Imperial College London

Operating Systems

Threads

Imperial College London

Processes recap

- One of the oldest abstractions in computing
  - An instance of a program being executed, a running program

- Allows a single processor to run multiple programs "simultaneously"
  - Processes turn a single CPU into multiple virtual CPUs
  - Each process runs on a virtual CPU

Imperial College London

Why Have Processes?

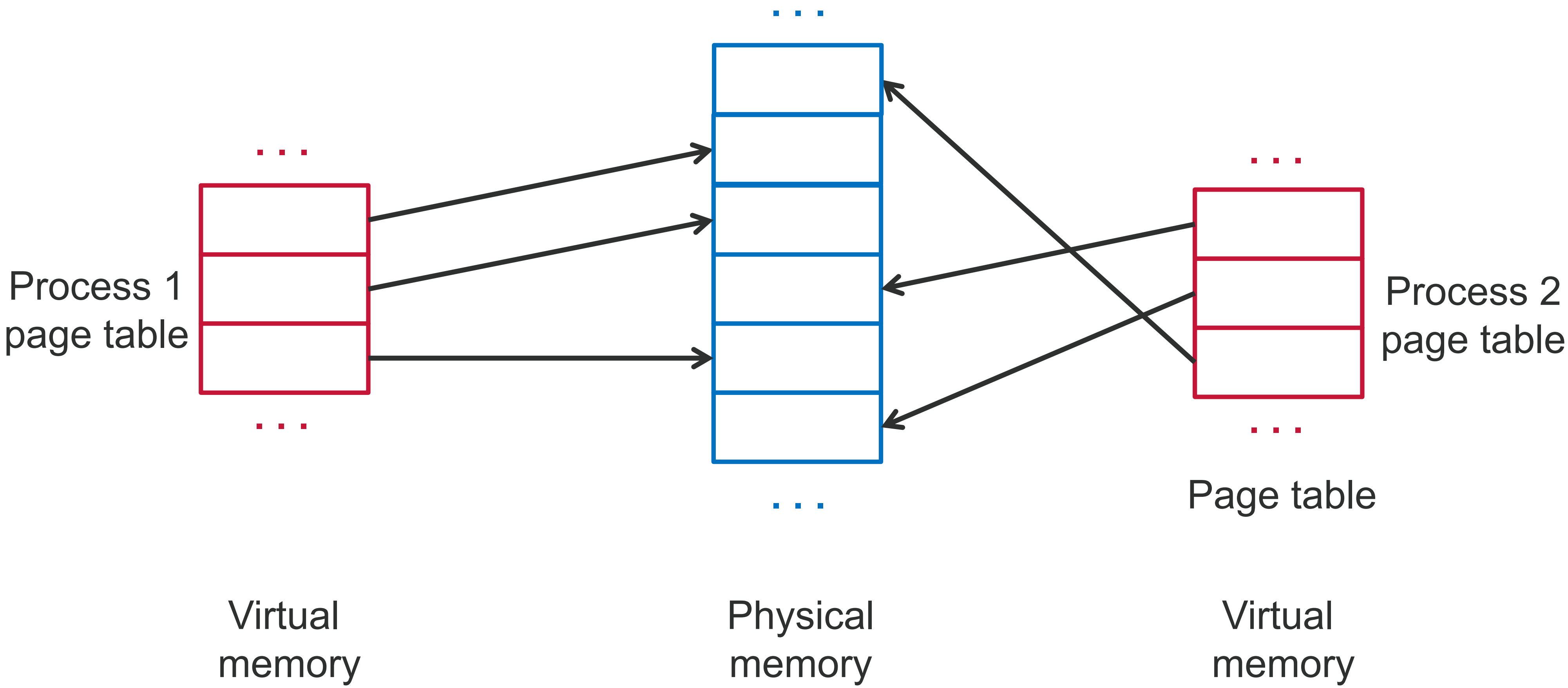
- Provide (the illusion of) concurrency
  - Real vs. apparent concurrency
- Provide isolation
  - Each process has its own address space
- Simplicity of programming
  - Firefox doesn’t need to worry about gcc
- Allow better utilisation of machine resources
  - Different processes require different resources at a certain time

Imperial College London

Remember optimizing fork()

- fork(): Creates a new child process by making an exact copy of the parent process image
- Copying the entire address space is expensive!
  - And very few memory pages are going to end up having different values in the two processes

Page tables: high level view



Schematic view: refer to “Memory management” lectures for details

Imperial College London

Copy On Write (COW)

- Give child its own page table pointing to parent's pages which are marked as *read only*
- When any process writes to a page
  - Protection fault causes trap to the kernel
  - Kernel allocates new copy of page so that both processes have their own private copies
  - Both copies are marked *read-write*

Imperial College London

Dirty Cow Exploit

- Difference between a clean abstraction and a buggy implementation
- http://thehackernews.com/2016/10/linux-kernel-exploit.html

Operating Systems

Threads

What Are Threads?

- Execution streams that share *the same address space*
- When multithreading is used, each process can contain one or more threads

Per process items	Per thread items
Address space	Program counter (PC)
Global variables	Registers
Open files	Stack
Child processes	
Signals	

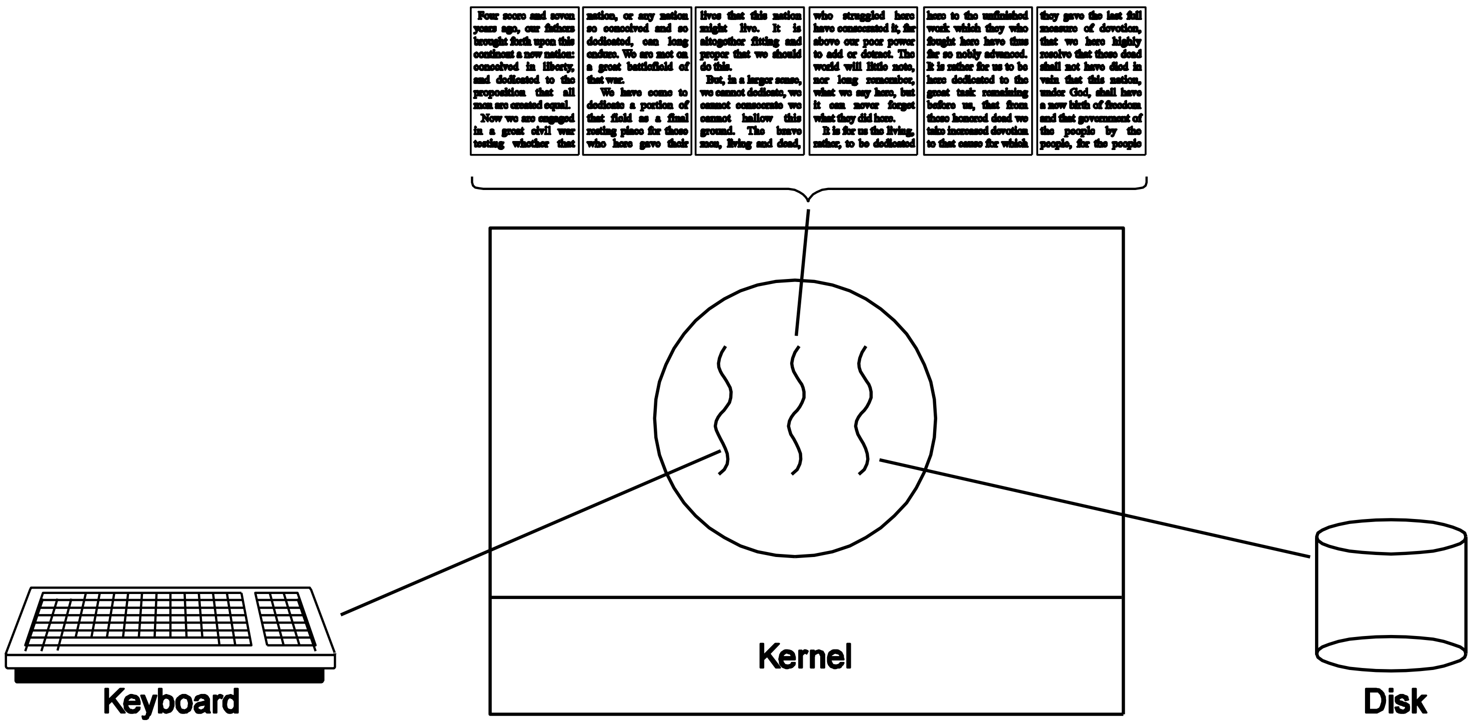
Imperial College London

Why Threads?

- Many applications contain multiple activities
  - Which execute in parallel
  - Which access and process the same data
  - Some of which might block

Example

Processing thread
- processes input buffer
- writes result into output buffer



© Pearson Prentice Hall

Input thread
- reads data into buffer

Output thread
- writes output buffer to disk

Why Not Processes?

- Many applications contain multiple activities
  - Which execute in parallel
  - Which access and process the same data
  - Some of which might block

- Processes are too heavyweight
  - Difficult to communicate between different address spaces
  - An activity that blocks might switch out the entire application
  - Expensive to context switch between activities
  - Expensive to create/destroy activities

Threads – Problems/Concerns

- Shared address space
  - Memory corruption
    - One thread can write another thread's stack
  - Concurrency bugs
    - Concurrent access to shared data (e.g., global variables)
- Forking
  - What happens on a `fork()`?
    - Create a new process with the same number of threads
    - Create a new process with a single thread?
- Signals
  - When a signal arrives, which thread should handle it?

Case Study: PThreads

PThreads (Posix Threads)

- Defined by IEEE standard 1003.1c
  - Implemented by most UNIX systems

```
#include <pthread.h>
#include <sys/types.h>

pthread_t        →  type representing a thread
pthread_attr_t   →  type representing the attributes of a thread
```

Creating Threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

- Creates a new thread
  - The newly created thread is stored in `*thread`
  - The function returns 0 if thread was successfully created, or error code
- Arguments:
  - `attr` -> specifies thread attributes, can be `NULL` for default attributes
    - Attributes include: minimum stack size, guard size, detached/ joinable, etc.
  - `start_routine` -> the C function the thread will start execute once created
  - `arg` -> The argument to be passed to start_routine (of pointer type `void*`). Can be `NULL` if no arguments are to be passed.

Terminating Threads

```
void pthread_exit(void *value_ptr);
```

- Terminates the thread and makes **`value_ptr`** available to any successful join with the terminating thread
- Called implicitly when the thread's start routine returns
  - But not for the initial thread which started main()
  - If **`main()`** terminates before other threads w/o calling **`pthread_exit()`**, the entire process is terminated
  - If **`pthread_exit()`** is called in **`main()`** the process continues executing until the last thread terminates (or **`exit()`** is called)

PThread Example (1)

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

void *thread_work(void *threadid) {
  long id = (long) threadid;
  printf("Thread %ld\n", id);
}

int main (int argc, char *argv[]) {
  pthread_t threads[5];
  long t;
  for (t=0; t<5; t++)
      pthread_create(&threads[t], NULL,
                     thread_work, (void *)t);

}
```

```
$ gcc pt.c -lpthread
$ ./a.out
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
```

Imperial College London

Passing Arguments to Threads

- What if we want to pass more than one argument to the start routine?
  - Create a structure containing the arguments and pass a pointer to that structure to `pthread_create()`

Yielding the CPU

```
int pthread_yield(void)
```

- Releases the CPU to let another thread run
- Returns 0 on success, or an error code
    - Always succeeds on Linux

- Why would a thread ever yield? (compare with nice() for processes)

Joining Other Threads

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- Blocks until `thread` terminates
- The value passed to `pthread_exit()` by the terminating thread is available in the location referenced by `value_ptr`
  - `value_ptr` **can be** `NULL`

Join Example

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

long a, b, c;
void *work1(void *x) { a = (long)x * (long)x;}
void *work2(void *y) { b = (long)y * (long)y;}

int main (int argc, char *argv[]) {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, work1, (void*) 3);
  pthread_create(&t2, NULL, work2, (void*) 4);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  c = a + b;
  printf("3^2 + 4^2 = %ld\n", c);
}
```

```
$ ./a.out
3^2 + 4^2 = 25
```

Imperial College London

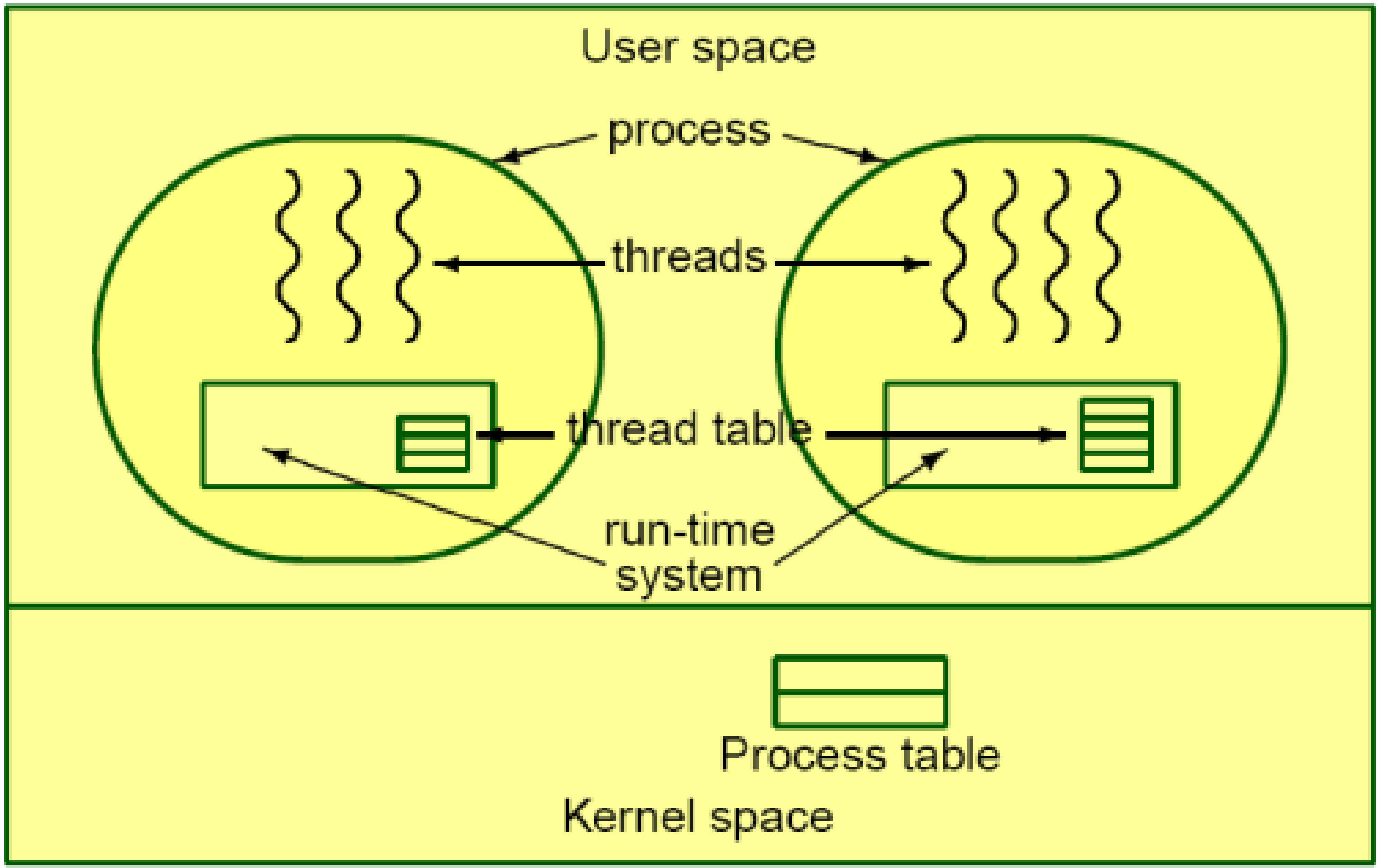
Two Ways to Implement Threads

- User-level threads
  - The kernel is not aware of threads
  - Each process manages its own threads
- Kernel-level threads
  - Managed by the kernel
- Trade-offs on each side
- Various hybrid approaches possible

User-Level Threads

- Kernel thinks it is managing processes only
- Threads implemented by software library
- Process maintains a thread table and does thread scheduling

User-Level Threads



© Pearson Prentice Hall

Advantages of User-Level Threads

- Better performance
  - Thread creation and termination are fast
  - Thread switching is fast
  - Thread synchronization (e.g., joining other threads) is fast
  - All these operations do not require any kernel activity

- Allows application-specific run-times
  - Each application can have its own scheduling algorithm

Disadvantages of User-Level Threads

- Blocking system calls stops *all threads* in the process
  - Denies one of the core motivations for using threads
- Non-blocking I/O can be used (e.g., `select()`)
  - Harder to use and understand, inelegant

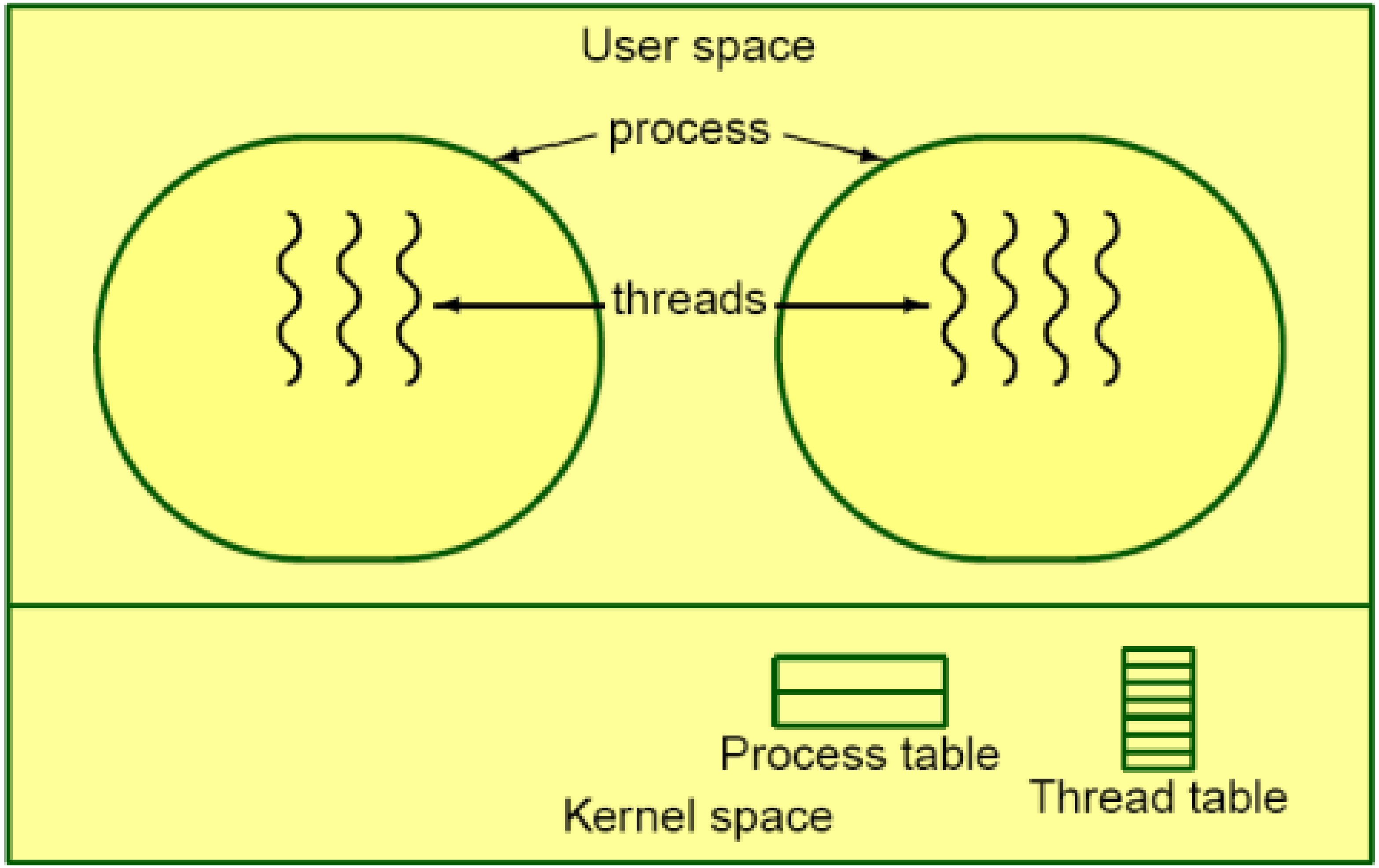
- During a page fault the OS blocks the whole process…
  - But other threads might be runnable

- Difficult to implement preemptive scheduling
  - Run-time can request a clock interrupt
    - Messy to program
    - High-frequency clock interrupts not always available
    - Individual threads may also need to use a clock interrupt

Tutorial question

- In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server, running on a single-CPU machine. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. For this problem, assume that thread switching time is negligible. How many requests/sec can the server handle if it is single-threaded?

Kernel Threads



© Pearson Prentice Hall

Advantages of Kernel Threads

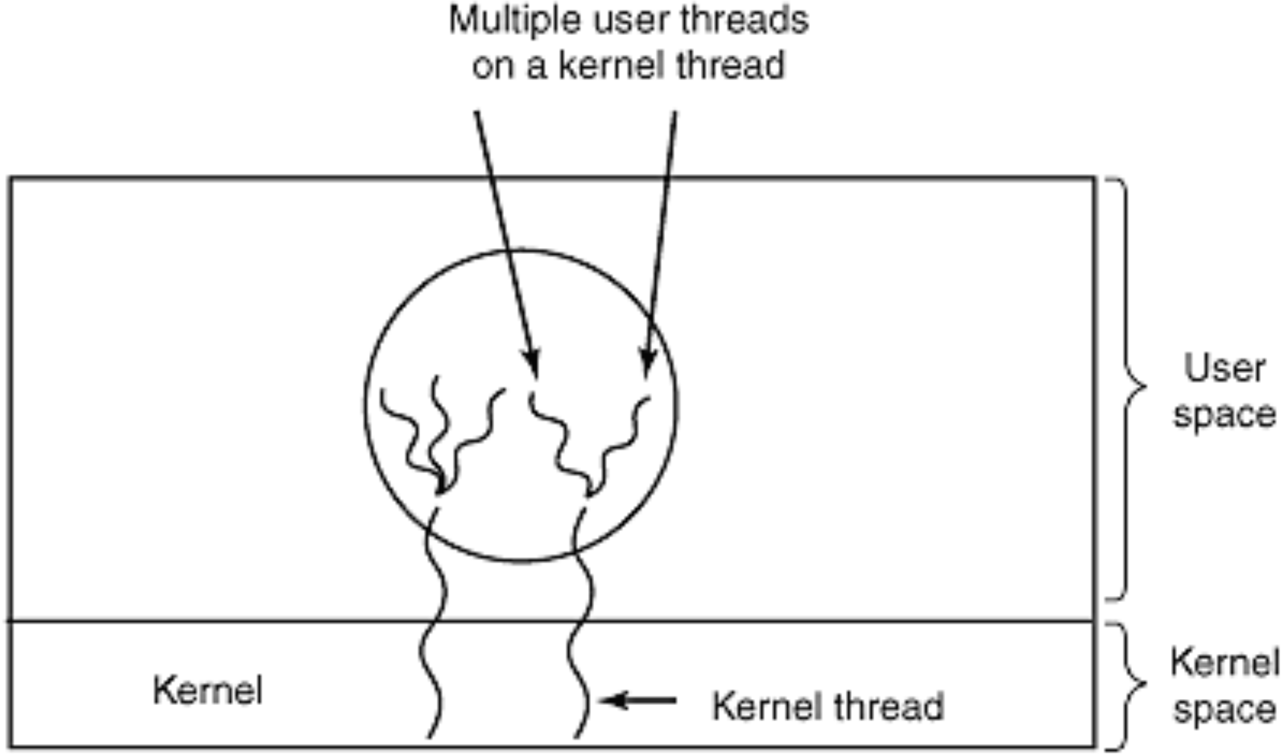
- Blocking system calls/page faults can be easily accommodated
    - If one thread calls a blocking system call or causes a page fault, the kernel can schedule a runnable thread from the same process

Disadvantages of Kernel Threads

- Thread creation and termination more expensive
  - Require kernel call
  - But still much cheaper than process creation/termination
  - One mitigation strategy is to recycle threads (*thread pools*)
- Thread synchronisation more expensive
  - Requires blocking system calls
- Thread switching is more expensive
  - Requires kernel call
  - But still much cheaper than process switches
    - Same address space
- No application-specific scheduler

Hybrid Approaches

- Use kernel threads and multiplex user-level threads onto some (or all) kernel threads



© Pearson Prentice Hall

Imperial College London

Tutorial

- If in a multithreaded web server the only way to read from a file is the normal blocking read() system call, do you think user-level threads or kernel-level threads are being used? Why?