CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

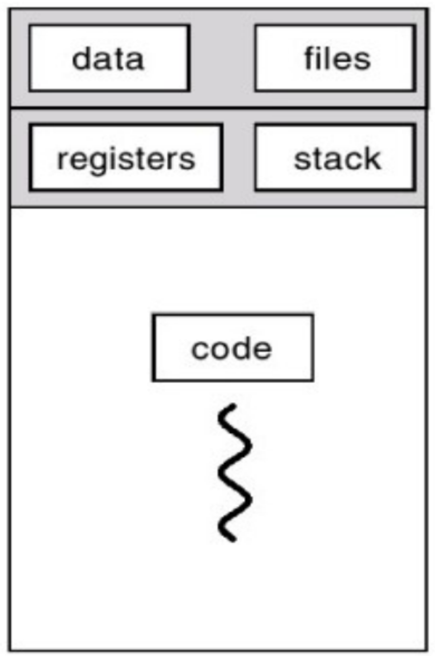
Jan27, 2021 – Threads & Concurrency

So, far we have studied single threaded programs

- Recall: Process execution
  - PC points to current instruction being run
  - SP points to stack frame of current function call
- A program can also have multiple threads of execution
- What is a thread?

Process Execution

- 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

A thread is like another copy of a process that executes independently

- Threads shares the same address space (code, heap)
- Each thread has separate PC ⟵ Private Regs, Stack

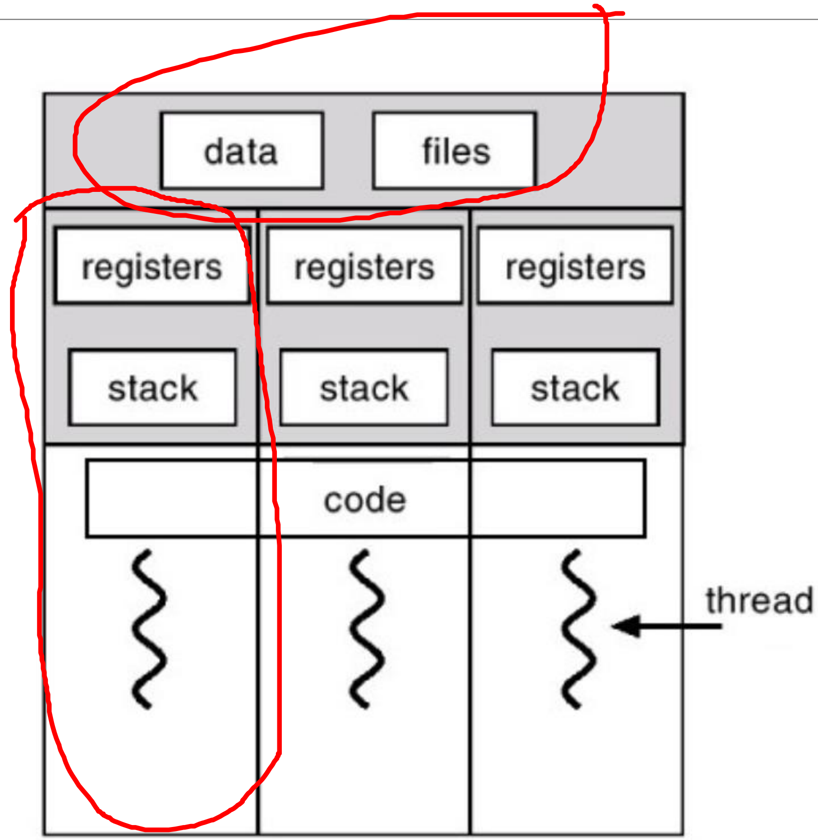
–Each thread may run over different part of the program

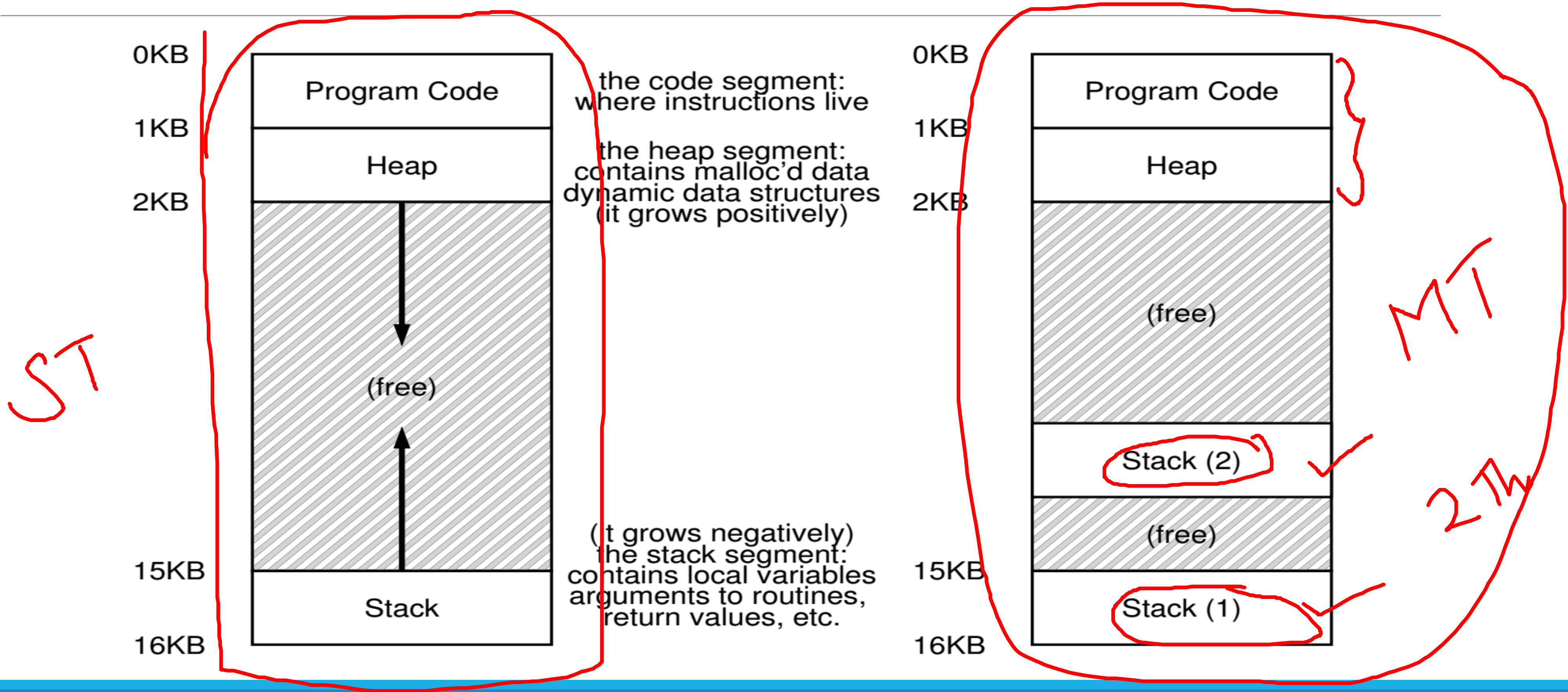
- Each thread has separate stack for independent function calls

Threads

- Separate streams of execution within a single process
- Threads in a process not isolated from each other
- Each thread state (thread control block) contains
  - Registers (including EIP, ESP)
  - stack

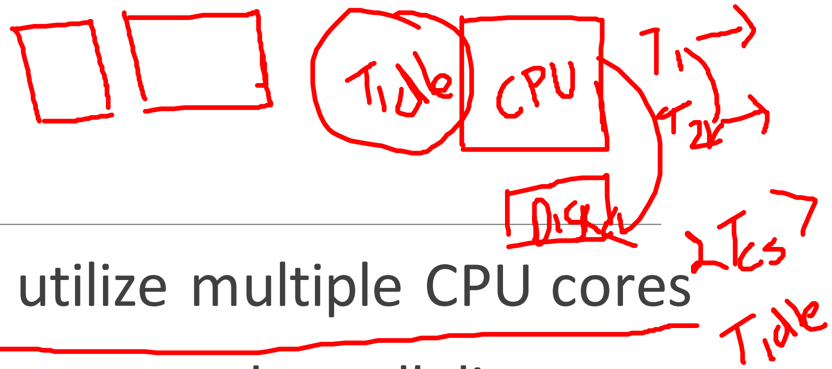
TCB





ST
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment:
where instructions live
the heap segment:
contains malloc'd data
dynamic data structures
(it grows positively)
(it grows negatively)
the stack segment:
contains local variables
arguments to routines,
return values, etc.
MT
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)
2Th

Why threads ?



- Parallelism: a single process can effectively utilize multiple CPU cores
  - –Understand the difference between concurrency and parallelism
  - –Concurrency: running multiple threads/processes at the same time, even on single CPU core, byinterleaving their executions
  - –Parallelism: running multiple threads/processes in parallel over different CPU cores
- Even if no parallelism, concurrency of threads ensures effective use of CPU when one of thethreads blocks (e.g., for I/O) TRUE

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

Threads vs Processes

Parent P forks a child C

–P and C do not share any memory

–Need complicated IPC mechanisms to communicate

–Extra copies of code, data in memory

- Parent P executes two threads T1 and T2

–T1 and T2 share parts of the address space

–Global variables can be used for communication

–Smaller memory footprint

- Threads are like separate processes, except they share the same address space

Amdahl's Law

Comparison

- 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.

Trivia

Do the following statements apply to Processes (P), Threads (T) or both (B)

Can share a virtual address space — Threads

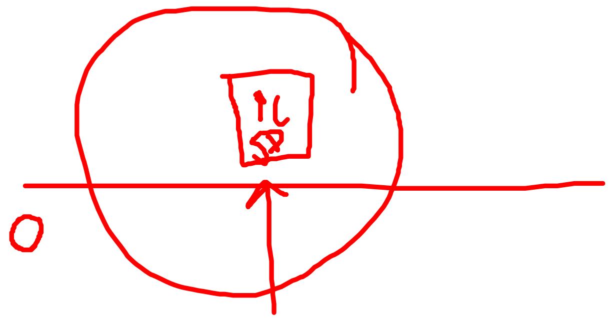
Take longer to context switch — P

Have an execution context — Both

Usually result in hotter caches when multiple exist — T

Make use of some communication mechanisms — B

Scheduling Threads



- OS schedules threads that are ready to run independently, much like processes
- The context of a thread (PC, registers) is saved into/restored from thread control block (TCB)
  - Every PCB has one or more linked TCBs

POSIX

- Threads that are scheduled independently by kernel are called kernel threads
  - E.g., Linux pthreads are kernel threads
- In contrast, some libraries provide user-level threads
  - User program sees multiple threads 1
  - Library multiplexes larger number of user threads over a smaller number of kernel threads
  - Low overhead of switching between user threads (no expensive context switch)
  - But multiple user threads cannot run in parallel -2

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

NULL

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 contd..

pthread_detach()

- 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 <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

3 A | B

```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;
}
```

Thread Trace

Single CPU

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		