

Join on Zoom if you are
remote (see website for link).



377 Operating Systems

Introduction to Concurrency



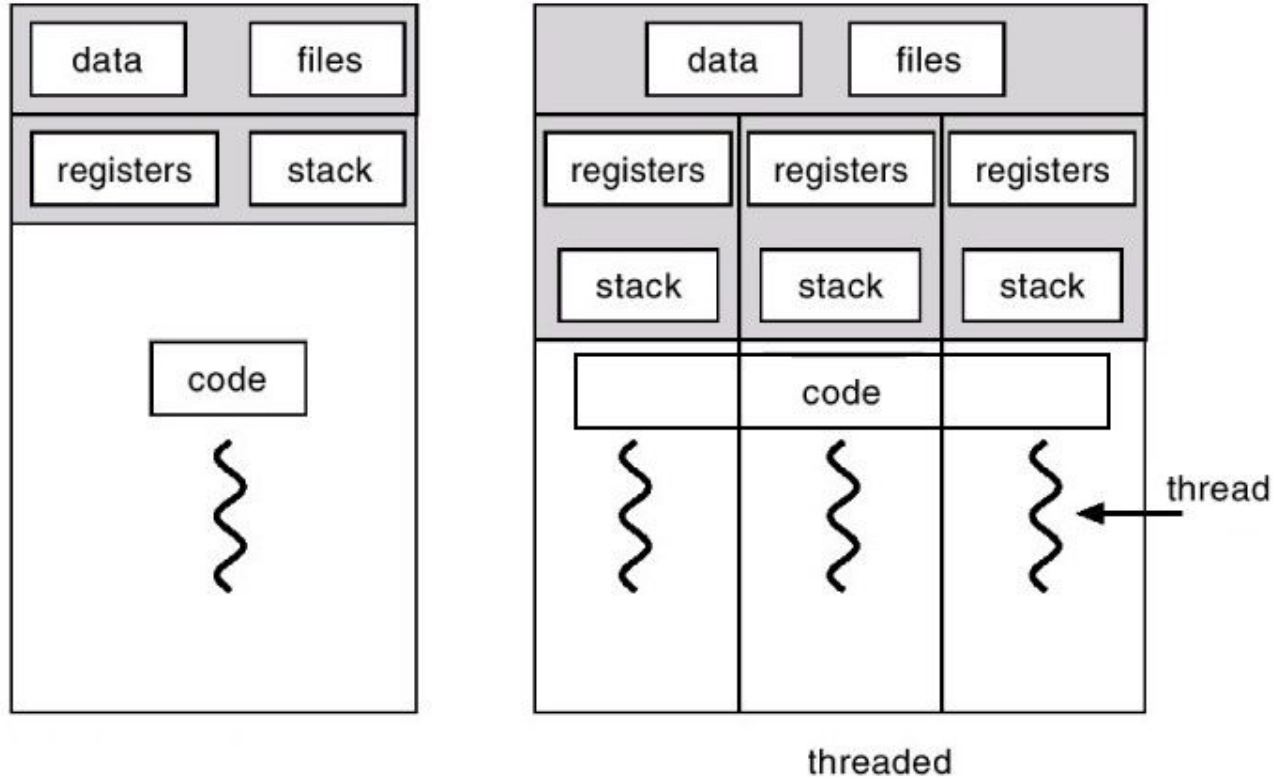
Virtualization: Processes

Reality: Single CPU

Virtualization: Multiple Processes

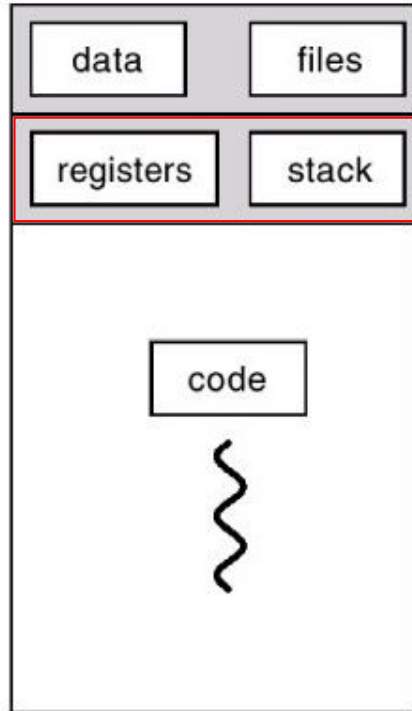


Another Abstraction: Threads

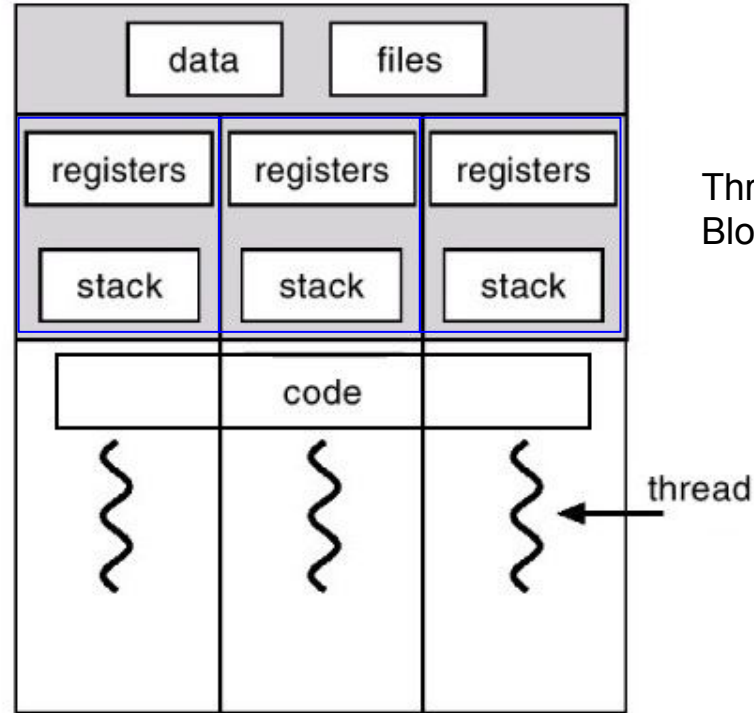


Another Abstraction: Threads

Process
Control Block
(PCB)



Thread Control
Block (TCB)



threaded

Single-Threaded and Multi-Threaded Address Space

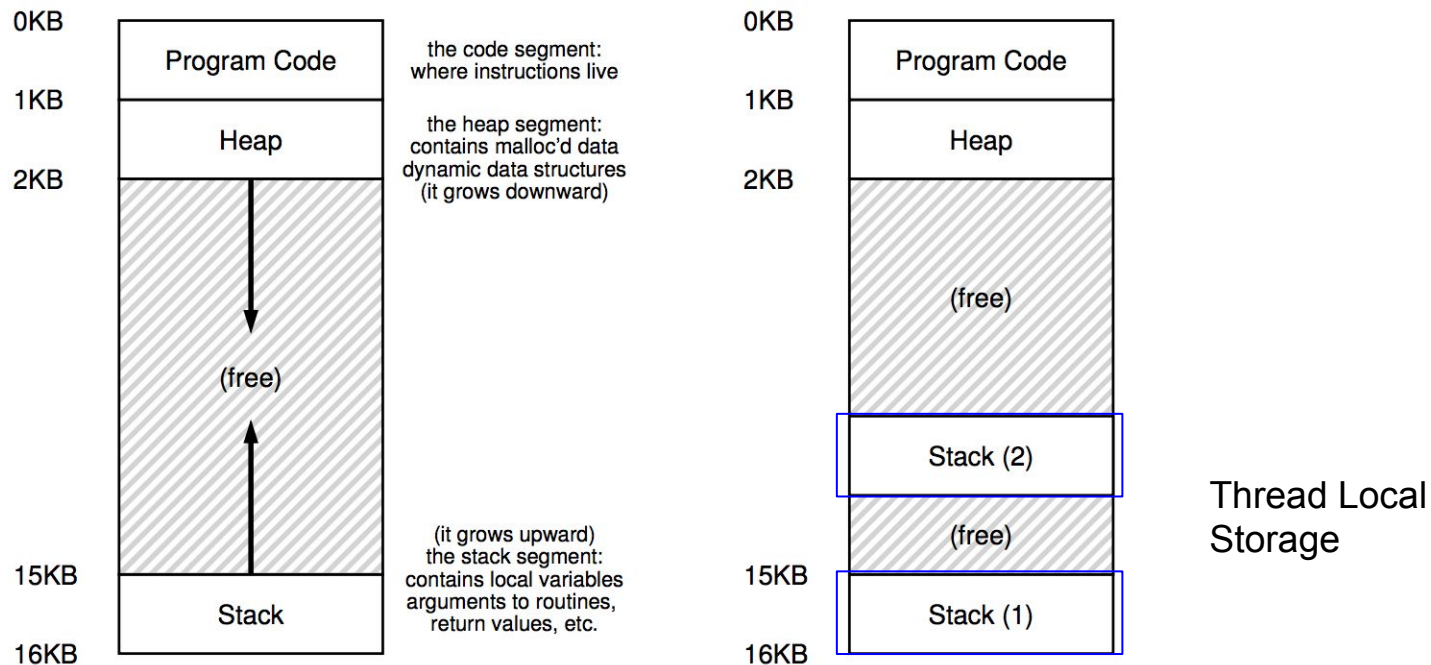


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

Why Use Threads?

The first one is simple: **parallelism**

Imagine you are writing a program that performs operations on very large arrays, for example, adding two large arrays together, or incrementing the value of each element in the array by some amount.

Why Use Threads?

The first one is simple: **parallelism**

Imagine you are writing a program that performs operations on very large arrays, for example, adding two large arrays together, or incrementing the value of each element in the array by some amount.

If you are running on just a single processor, the task is straightforward: just perform each operation and be done.

Why Use Threads?

The first one is simple: **parallelism**

Imagine you are writing a program that performs operations on very large arrays, for example, adding two large arrays together, or incrementing the value of each element in the array by some amount.

If you are running on just a single processor, the task is straightforward: just perform each operation and be done.

However, if you are executing the program on a system with multiple processors, you have the potential of speeding up this process considerably by using the processors to each perform a portion of the work.

Why Use Threads?

The second reason is more subtle:

to avoid blocking program progress due to slow I/O

Why Use Threads?

The second reason is more subtle:

to avoid blocking program progress due to slow I/O

Imagine that you are writing a program that performs different types of I/O: either waiting to send or receive a message, for an explicit disk I/O to complete, or even (implicitly) for a trap/fault to finish.

Why Use Threads?

The second reason is more subtle:

to avoid blocking program progress due to slow I/O

Imagine that you are writing a program that performs different types of I/O: either waiting to send or receive a message, for an explicit disk I/O to complete, or even (implicitly) for a trap/fault to finish.

Instead of waiting, your program may wish to do something else, including utilizing the CPU to perform computation, or even issuing further I/O requests. Using threads is a natural way to avoid getting stuck.

Why Use Threads?

The second reason is more subtle:

to avoid blocking program progress due to slow I/O

while one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful.

Why Use Threads?

The second reason is more subtle:

to avoid blocking program progress due to slow I/O

while one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful.

Threading enables overlap of I/O with other activities within a single program, much like multiprocessing did for processes across programs.

Why Use Threads?

The second reason is more subtle:

to avoid blocking program progress due to slow I/O

while one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful.

Threading enables overlap of I/O with other activities within a single program, much like multiprocessing did for processes across programs.

Examples: many modern server-based applications (web servers, database management systems, and the like) make use of threads in their implementations.

Why Not Use Processes?

Of course, in either of the cases mentioned above, you could use multiple processes instead of threads.

Why Not Use Processes?

Of course, in either of the cases mentioned above, you could use multiple processes instead of threads.

However, threads **share an address space** and thus make it easy to share data.

Why Not Use Processes?

Of course, in either of the cases mentioned above, you could use multiple processes instead of threads.

However, threads **share an address space** and thus make it easy to share data.

Thus, threads are a natural choice when constructing these types of programs.

Processes are a more sound choice for logically separate tasks where little sharing of data structures in memory is needed.

Thread Example

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Thread Example

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Thread Example

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Thread Example

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Thread Example

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Question

Which prints first A or B?

```
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");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (1)

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"		

Thread Trace (2)

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"

Thread Trace (2)

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		
<i>returns immediately; T1 is done</i>		
waits for T2		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (2)

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		
<i>returns immediately; T1 is done</i>		
waits for T2		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (2)

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 <i>returns immediately; T1 is done</i> waits for T2 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (2)

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 <i>returns immediately; T1 is done</i> waits for T2 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (2)

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 <i>returns immediately; T1 is done</i> waits for T2 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (2)

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 <i>returns immediately; T1 is done</i>		
waits for T2 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (2)

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		
<i>returns immediately; T1 is done</i>		
waits for T2		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (2)

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		
<i>returns immediately; T1 is done</i>		
waits for T2		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (3)

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		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (3)

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		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (3)

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		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (3)

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		
<i>returns immediately; T2 is done</i>		
prints "main: end"		

Thread Trace (3)

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 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (3)

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 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (3)

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 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (3)

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 <i>returns immediately; T2 is done</i> prints "main: end"		

Thread Trace (3)

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 <i>returns immediately; T2 is done</i>		
prints "main: end"		

Threads are useful, but there are problems.

```
static volatile int counter = 0;

void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    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: done with both (counter = %d)\n", counter);
    return 0;
}
```

Threads are useful, but there are problems.

```
static volatile int counter = 0;

void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    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: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Threads are useful, but there are problems.

```
static volatile int counter = 0;

void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    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: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Threads are useful, but there are problems.

```
static volatile int counter = 0;

void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    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: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

problems

Not only is each run wrong, but also yields a different result!

why does this happen?

```
void *  
mythread(void *arg)
```

```
int
main(int argc, char
{
```

```
// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done with both (counter = %d)\n", counter);
return 0;
}
```

```
(counter = 19221041)
```

The Heart of the Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates for the update to counter. In this case, we wish to simply add a number (1) to counter.

The Heart of the Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates for the update to counter. In this case, we wish to simply add a number (1) to counter.

Thus, the code sequence for doing so might look something like this (in x86):

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

The Heart of the Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates for the update to counter. In this case, we wish to simply add a number (1) to counter.

Thus, the code sequence for doing so might look something like this (in x86):

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

counter

The Heart of the Problem: Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
	mov %eax, 0x8049a1c		113	51	51

The Heart of the Problem: Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt					
	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
	mov %eax, 0x8049a1c		113	51	51

The Heart of the Problem: Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
	mov %eax, 0x8049a1c		113	51	51

The Heart of the Problem: Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	<i>mov 0x8049a1c, %eax</i>		105	50	50
	<i>add \$0x1, %eax</i>		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		<i>mov 0x8049a1c, %eax</i>	105	50	50
		<i>add \$0x1, %eax</i>	108	51	50
		<i>mov %eax, 0x8049a1c</i>	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
	<i>mov %eax, 0x8049a1c</i>		113	51	51

The Heart of the Problem: Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
	mov %eax, 0x8049a1c		113	51	51

The Heart of the Problem: Uncontrolled Scheduling

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add \$0x1, %eax		108	51	50
interrupt	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add \$0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
	mov %eax, 0x8049a1c		113	51	51

RACE CONDITION!

The Wish for Atomicity

The problem is that we must execute 3 instructions to perform the increment:

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

The Wish for Atomicity

The problem is that we must execute 3 instructions to perform the increment:

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

What if we could do this with only a single atomic instruction:

```
memory-add 0x8049a1c, $0x1
```

The Wish for Atomicity

Thus, what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call synchronization primitives.

By using these hardware synchronization primitives, in combination with some help from the operating system, we will be able to build multi-threaded code that accesses critical sections in a synchronized and controlled manner.

THE CRUX:

HOW TO PROVIDE SUPPORT FOR SYNCHRONIZATION

What support do we need from the hardware in order to build useful synchronization primitives? What support do we need from the OS? How can we build these primitives correctly and efficiently? How can programs use them to get the desired results?

One more problem: waiting for another

We have set up the problem of concurrency as if only one type of interaction occurs between threads:

accessing shared variables and the need to support atomicity for critical sections.

One more problem: waiting for another

We have set up the problem of concurrency as if only one type of interaction occurs between threads:

accessing shared variables and the need to support atomicity for critical sections.

As it turns out, there is another common interaction that arises:

One thread must wait for another to complete some action before it continues.

This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.

One more problem: waiting for another

We have set up the problem of concurrency as if only one type of interaction occurs between threads:

accessing shared variables and the need to support atomicity for critical sections.

As it turns out, there is another common interaction that arises:

One thread must wait for another to complete some action before it continues.

This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.

We will be looking at how synchronization primitives and conditional variables help tame **uncontrolled scheduling**.

Why study this in an OS class?

Before wrapping up, one question that you might have is:

why are we studying this in OS class?

“History” is the one-word answer!

The OS was the **first concurrent program**, and many techniques were created for use within the OS. Later, with multi-threaded processes, application programmers also had to consider such things.

The End