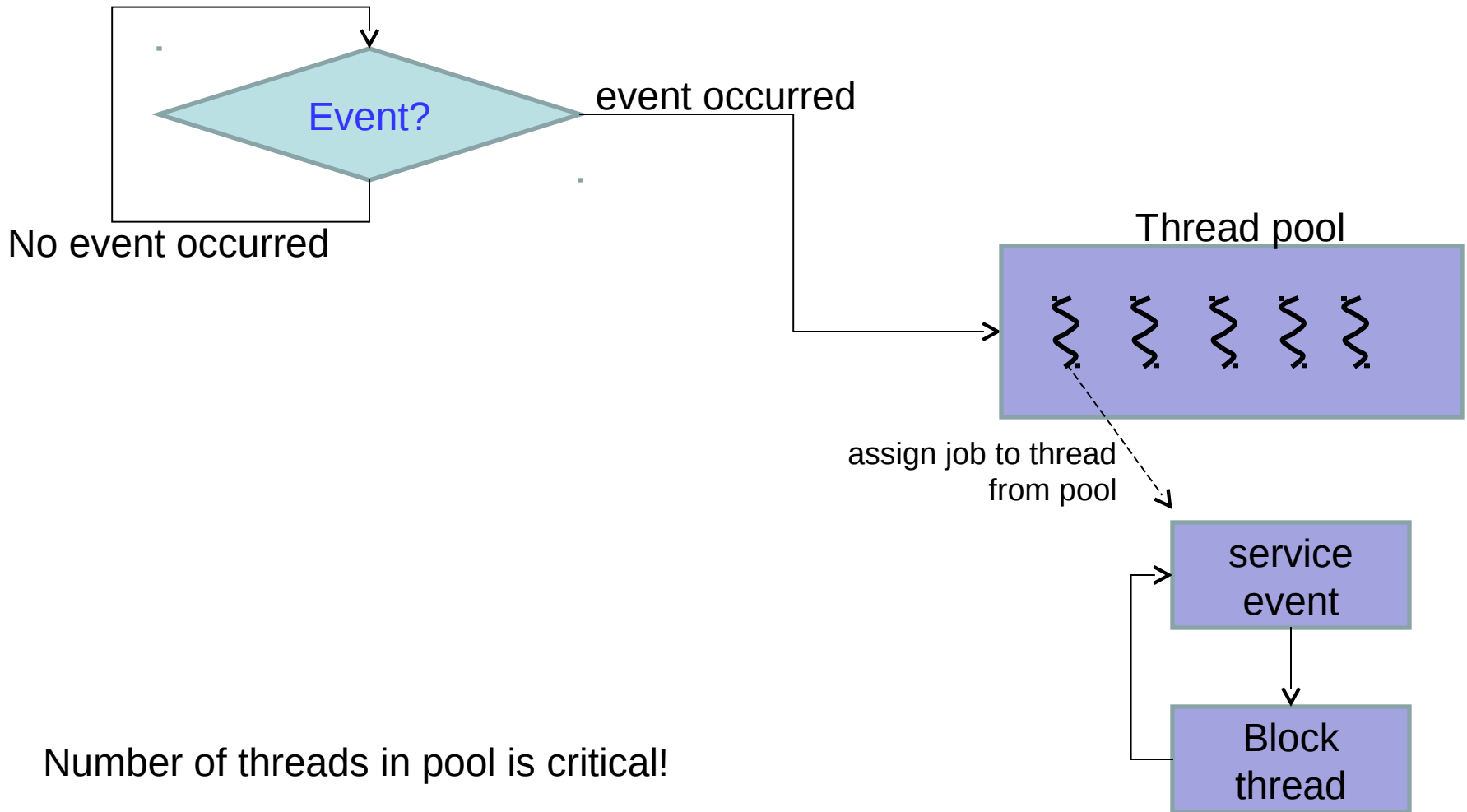- What happens when a thread invokes fork?
  - Duplicate all threads?
    - Not easily done… other threads may be running or blocked in a system call
  - Duplicate only the caller thread?
    - More feasible.
- Segmentation fault in a thread. Should only the thread terminate or the entire process?

Typical usage of threads

Event?

event occurred

No event occurred

create thread

Service event

terminate thread

Creating and terminating thread lead to overheads

Thread pools

Event?

event occurred

No event occurred

Thread pool

assign job to thread
from pool

service
event

Block
thread

Number of threads in pool is critical!

Assignment

- Write a multi-threaded program to perform matrix multiplication.

  - The input matrices are of dimension 1000 x 1000

  - Tabulate time taken vs number of threads used.