Operating Systems

Dr. Shu Yin

Part II: Process Management

- Processes
- Threads
- Process Synchronization
- CPU Scheduling
- Deadlocks

Goals

- Processes
- Threads

Thread

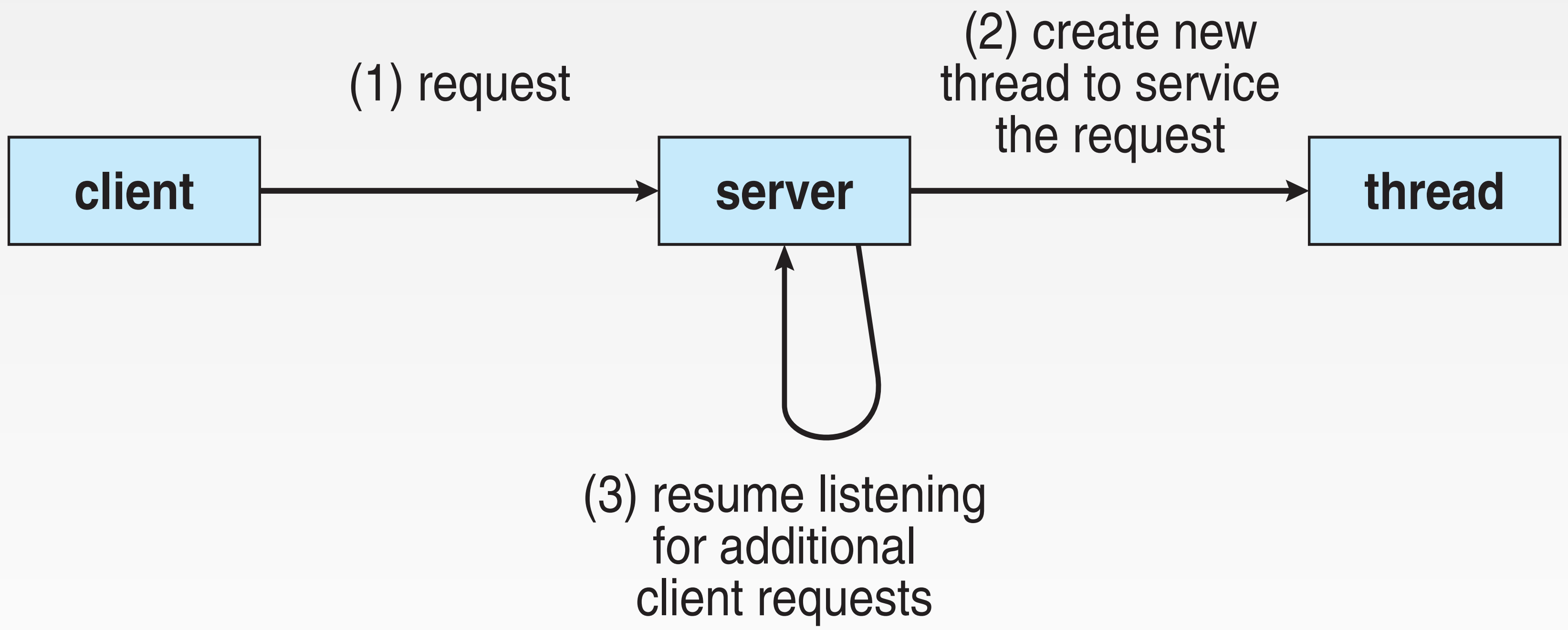
- A fundamental unit of CPU utilization that forms the basis of multithreaded computer systems

Motivation

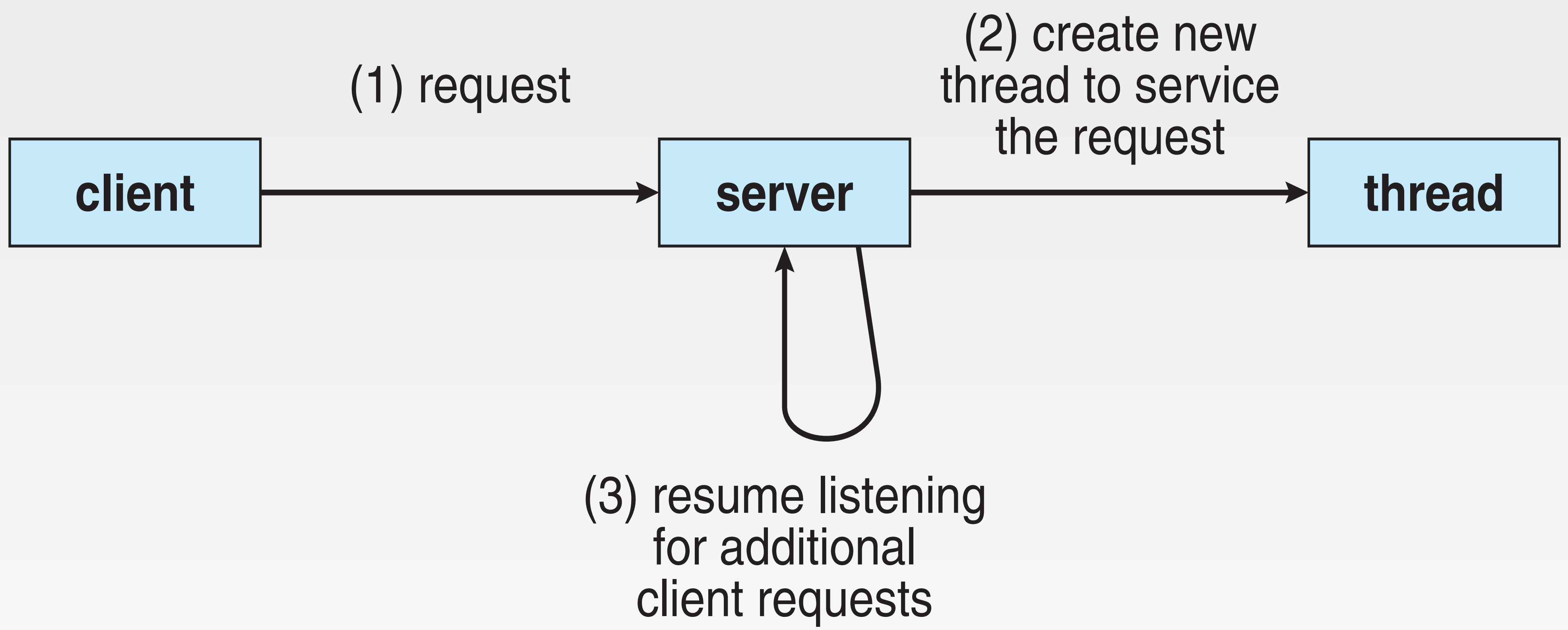
- Most modern apps are multithreaded
- Threads run within application
- Multiple tasks with the app can be implemented by separate threads
  - update display
  - fetch data
  - spell checking
  - answer a network request

Motivation (cont.)

- Can simplify code, increase efficiency
- Kernels are generally multithreaded



Multithreaded Server Architecture



Benefit

- Responsiveness
- Resource sharing
- Economy
- Scalability

Multicore programming

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

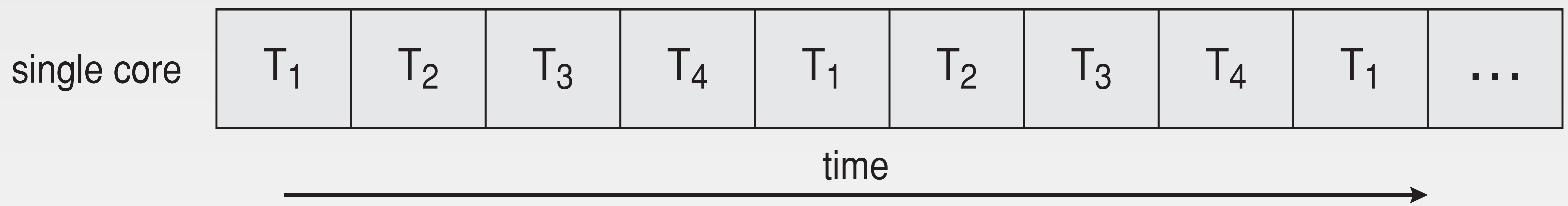
Multicore programming (cont.)

- Parallelism
  - A system can perform more than one task simultaneously
- Concurrency
  - Supports more than one task making progress
    - in single core, scheduler provide concurrency

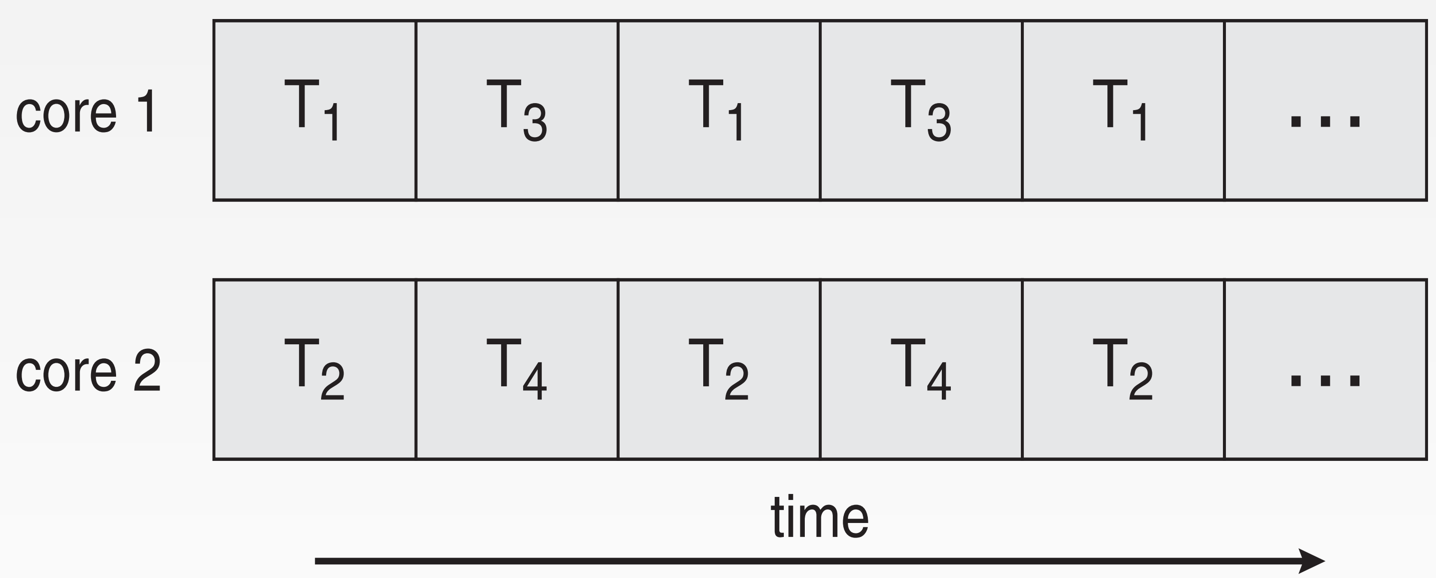
Multicore programming (cont.)

- Types of parallelism
  - Data parallelism
    - subset of same data across multiple cores
    - same operation on each
  - Task parallelism
    - threads across cores, each thread performing unique operation

Recall: Concurrency vs. Parallelism

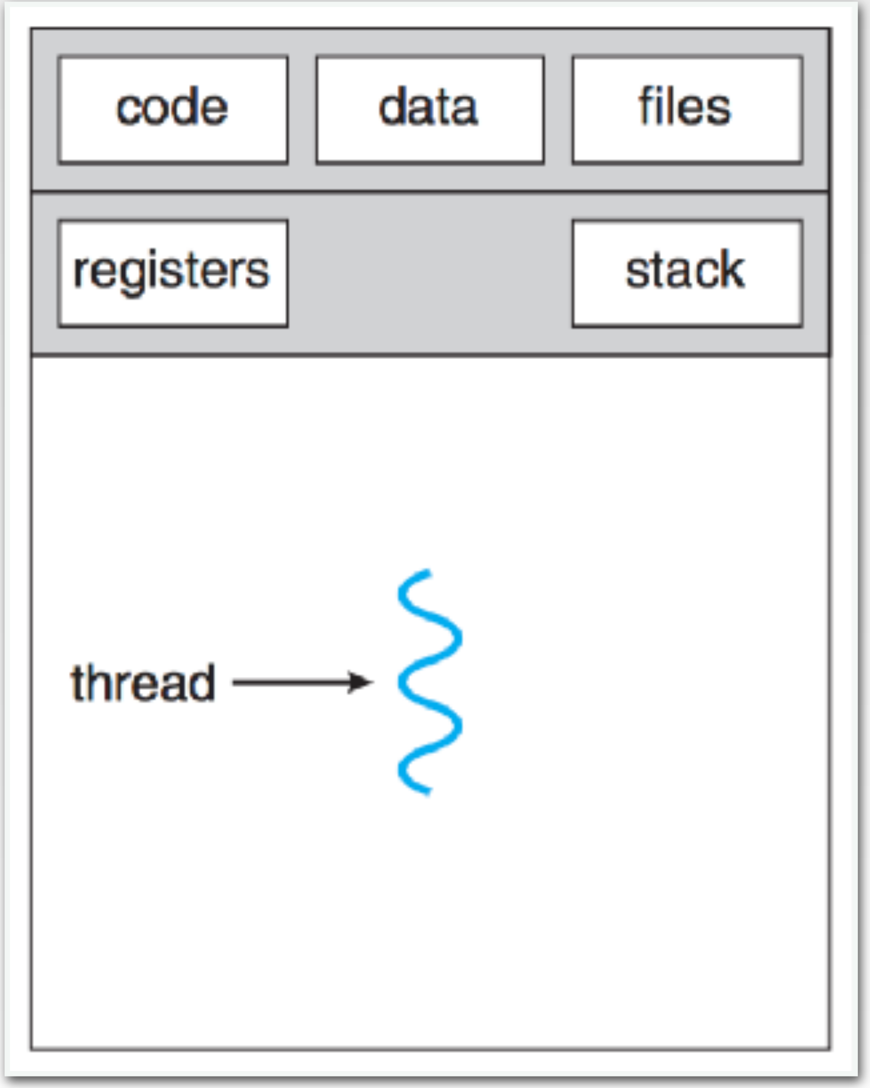


concurrent execution on single-core system



parallelism on a multi-core system

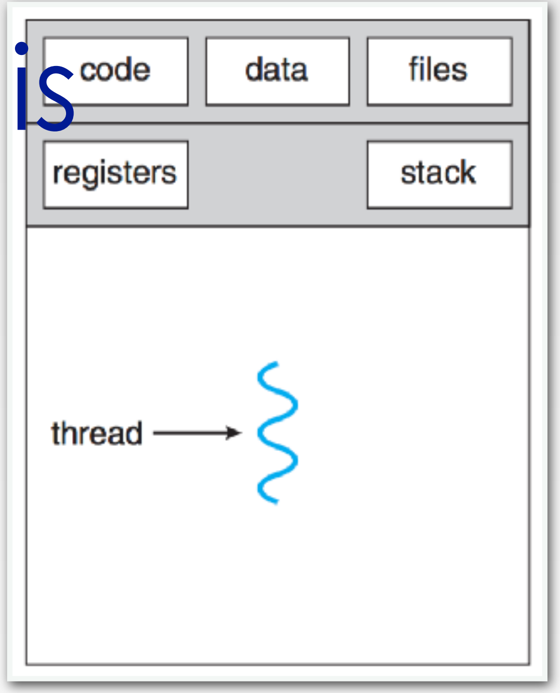
Recall: Traditional UNIX Process



Single-threaded Process

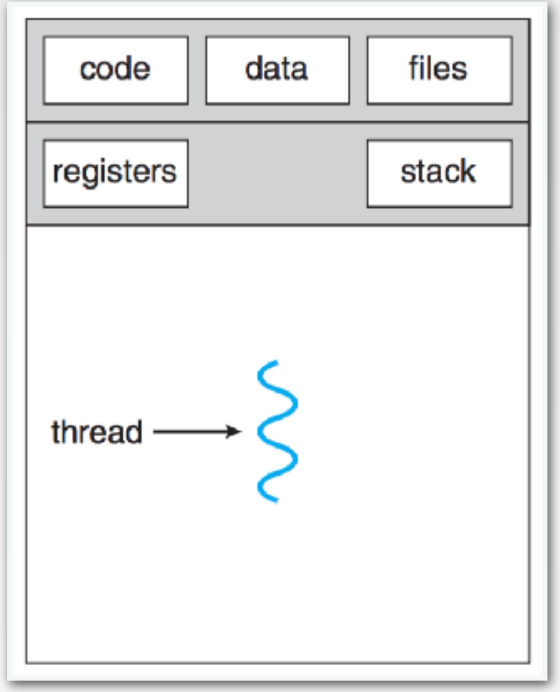
Single Threaded Process (cont.)

- Process: OS abstraction of what is needed to run a single program
  - "Heavyweight Process"
  - No concurrency
- Two Parts:
  - Active Part: Sequential program execution stream
  - Passive Part: Protected resources



上海科技大学 ShanghaiTech University

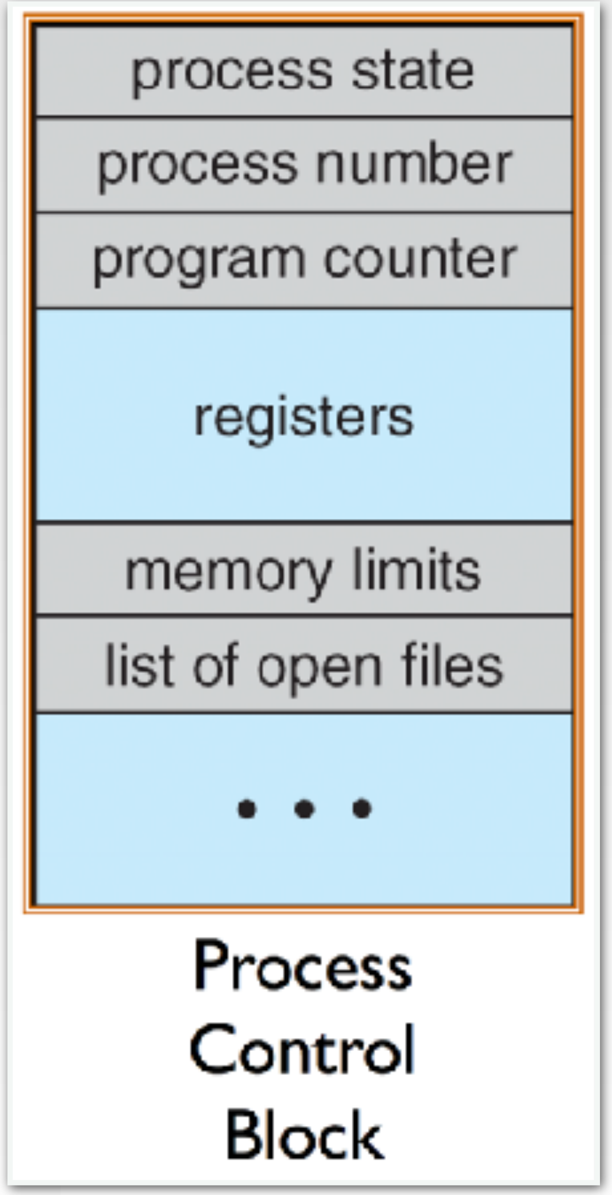
Single Threaded Process (cont.)



- Active Part
  - Code executed as a stream of execution (i.e. thread)
  - Includes state of CPU registers
- Passive Part
  - Main memory state (contents of Address Space)
  - I/O state (i.e. file descriptors)

How do we Multiplex Processes?

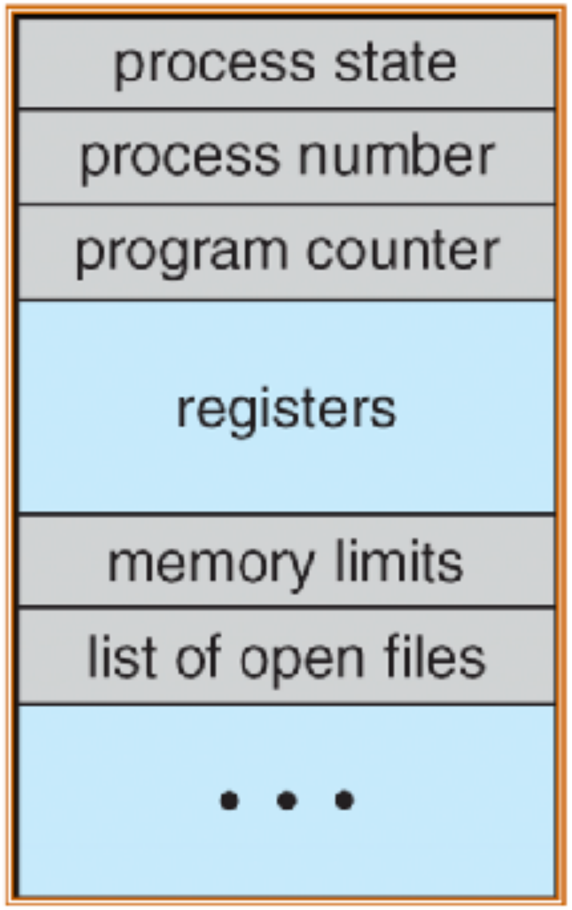
- The current state of process held in a PCB
  - A "snapshot" of the execution and protection environment
  - Only one PCB active at a time



上海科技大学
ShanghaiTech University

Multiplex Processes(cont.)

- Give out CPU time to different processes (Scheduling)
  - Only one process "running" at a time
  - Give more time to important processes

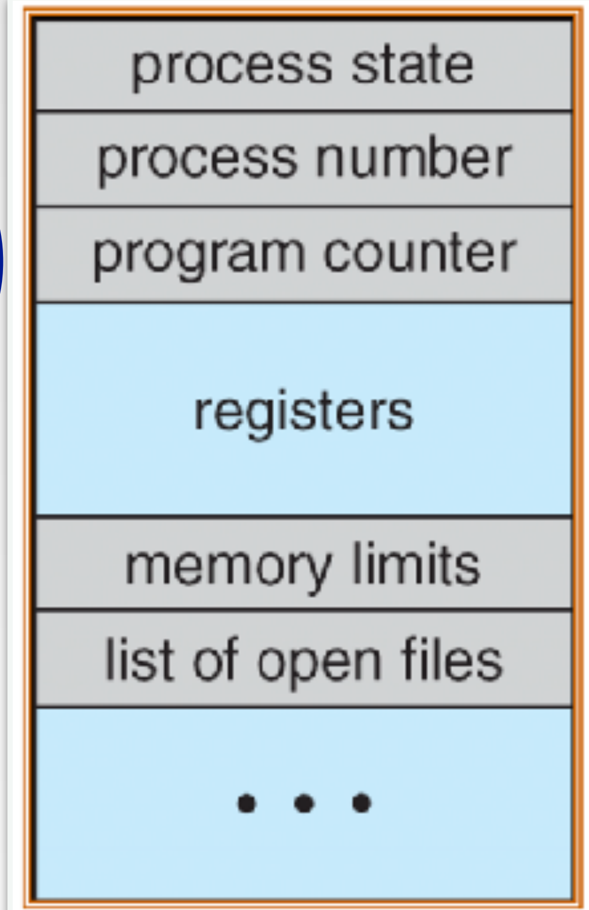


Process Control Block

上海科技大学 ShanghaiTech University

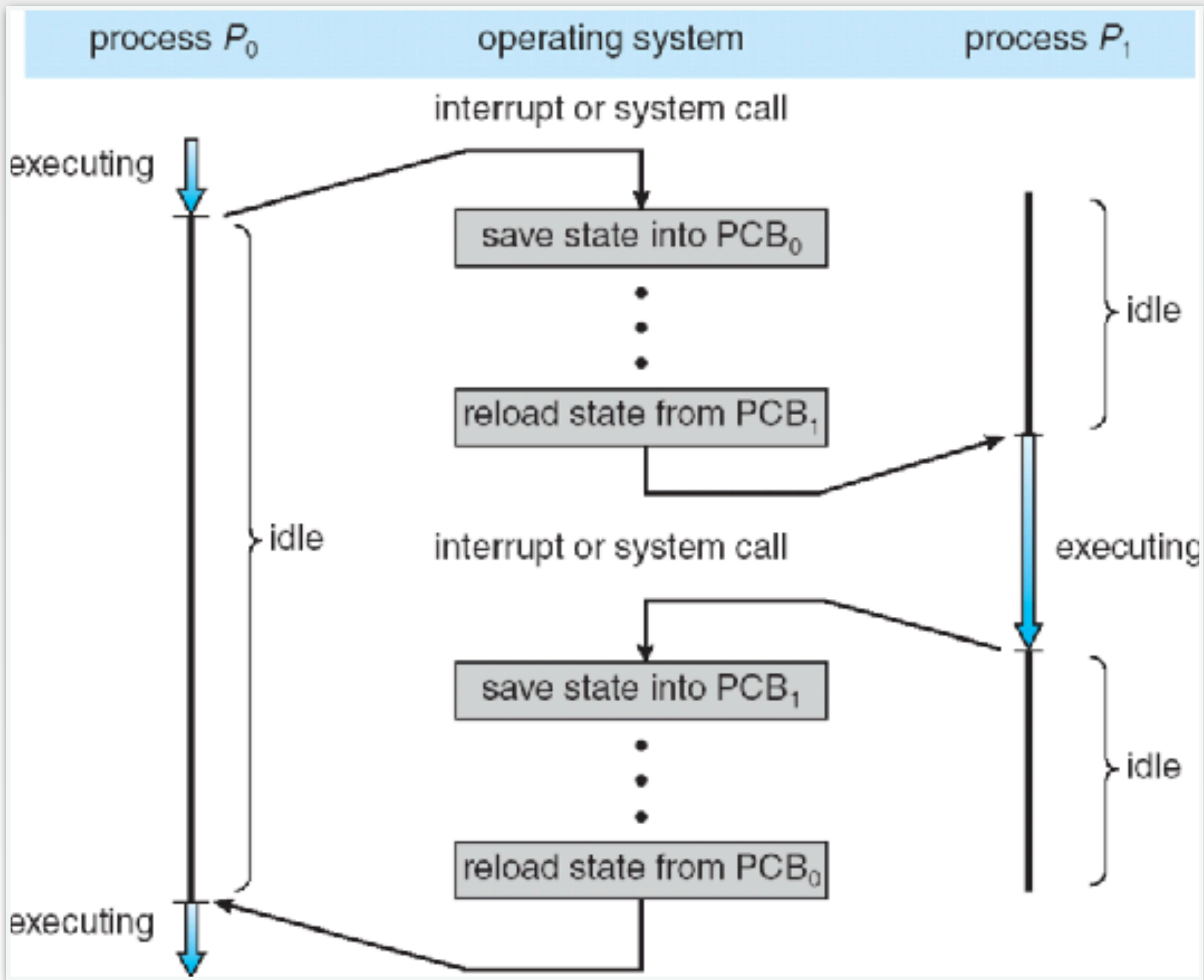
Multiplex Processes(cont.)

- Give pieces of resources to different processes (Protection)
  - Controlled access to non-CPU resources
  - Example mechanisms:
    - Memory Mapping
      - Give each process their own address space
    - Kernel/User duality
      - Arbitrary multiplexing of I/O through system



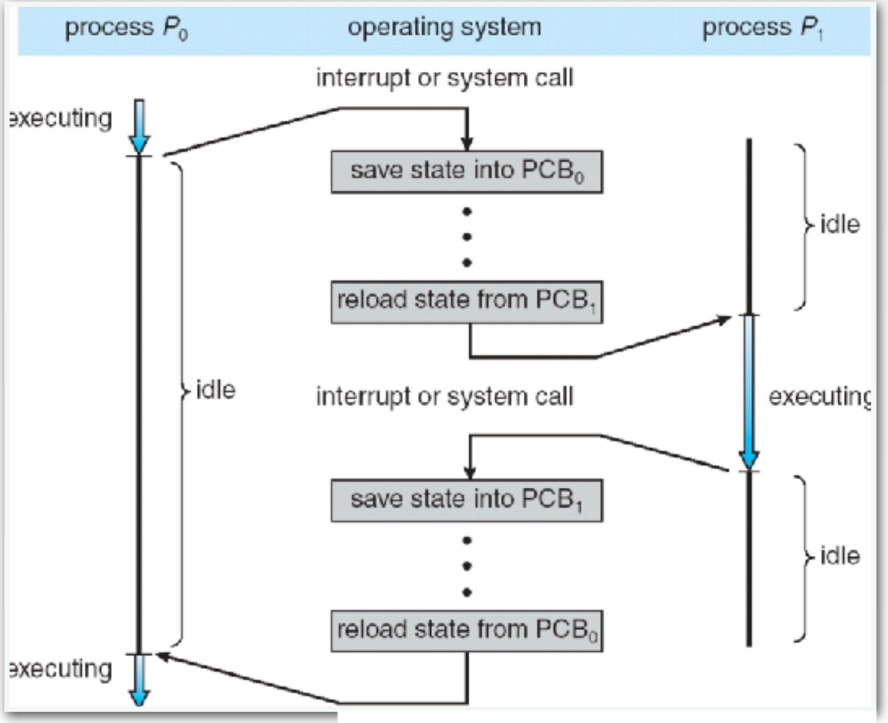
Process Control Block

CPU Switch from Process A to B

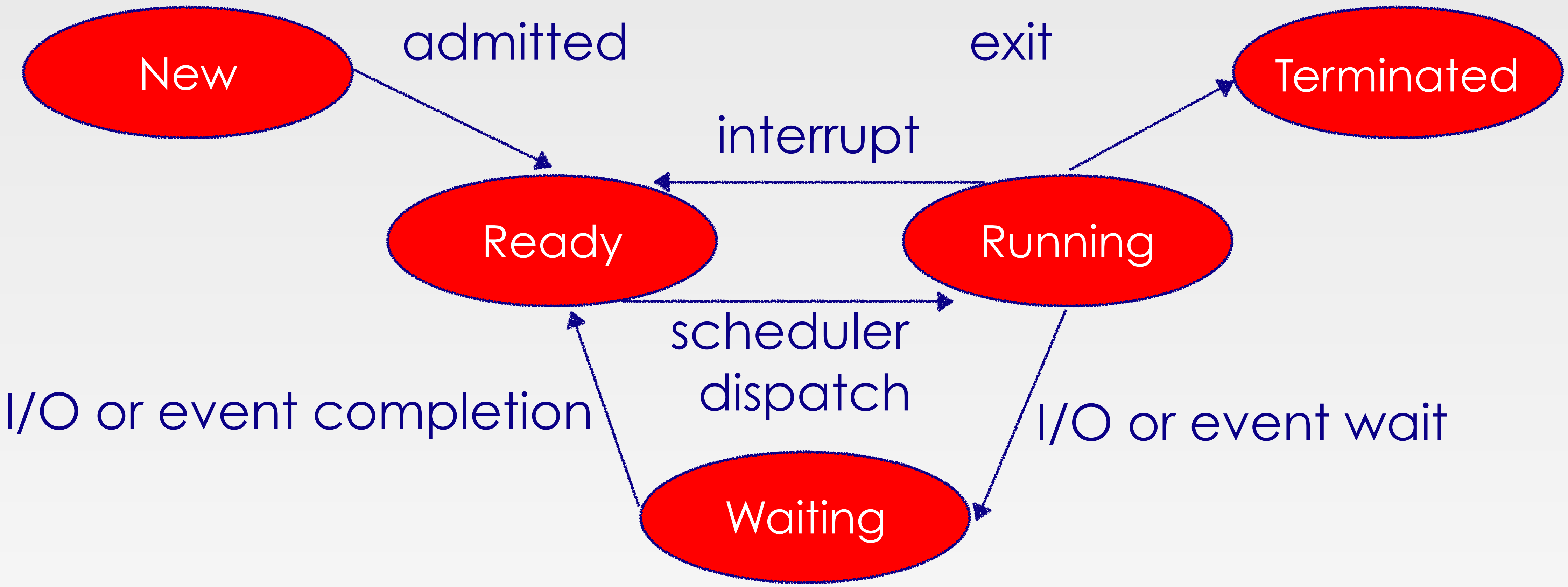


Switch from Process A to B (cont.)

- Context Switch
- Code executed in kernel is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/HT, but contention for resources

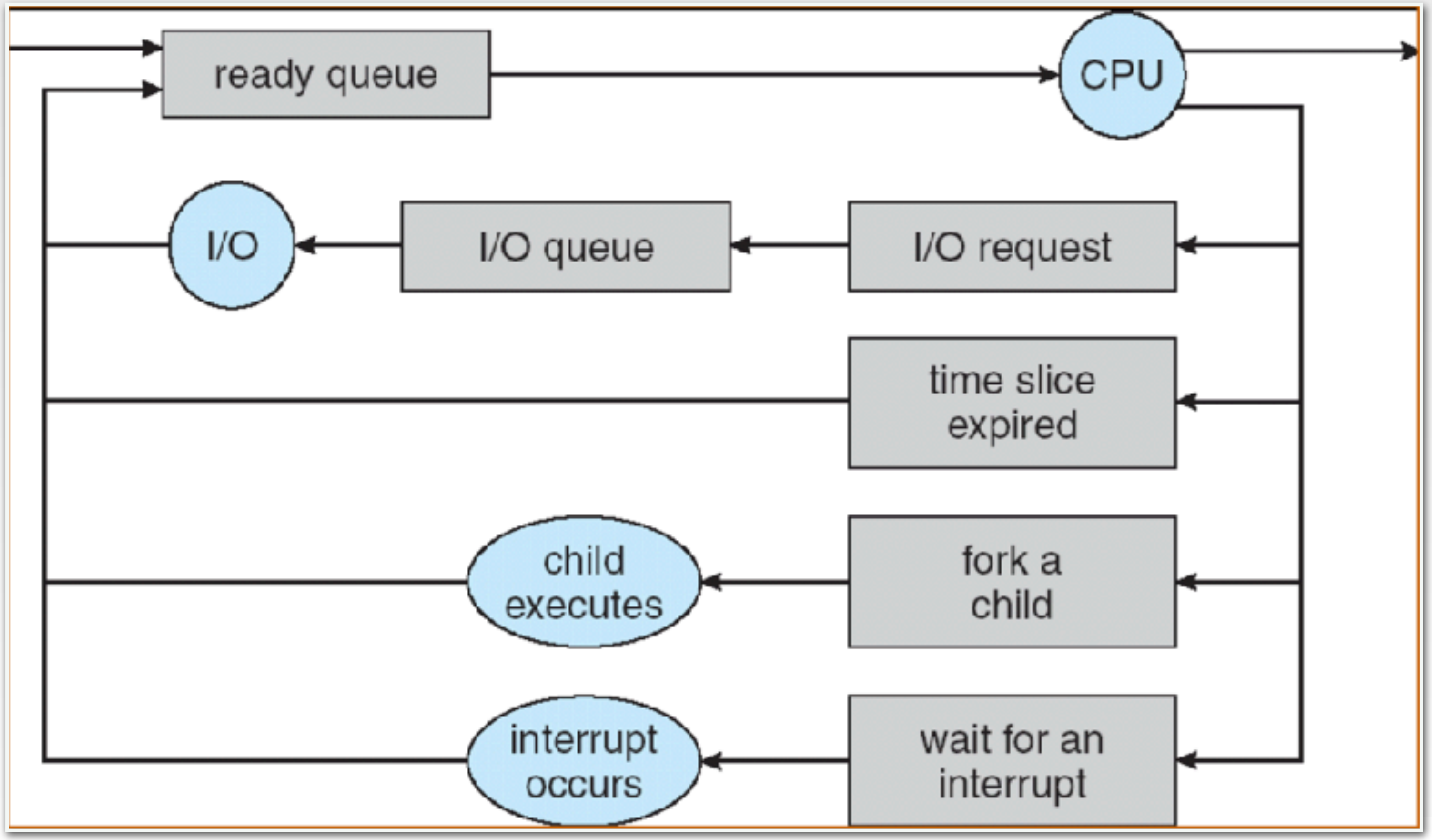


Lifecycle of a Process



- As a process executes, it changes state
  - New, Ready, Running, Waiting, Terminated

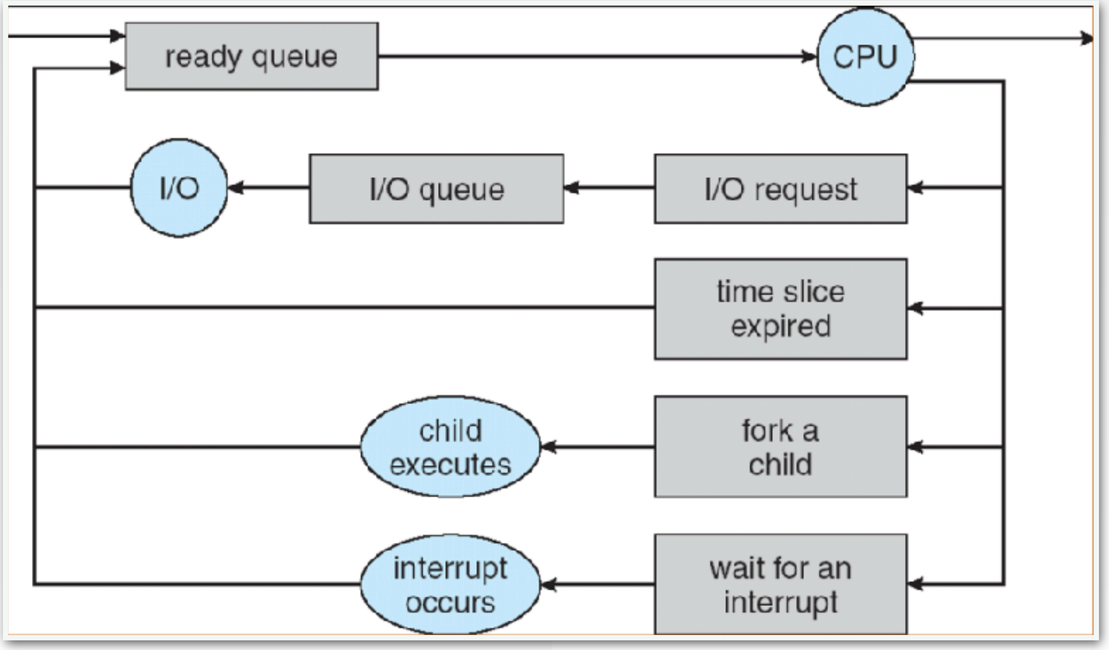
Process Scheduling



上海科技大学
ShanghaiTech University

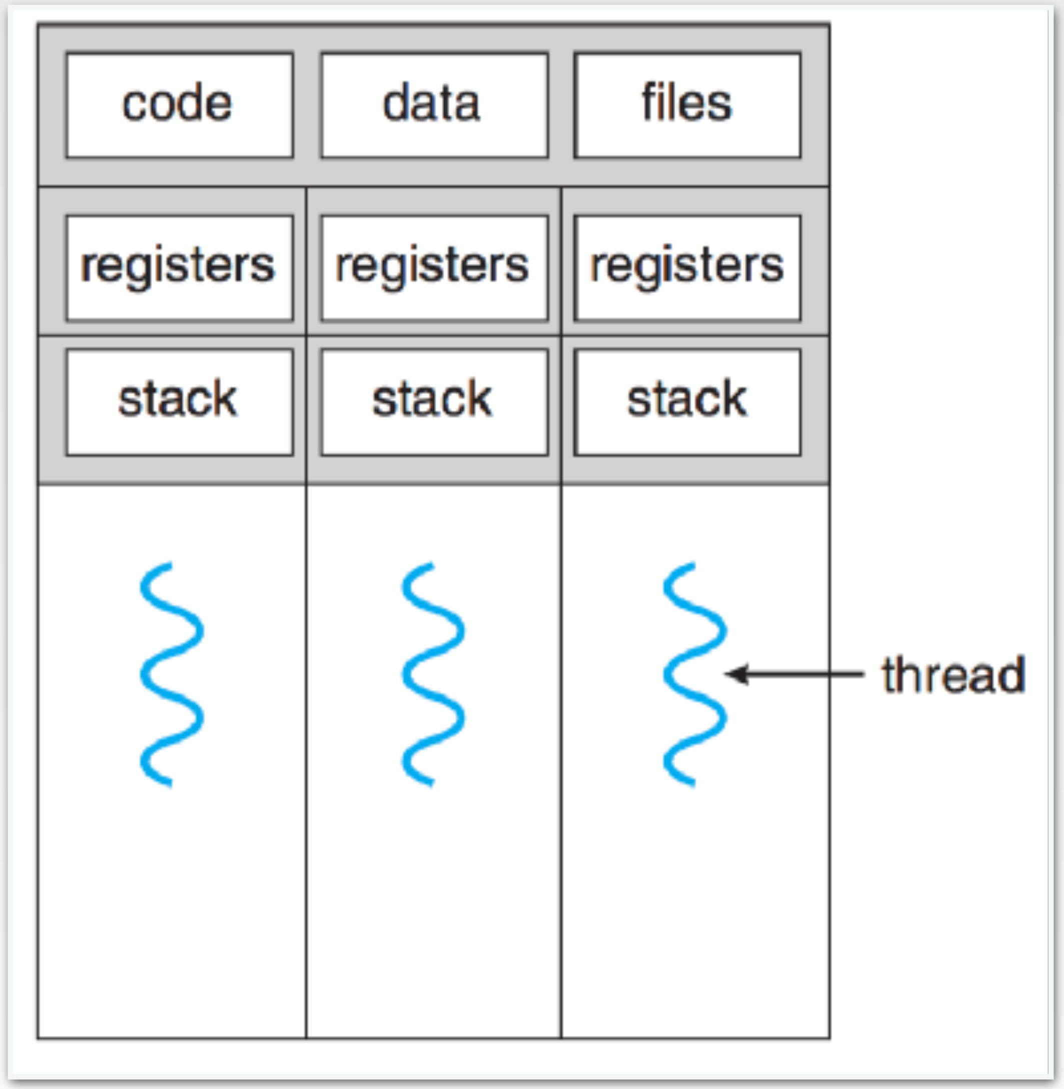
Process Scheduling (cont.)

- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many Algorithms



上海科技大学 ShanghaiTech University

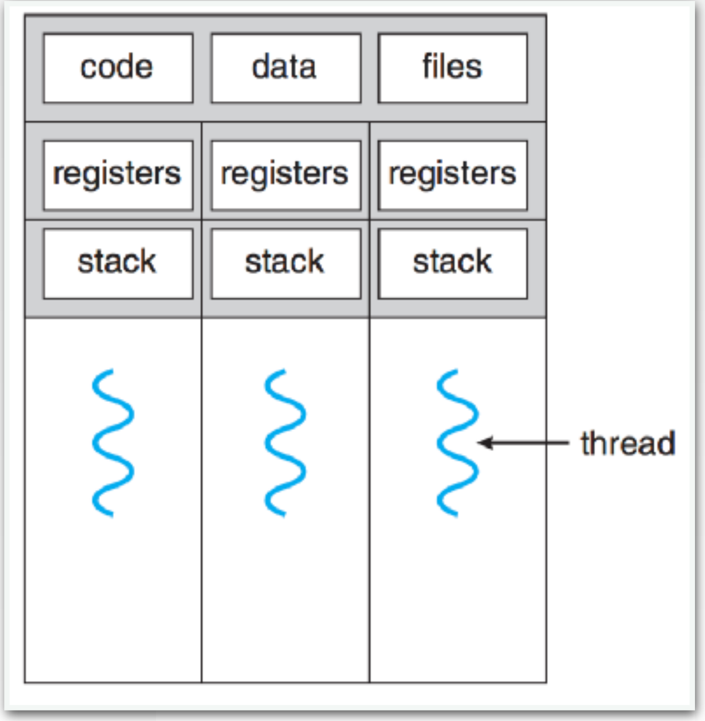
Modern Process with Threads



Multithreaded Process

Modern Process with Threads

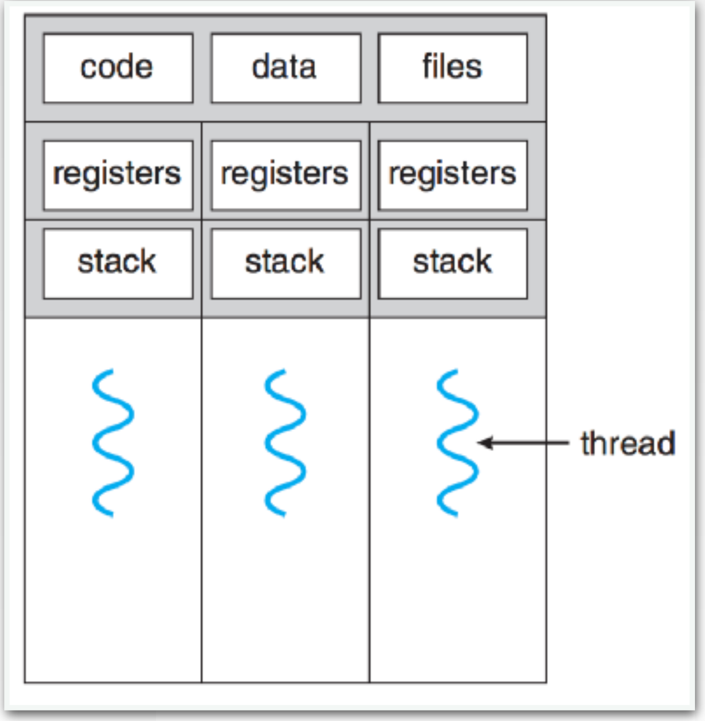
- Thread: a sequential execution stream within process ("Lightweight Process")
  - Process still contains a single Address Space
  - No protection between threads



上海科技大学
ShanghaiTech University

Modern Process with Threads

- Multithreading: a single program, a number of different concurrent activities
  - Sometimes called multitasking

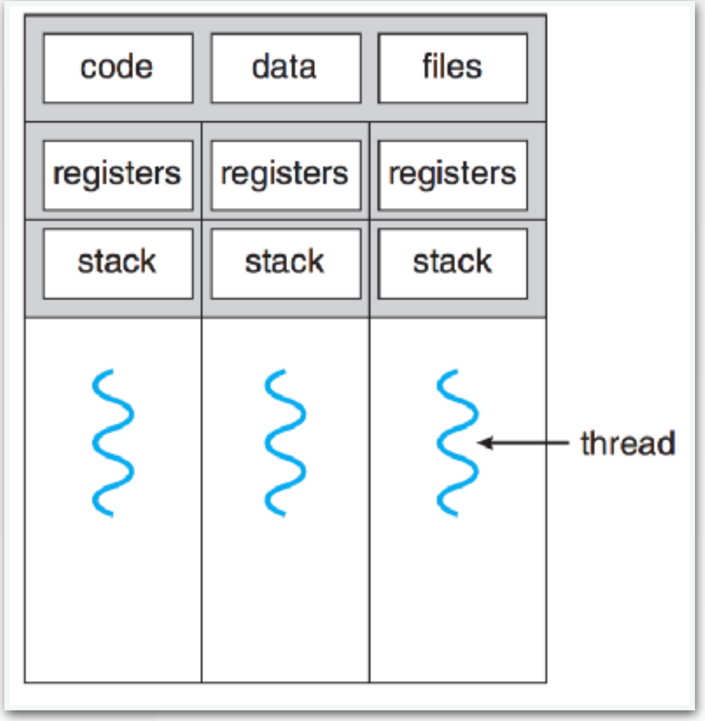


上海科技大学 ShanghaiTech University

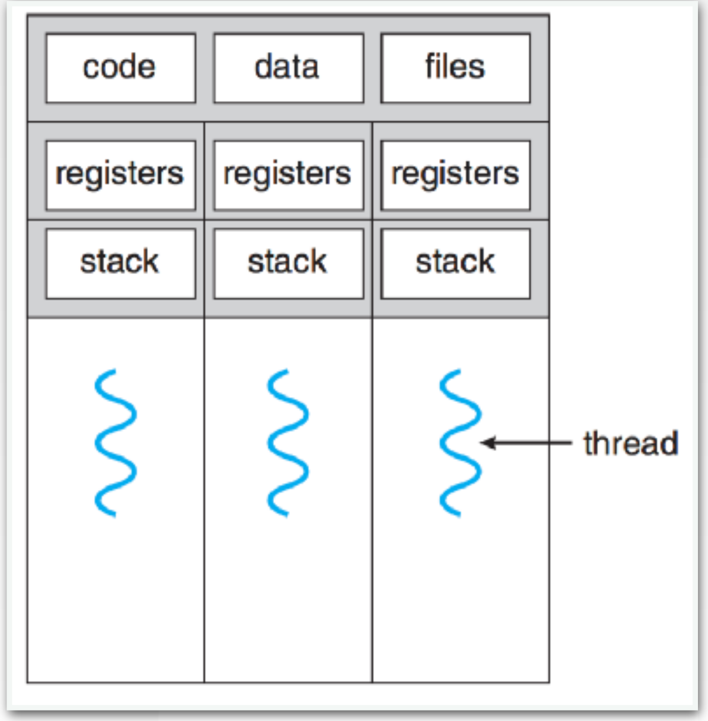
Modern Process with Threads

- Multithreading: a single program, a number of different concurrent activities
  - Sometimes called multitasking

Why separate the concept of a thread from that of a process?



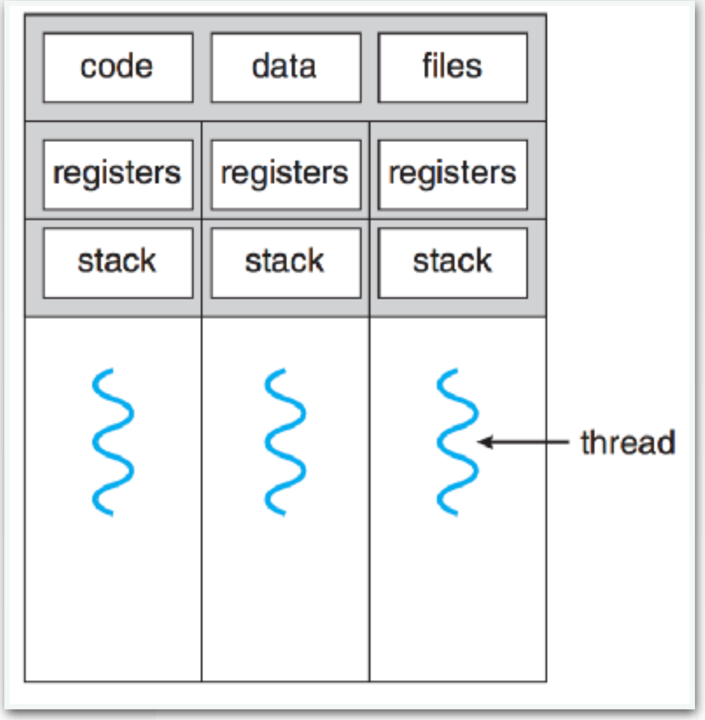
- Discuss the "thread" part of a process
  - Concurrency
- Separate from the "address space"
  - Protection
- Heavyweight process is
  - Process with one thread



上海科技大学
ShanghaiTech University

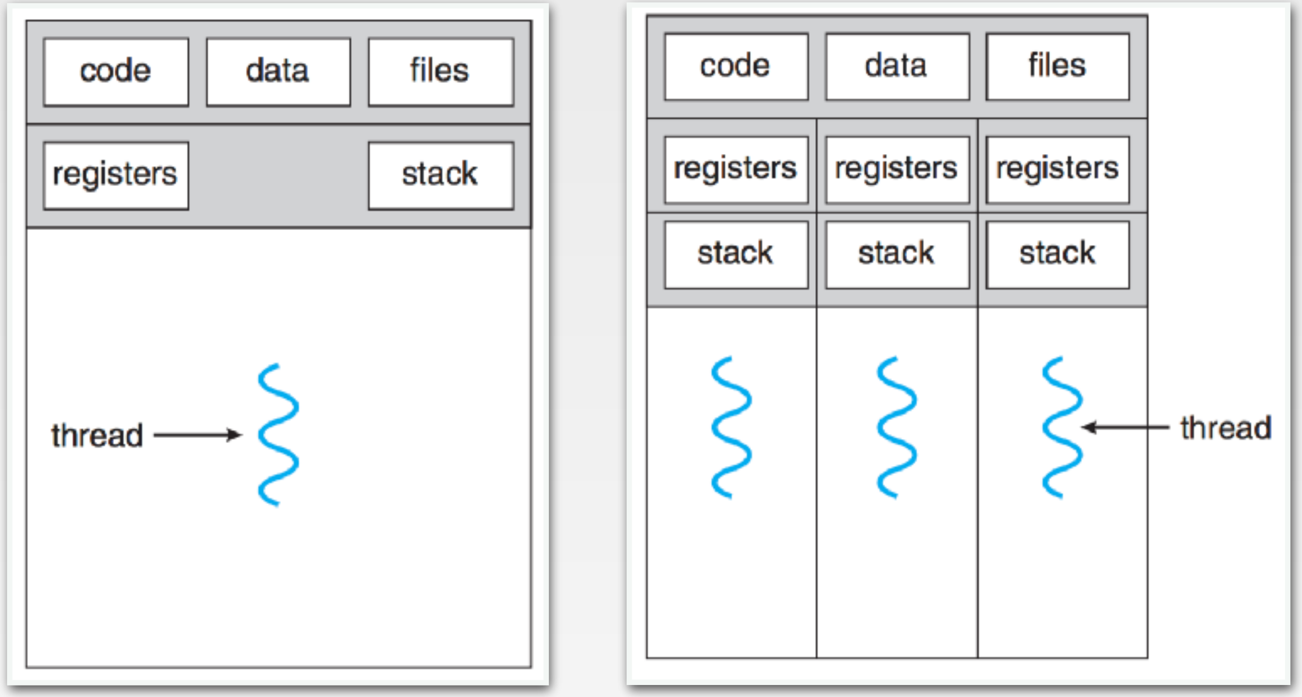
Why separate the concept of a thread from that of a process?

- Discuss the "thread" part of a process
  - Concurrency
- Separate from the "address space"
  - Protection
- Heavyweight process is
  - Process with one thread



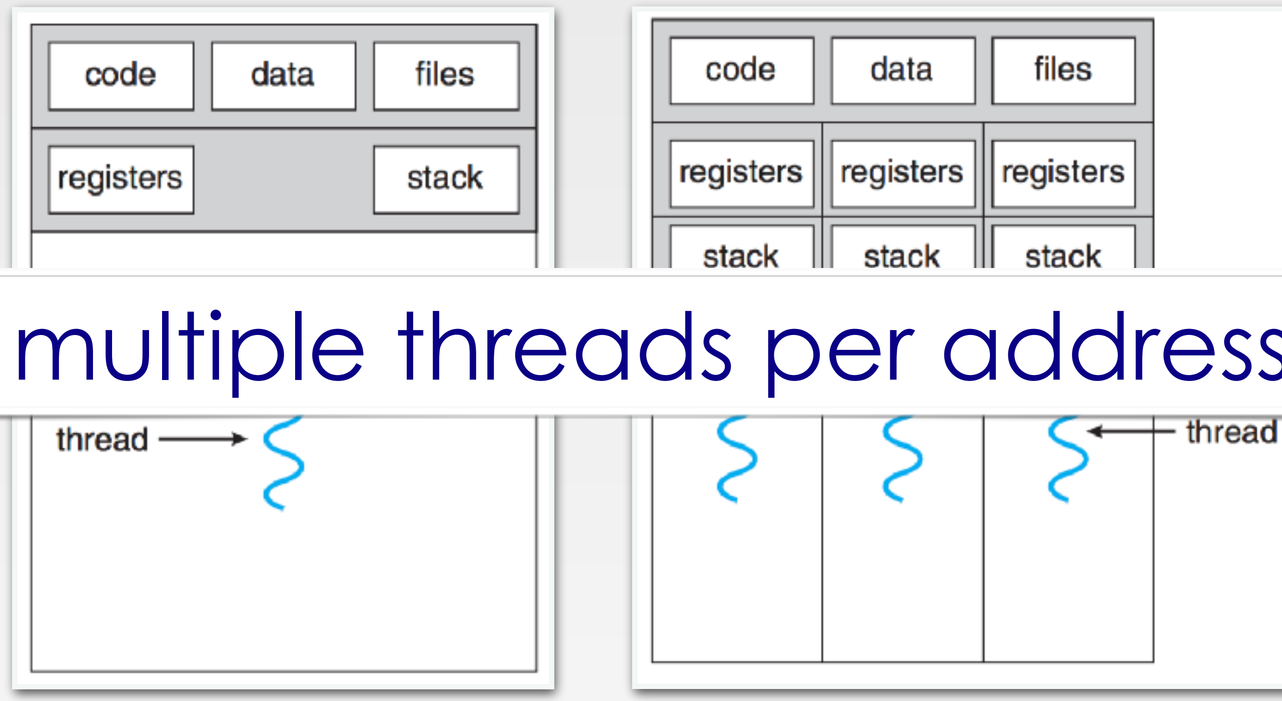
上海科技大学
ShanghaiTech University

Single vs. Multithreaded Processes



- Thread encapsulate concurrency(Active part)
- Address space encapsulate protection(Passive part)
  - Keeps buggy program from trashing the system

Single vs. Multithreaded Processes



Why have multiple threads per address space?

- Thread encapsulate concurrency(Active part)
- Address space encapsulate protection(Passive part)
  - Keeps buggy program from trashing the system

In general, Threads

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run
  - Cooperation of multiple threads in the same job confers higher throughput and improved performance
  - Apps that require sharing a common buffer

Threads (cont.)

- Threads provide a mechanism that allows sequential processes to make blocking system calls
- While also achieving parallelism
- Threads share CPU, only one thread can run at a time

Thread State

- State shared by all threads in process/ address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections)
- State "private" to each thread
  - Kept in Thread Control Block (TCB)
  - CPU registers (including PC)
  - Execution Stack

Thread State (cont.)

- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing
- Four states:
  - ready, blocked, running, terminated

上海科技大学 ShanghaiTech University

Shared vs. Per- Thread State

Shared State

- Heap
- Global Variables
- Code

Per-Thread State

- Thread Control Book (TCB)
- Stack Information
- Saved Registers
- Thread Metadata
- Stack

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}

B() {
  C(2);
}

C() {
  A(2);
}

A(1);
```

Stack Pointer →

A: tmp =1 ret = exit
B: ret = A+2
C: ret = B+1
A: tmp =2 ret = C+1

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

MIPS: SW Convention for Registers

- Before calling procedure
  - Save caller-saves regs
  - Save v0, v1
  - Save ra
- After return, assume
  - Callee-save reg OK
  - gp, sp, fp restored
  - Other things trashed

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	...		(callee must save)
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont'd)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	Pointer to global area
8	t0	temporary: caller saves	29	sp	Stack pointer
...		(callee can clobber)	30	fp	frame pointer
15	t7		31	ra	Return Address (HW)

上海科技大学
ShanghaiTech University

Motivational Example for Threads

- Imagine the following C program:

```
main(){
  ComputePI("pi.txt");
  PrintClassList("classlist.txt");
}
```

- What is the behavior here?
  - Program would never print out class list
  - Why? `ComputePI` would never finish

User vs. Kernel Threads

- User threads: user-level threads library
  - 3 primary thread libraries
    - POSIX Pthreads
    - Windows threads
    - Java threads
- Kernel threads: supported by Kernel
  - Examples: Windows, Linux, Mac OS X

Kernel Threads

- Native threads supported directly by kernel
- Every thread can run or block independently
- One process may have several threads waiting on different things
- Downside: Expensive

User Threads

- Supported above the kernel, via a set of library calls at the user level
  - May have several user threads per kernel thread
  - May be scheduled non-preemptively relative to each other (only switch on **`yield()`**)
- Advantages: Cheap, Fast
- Downside: If kernel is single threaded, system call from any thread can block the entire task

上海科技大学 ShanghaiTech University

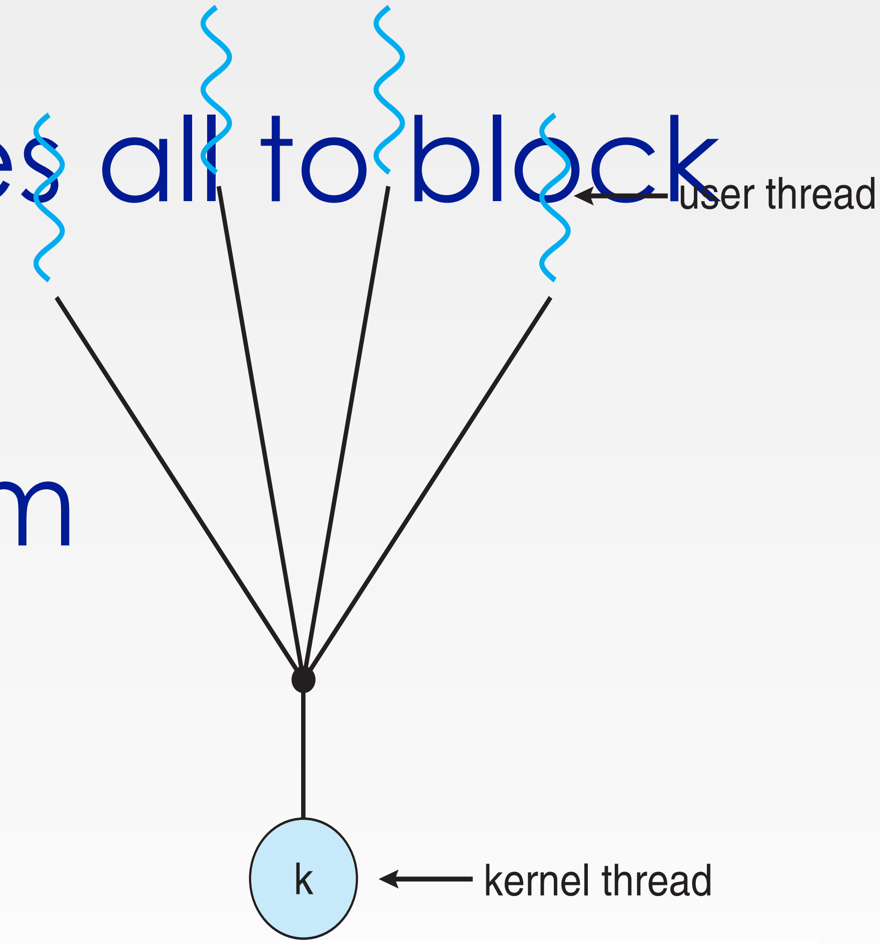
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

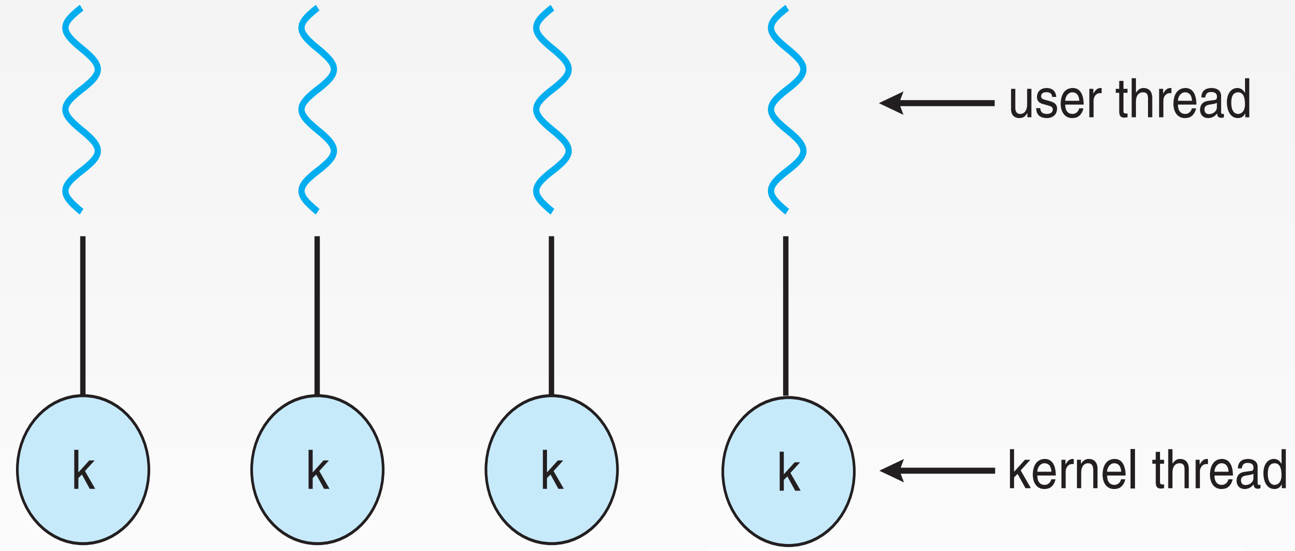
- User-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multithreads may not run

in parallel on multicore system



One-to-One

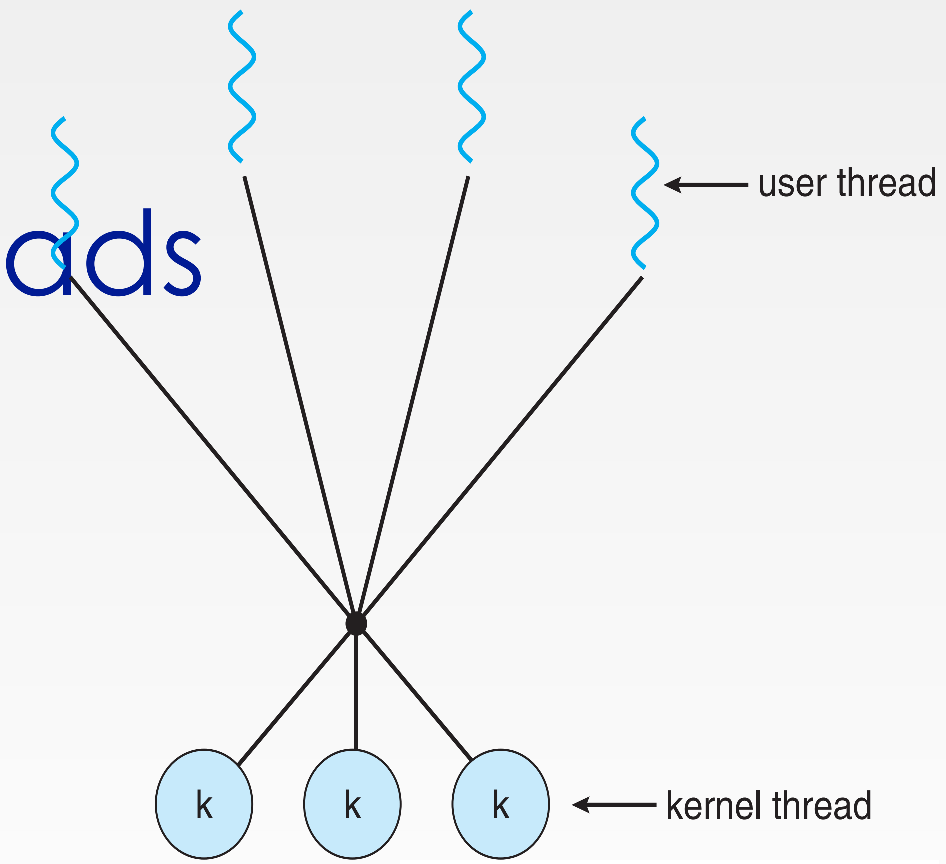
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency
- Overhead



Many-to-Many

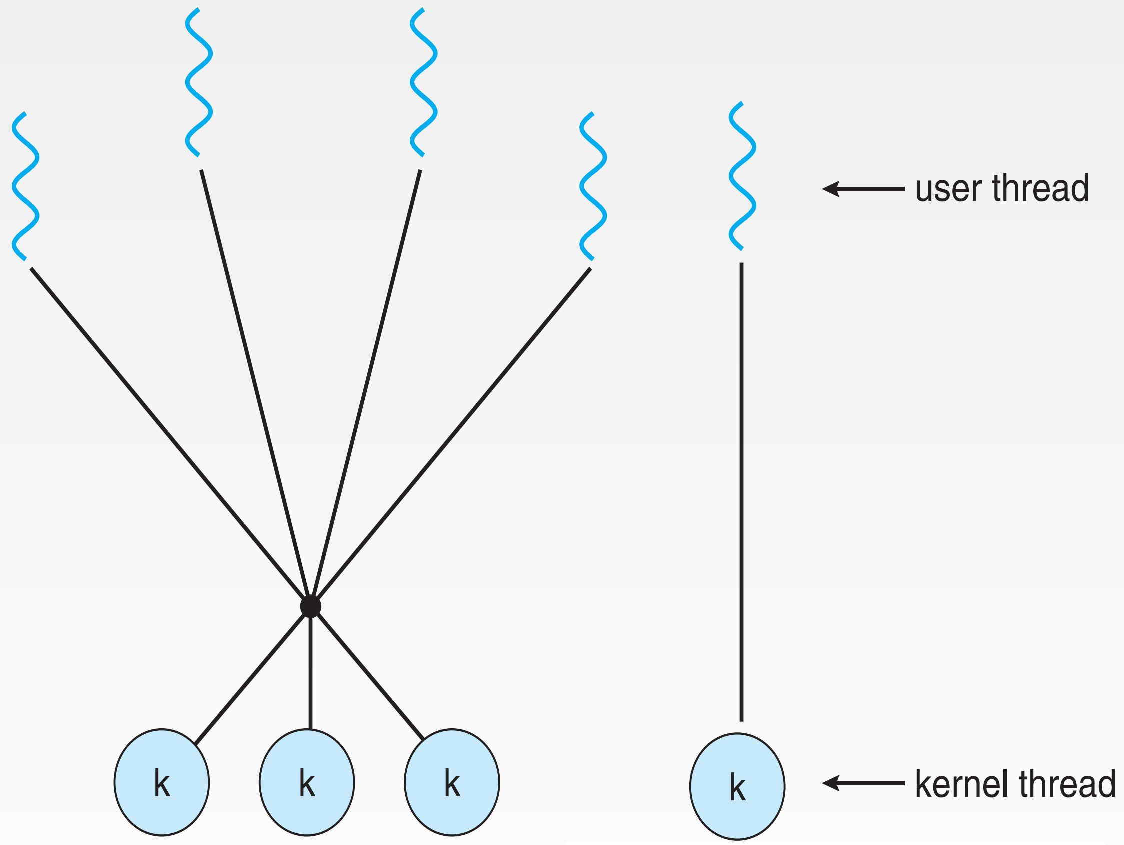
- Allows many user-level threads to be mapped to many kernel threads
- Allows the OS to create

a sufficient # of kernel threads



Two-Level Model

- Similar to M-t-M
- Allows a user thread to be bound to kernel thread



Thread Libraries

- Provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user/kernel level
- a POSIX standard API for thread creation and synchronization
- Specification, NOT implementation
- API specifies behavior of the thread library

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

```
  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

上海科技大学
ShanghaiTech University

Pthread Example 2

- Joining 10 threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

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

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Slightly faster to service a request with an existing thread than create a new one
  - Allows the number of threads in the apps to be bound to the size of the pool
  - Allows different strategies for running task

OpenMP

- Set of complier directives and an API for C/C++/Fortran
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions

OpenMP Example

```
void simple(int n, float *a, float *b)
{
    int i;

#pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

create as many threads as there are cores

上海科技大学 ShanghaiTech University

Use of Threads

- Version of program with Threads (loose syntax)

```
main(){
  ThreadFork(ComputePI, "pi.txt");
  ThreadFork(PrintClassList, "classlist.txt");
}
```

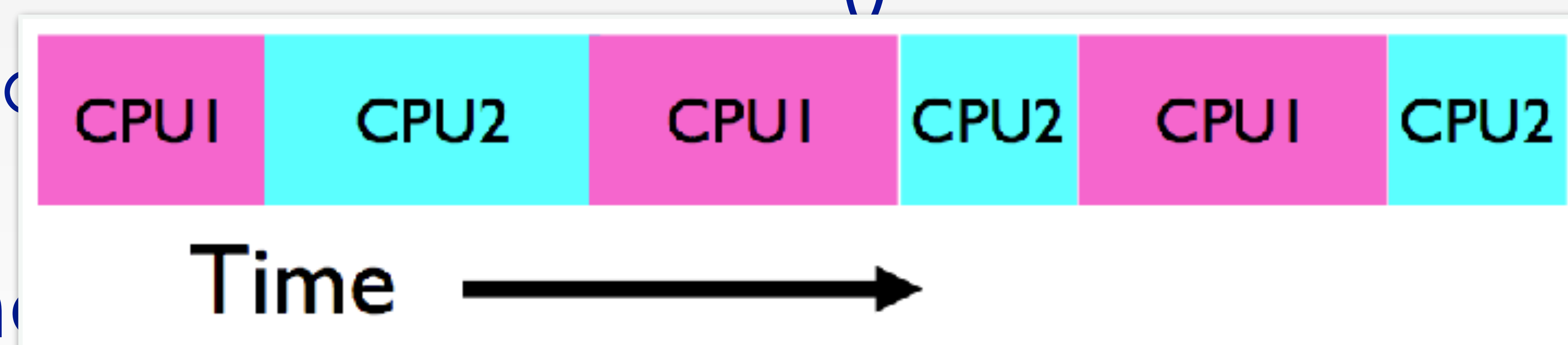
- What does ThreadFork() do?
  - Start independent thread running given procedure
- What is the behavior here?
  - As if there are two separate CPUs

Use of Threads

- Version of program with Threads (loose syntax)

```
main(){
  ThreadFork(ComputePI, "pi.txt");
  ThreadFork(PrintClassList, "classlist.txt");
}
```

- What does ThreadFork() do?
  - St… …edure



- Wh…
  - As if there are two separate CPUs

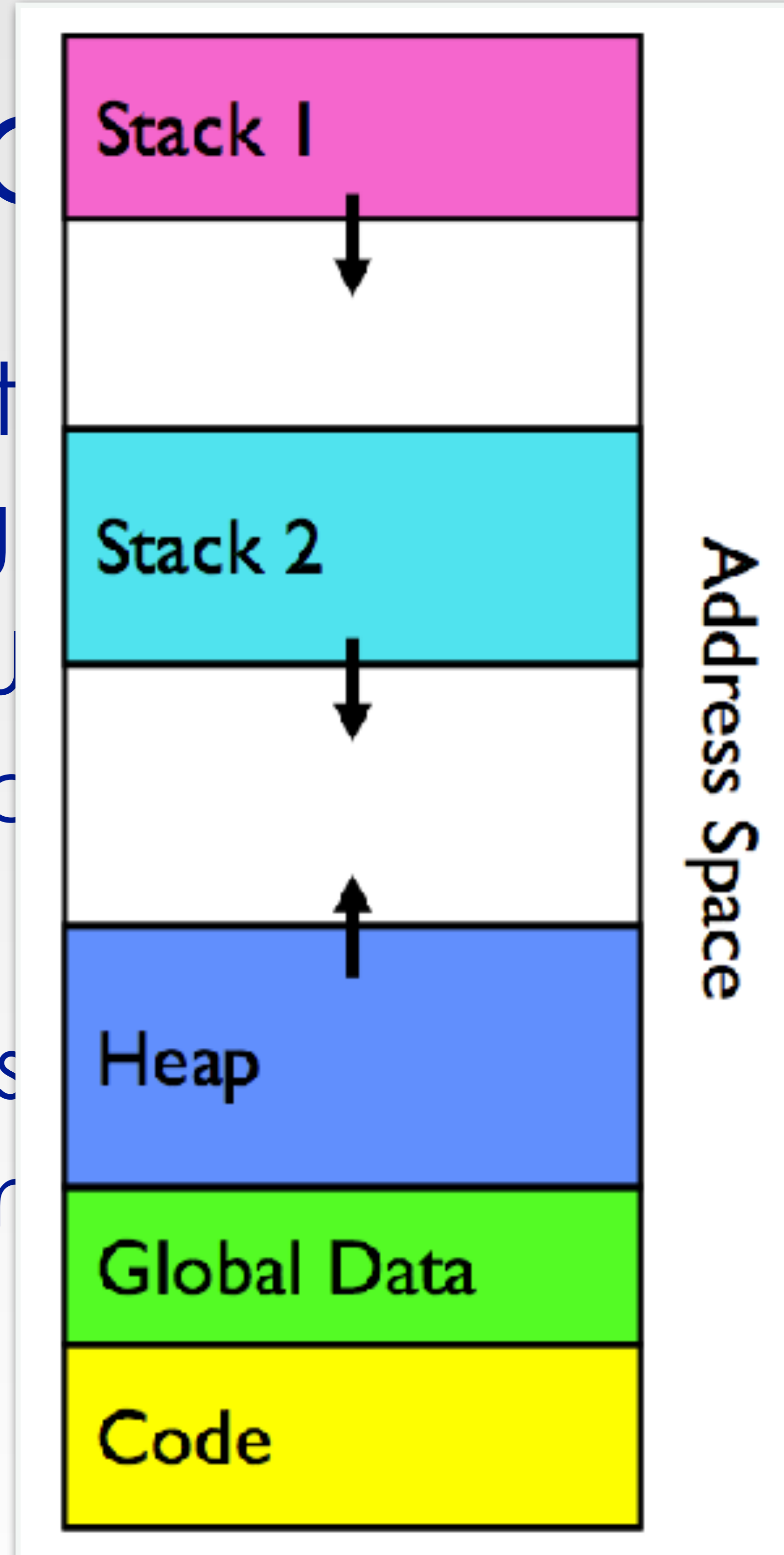
Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see:
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks
  - What happens if threads violate this?
  - How might you catch violations?

上海科技大学
ShanghaiTech University

Memory Fo vo-Threads

- If we stopped t d examined it with a debugg e:
  - Two sets of CPU
  - Two sets of Stac
- Questions:
  - How do we pos e to each other?
  - What maximum oose for the stacks
  - What happens his?
  - How might you



上海科技大学 ShanghaiTech University

Actual Thread Operations

- `thread_fork(func, args)`
  - Create a new thread to run func(args)
- `thread_yield()`
  - Relinquish processor voluntarily
- `thread_join(thread)`
  - In parent, wait for forked thread to exit, then return
- `thread_exit`
  - Quit thread and clean up, wake up joiner if any
- `pThreads`
  - POSIX standard for thread programming

Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOFCPU(newTCB);
}
```

- This is an infinite loop
  - Once could argue that this is all that the OS does
- Should we ever exit this loop?
  - When would that be?

上海科技大学 ShanghaiTech University

Running a Thread

- Consider first position: `RunThread()`
- How do I run a thread?
  - Load its state (registers, PC, SP) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets preempted

Internal Events

- Blocking on I/O
- Waiting on a "signal" from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

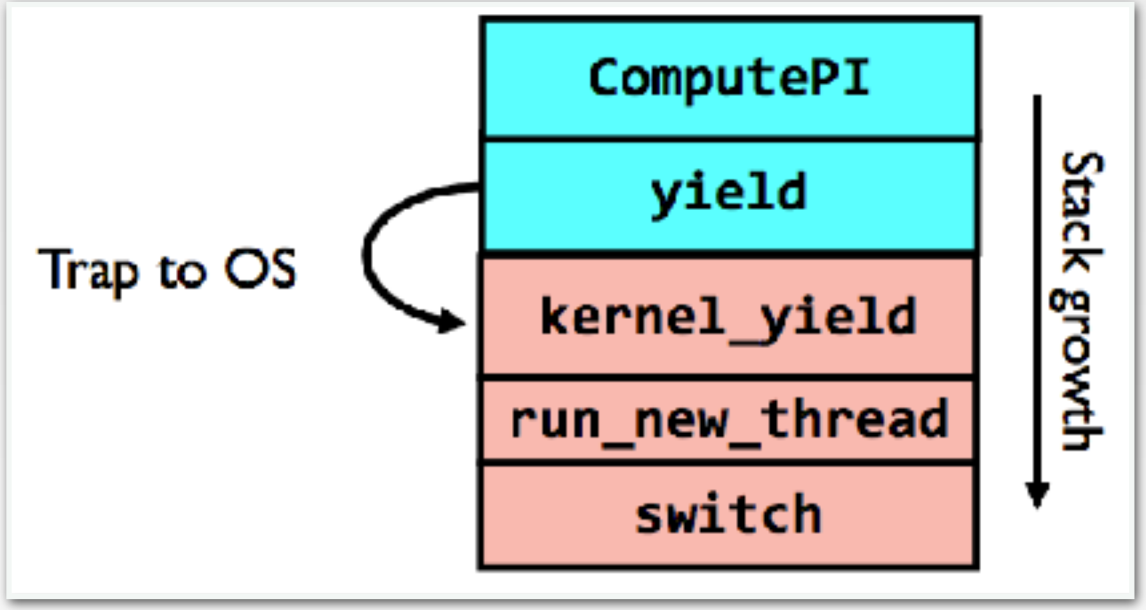
```
computePI(){
    while(TRUE){
      ComputeNextDigit();
      yield();
      }
}
```


上海科技大学 ShanghaiTech University

Stack for Yielding Thread

- How do we run a new thread?

```
run_new_thread() {
   newThread = PickNewThread();
   switch(curThread, new Thread);
   ThreadHouseKeeping();
}
```



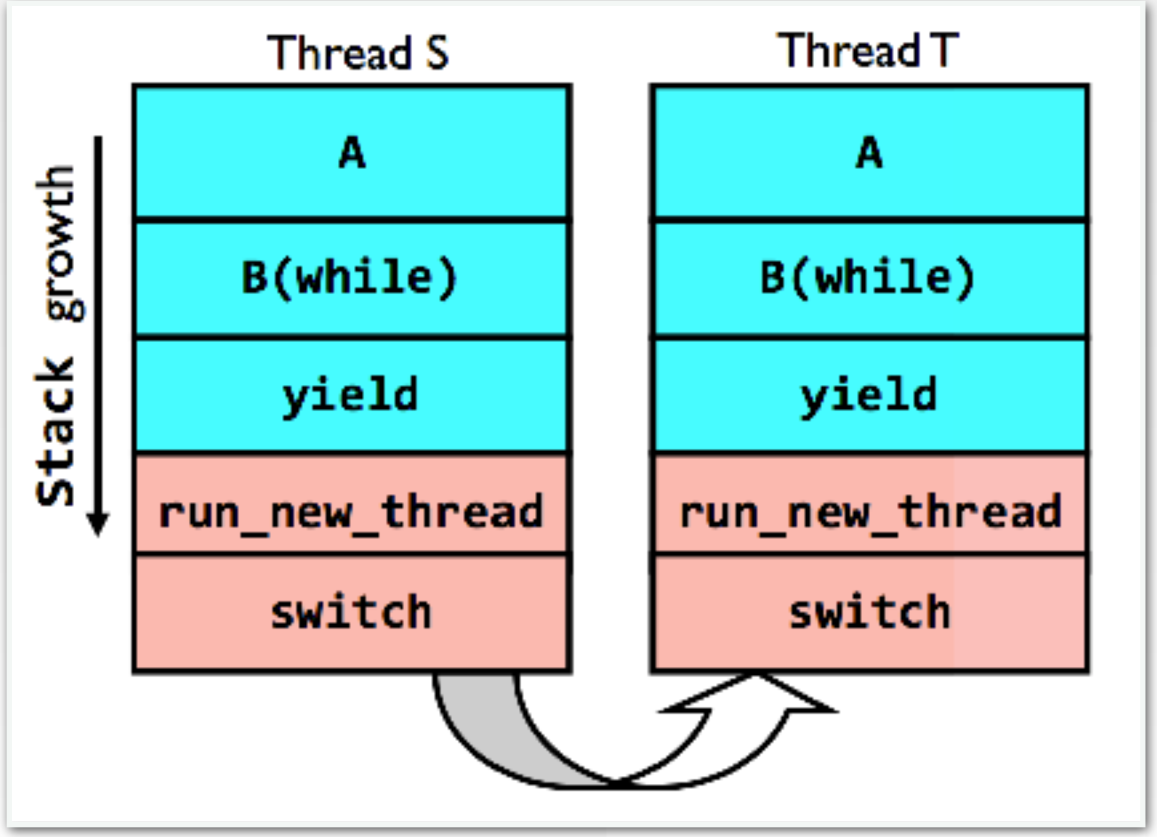
- How does dispatcher switch to a new thread?
  - Save PC, regs, SP
  - Maintain isolation

What Do the Stacks Look Like?

- Consider the following code blocks:

```
proc (A){
      B();
}
proc (B){
      while(TRUE){
          yield();
      }
}
```

- Suppose we have two threads:
  - Threads S & T



Saving/Restoring State

- Often called “Context Switch”

```
switch(tCur, tNew){
  /*unload old thread*/
  TCB[tCur].regs.r7 = CPU.r7;
  …
  TCB[tCur].regs.r0 = CPU.r0;
  TCB[tCur].regs.sp = CPU.sp;
  TCB[tCur].regs.retpc = CPU.retpc;/*return addr*/

  /*load and execute new thread*/
  CPU.r7 = TCB[tNew].regs.r7;
  …
  CPU.r0 = TCB[tNew].regs.r0;
  CPU.sp = TCB[tNew].regs.sp;
  CPU.retpc = TCB[tNew].regs.retpc;
  return; /*return to CPU.retpc*/
}
```

Switch Details (cont.)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - System will give wrong result without warning
- Can you devise an exhaustive test to test switch code?
  - No.
  - Too many combinations and inter-leavings

Switch Details (cont.)

- Cautionary tale:
  - For speed, Topaz kernel saved one instruction in **`switch{}`**
  - Carefully documented. Only works as long as kernel size < 1MB
  - What happens:
    - Time passed, people forget
    - Later, they added features to kernel (no one removes features)
    - Very weird behavior started happening

Some Numbers

- Frequency of performing context switch:
  - 10-100ms
- Context switch time in Linux: 3-4μ secs
  - Thread switching faster than process switching (100 ns)
  - But switching across cores ~2X more expensive than within-core
- Context switch time increases sharply with size of working set
  - Can increase 100x or more

Some Numbers (cont.)

- Moral: context switching depends mostly on cache limits and the process or thread's huger for memory
- Many processed are multi-threaded, so thread context switches may be either within-process or across-processes

Some Numbers (cont.)

- More ... mostly on ... three
- Man ..., so three ... her with

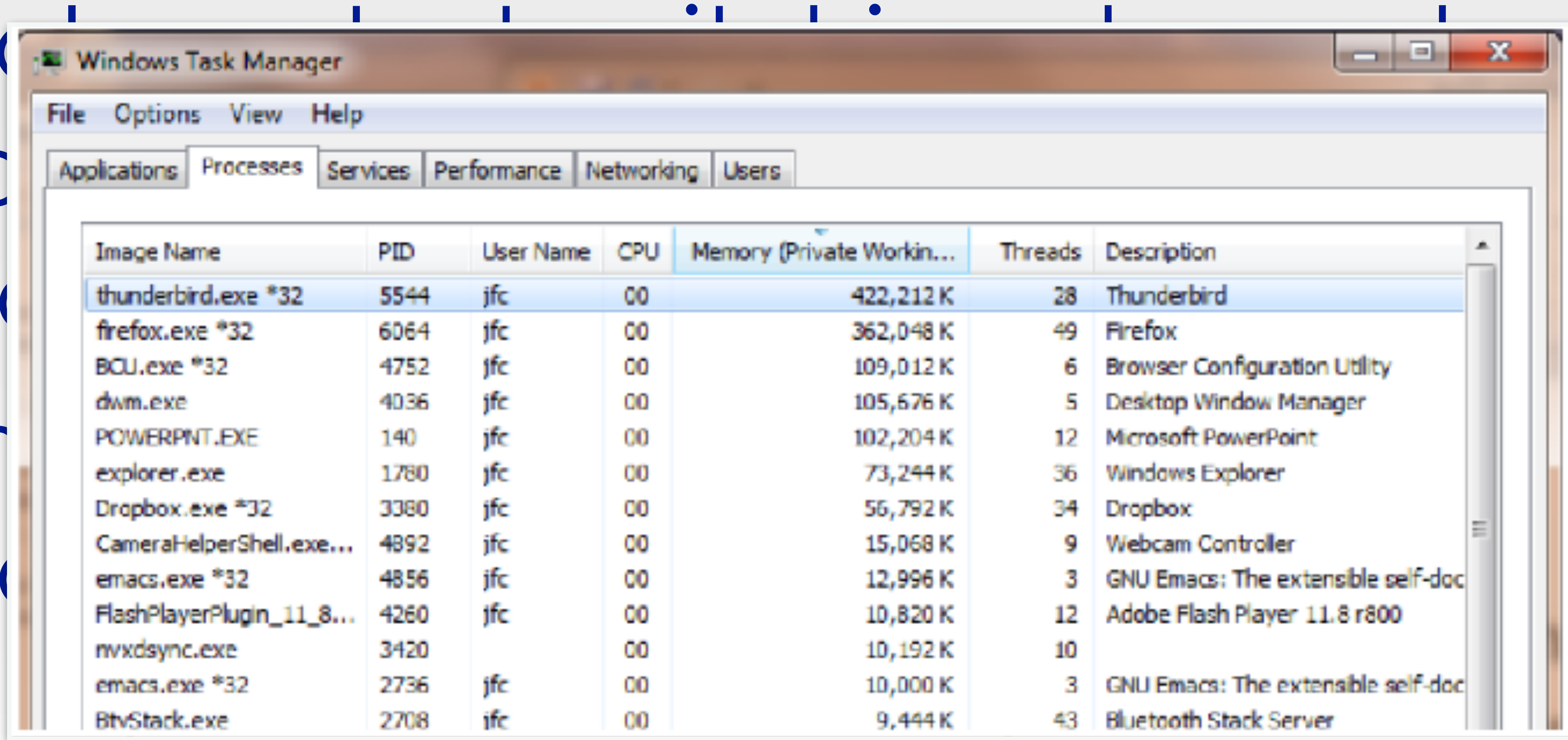
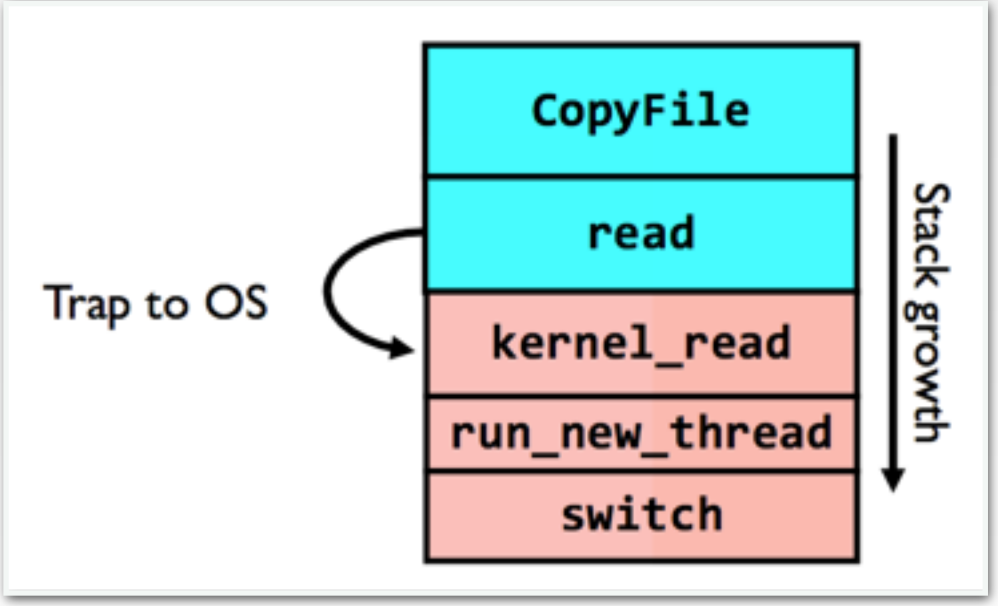


Image Name	PID	User Name	CPU	Memory (Private Workin...	Threads	Description
thunderbird.exe *32	5544	jfc	00	422,212 K	28	Thunderbird
firefox.exe *32	6064	jfc	00	362,048 K	49	Firefox
BCU.exe *32	4752	jfc	00	109,012 K	6	Browser Configuration Utility
dwm.exe	4036	jfc	00	105,676 K	5	Desktop Window Manager
POWERPNT.EXE	140	jfc	00	102,204 K	12	Microsoft PowerPoint
explorer.exe	1780	jfc	00	73,244 K	36	Windows Explorer
Dropbox.exe *32	3380	jfc	00	56,792 K	34	Dropbox
CameraHelperShell.exe...	4892	jfc	00	15,068 K	9	Webcam Controller
emacs.exe *32	4856	jfc	00	12,996 K	3	GNU Emacs: The extensible self-doc
FlashPlayerPlugin_11_8...	4260	jfc	00	10,820 K	12	Adobe Flash Player 11.8 r800
nvxdsync.exe	3420		00	10,192 K	10	
emacs.exe *32	2736	jfc	00	10,000 K	3	GNU Emacs: The extensible self-doc
BtvStack.exe	2708	jfc	00	9,444 K	43	Bluetooth Stack Server

上海科技大学 ShanghaiTech University

What happens when thread blocks on I/O?

- What happens when a thread requests a block of data from the FS?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication
  - Similar to Networking
  - or Wait for Signal/Join



ShanghaiTech University

External Events

- What happens if thread never does any I/O, never waits, or never yield control?
  - Could the **`ComputePI`** program grab all resources and never release processor?
  - Must find ways that dispatcher can regain control.
- Answer: Utilize External Events

上海科技大学 ShanghaiTech University

External Events

- Interrupts: signals from HW or SW that stop the running code and jump to kernel
- Timer: like an alarm clock that goes off every some may milliseconds

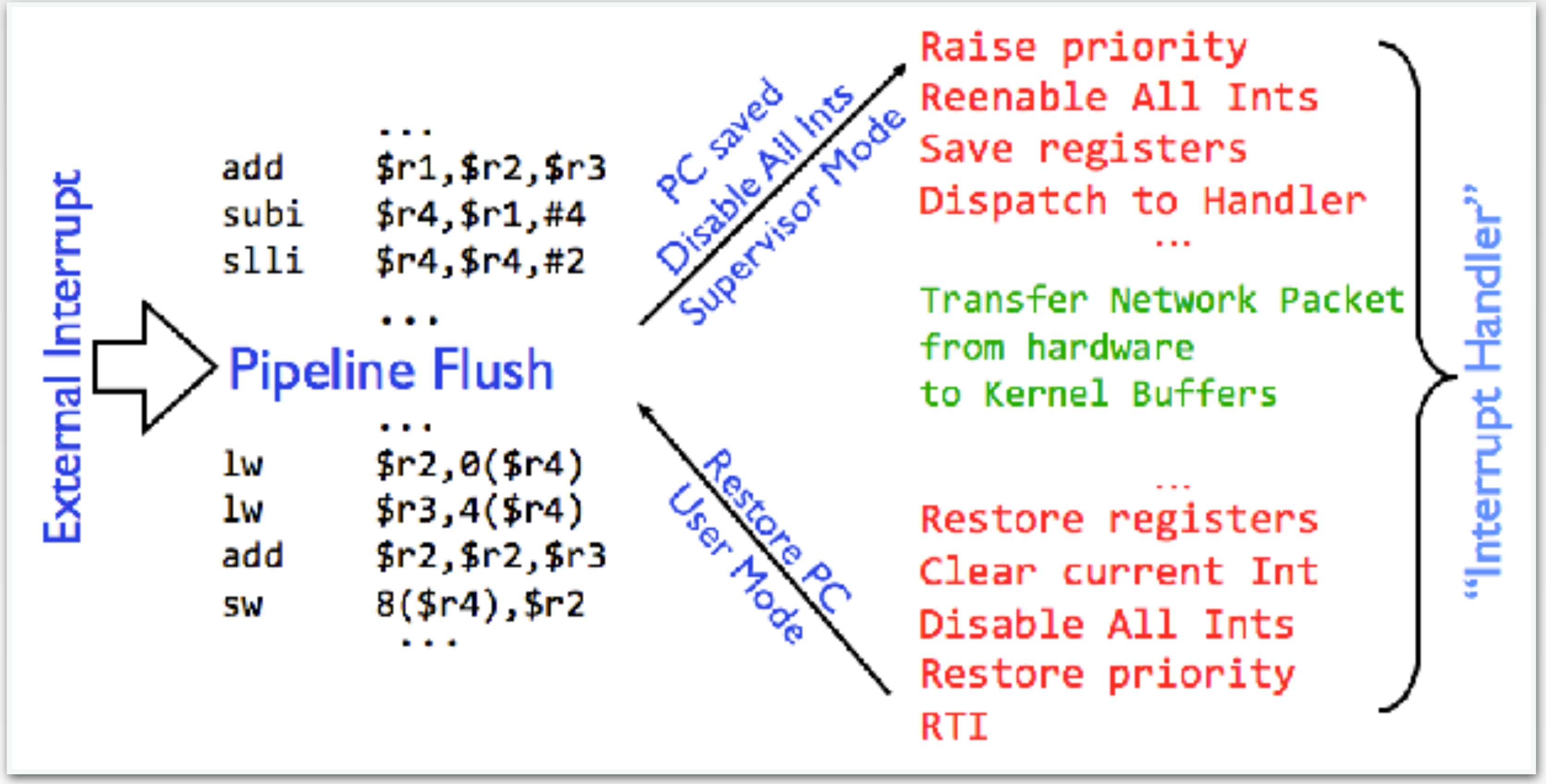
External Events

- Interrupts: signals from HW or SW that stop the running code and jump to kernel
- Timer: like an alarm clock that goes off every some may milliseconds

If we make sure that external events occur frequently enough, can ensure dispatcher runs

上海科技大学
ShanghaiTech University

Example: Network Interrupt

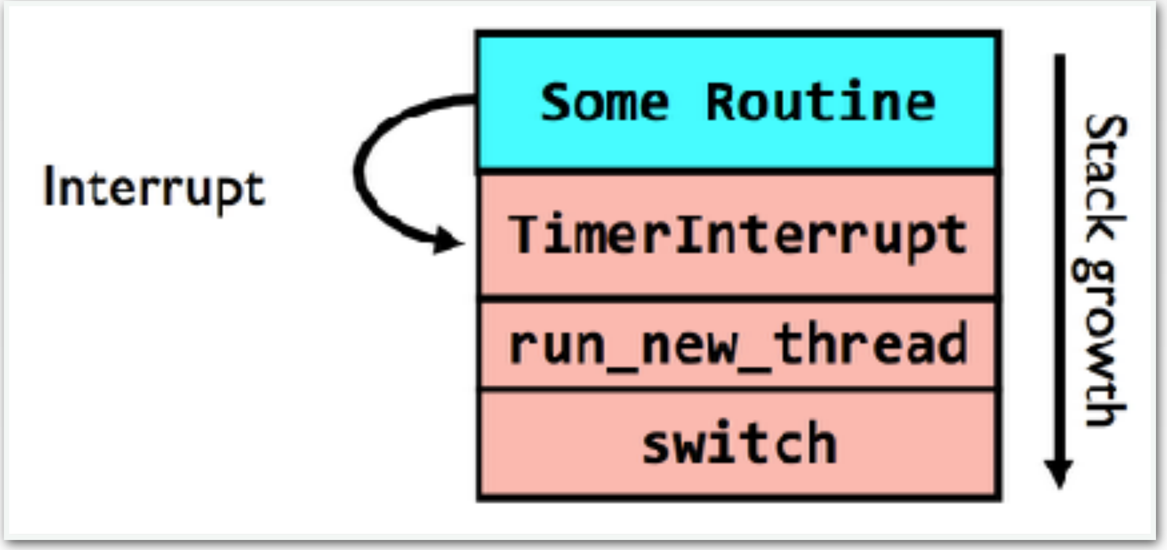


- An interrupt is a HW-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Example: Use Timer to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions
- Timer interrupt routine:

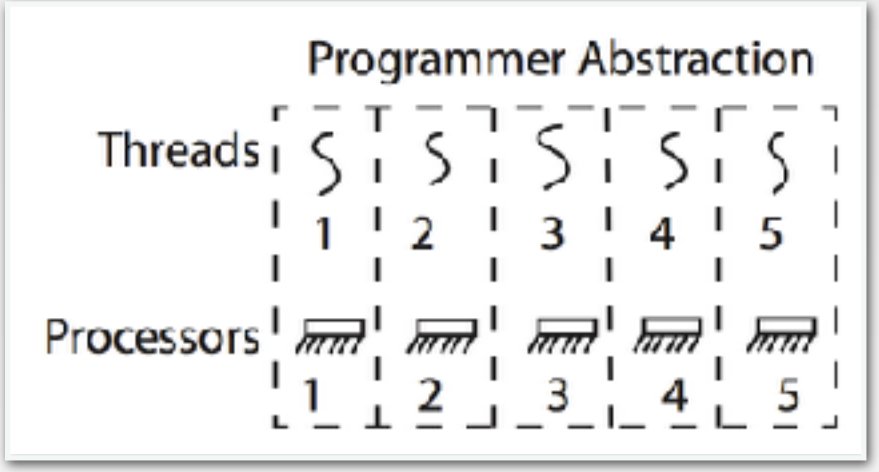
```
TimerInterrupt() {
  DoPeriodicHouseKeeping();
  run_new_thread();
}
```



Thread Abstraction

- Illusion:

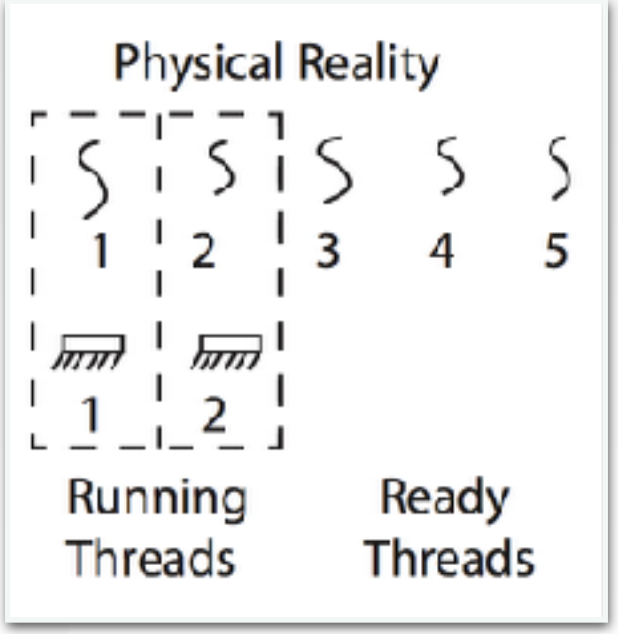
Infinite number of processors



- Reality:

Threads execute w/ variable speed

  – Programs must be designed to work with any schedule



上海科技大学
ShanghaiTech University

Programmer vs. Processor View

Programmer's View	Possible Execution #1
.	.
.	.
.	.
$x = x + 1;$	$x = x + 1;$
$y = y + x;$	$y = y + x;$
$z = x + 5y;$	$z = x + 5y;$
.	.
.	.
.	.

Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2
.	.	.
.	.	.
.	.	.
x = x + 1;	x = x + 1;	x = x + 1
y = y + x;	y = y + x;	....
z = x +5y;	z = x + 5y;	thread is suspended
.	.	other thread(s) run
.	.	thread is resumed
.	.	....
		y = y + x
		z = x + 5y

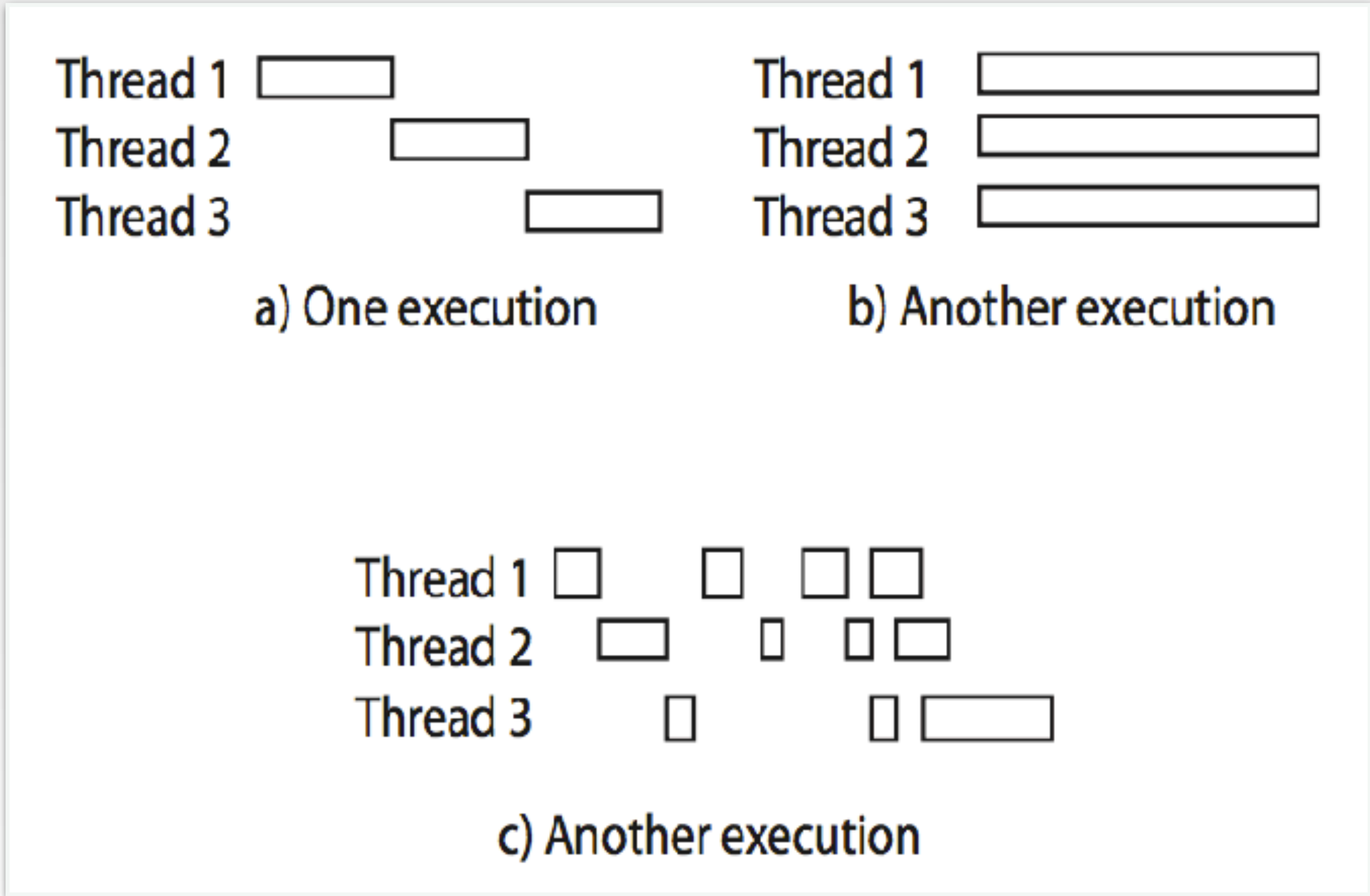
上海科技大学
ShanghaiTech University

Programmer vs. Processor View

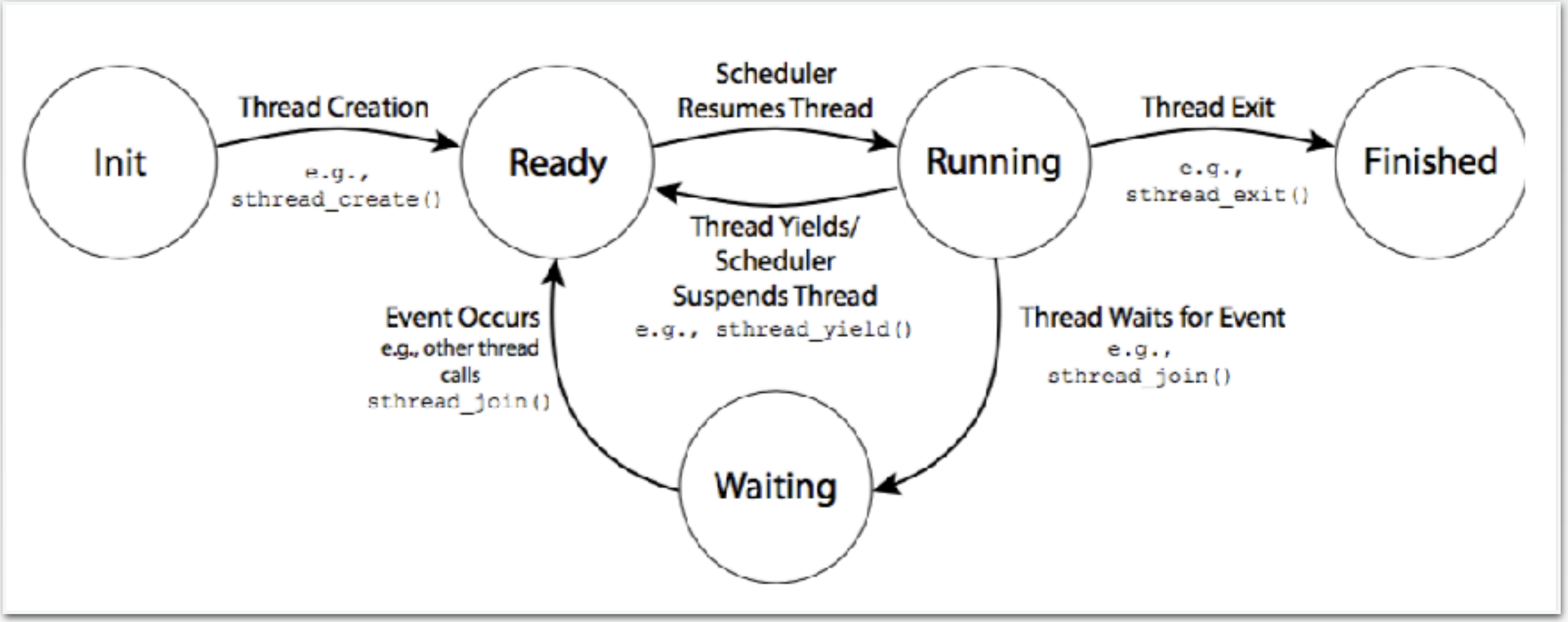
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	....	thread is resumed
		y = y + x	....
		z = x + 5y	z = x + 5y

上海科技大学 ShanghaiTech University

Possible Executions



Thread Lifecycle



Summary

- Processes have two parts
  - One or more Threads (concurrency)
  - Address Spaces (protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary or involuntary

Discussion

- Multi-processing/programming/threading
  - Definition:
    - Multiprocessing → multiple CPUs
    - Multiprogramming → Multiple Jobs or Processes
    - Multithreading → Multiple Threads per Process

Discussion (cont.)

- What does it mean to run threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, etc.
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks

Disc

- Who
  “cor
  - Sch rder
    anc
  - Disp ead to
    cor r small
    chu