System Calls

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

Copyright © 2024

UNIVERSITY OF CALGARY

Topics

- thread cancellation
- race conditions
- critical sections

Thread Cancellation

UNIVERSITY OF CALGARY

Thread/work cancellation

- imagine writing a program that detects whether a given word occurs anywhere in a set of files
  - i.e. as soon as the program detects the word in any file, it can stop the search

- we want to make the search faster, by using threads
  - we create multiple threads, each searching for the word in different files
  - as soon as one thread finds a file that contains the word,
    that thread should notify the other threads, so that they can stop searching

- two general approaches:
  - asynchronous cancellation
  - deferred cancellation (aka. synchronous cancellation)

UNIVERSITY OF CALGARY

Asynchronous thread/work cancellation

- one thread manually terminates the target thread, by calling `pthread_kill(tid, SIGUSR1)`
- target thread (`tid`) is killed nearly instantly
- what happens to data currently being updated by the target thread?
  - target thread has no chance to "clean up"
  - this can (likely) lead to leaving data in undefined state
  - for example, if the target thread is in the middle of allocating memory, the memory allocator could become corrupted and crash the entire program
- in many/most cases asynchronous cancellation is an unacceptable solution
- much better solution is to use synchronous thread cancellation

Deferred (Synchronous) thread/work cancellation

- the controlling thread somehow *indicates* it wishes to cancel a target thread (or the work in the thread)
  - e.g. by setting some shared global flag variable
  - or using `pthread_cancel()` and related mechanisms (`man pthread_cancel` for details)

- target thread periodically checks whether it should terminate
  - checking done only at *cancellation points*, where the thread can cancel itself safely
  - these are carefully chosen points, selected by the programmer

- some issues:
  - less performance – checking for cancellation flag requires at least 1 instruction…
  - target thread might not react immediately
    - it could run for a while before noticing the cancellation requested
    - e.g. continue to report results

- more flexible than asynchronous cancellation, but requires more effort to use (correctly)

UNIVERSITY OF CALGARY

Deferred cancellation example

```
void * thread_print(void * tid) {
  while(1) {
    printf("thread %ld running\n", tid);
    sleep(1);
    /* here we need to check if cancellation was requested */
  }
}
int main() {
  pthread_t threads[N_THREADS];
  for (long i = 0; i < N_THREADS; i++) {
    if( 0 != pthread_create(& threads[i], NULL, thread_print, (void *) i)) {
      printf("Oops, pthread_create failed.\n"); exit(-1);
    }
  }
  sleep(5); // pretend to do something
  /* here we request cancellation */
  for (long i = 0; i < N_THREADS; i++)
    pthread_join(threads[i], NULL);
  printf("All threads done.\n");
}
```

keeps printing message forever

note: without thread cancellation, this program will run forever

UNIVERSITY OF CALGARY

Deferred cancellation example (non-portable)

```
volatile int cancel_flag = 0;
void * thread_print(void * tid) {
  while(1) {
    printf("thread %ld running\n", tid);
    sleep(1);
    if(cancel_flag) return NULL;
  }
}
int main() {
  pthread_t threads[N_THREADS];
  for (long i = 0; i < N_THREADS; i++) {
    if( 0 != pthread_create(& threads[i], NULL, thread_print, (void *) i)) {
      printf("Oops, pthread_create failed.\n"); exit(-1);
    }
  }
  sleep(5); // pretend to do something
  cancel_flag = 1;
  for (long i = 0; i < N_THREADS; i++)
    pthread_join(threads[i], NULL);
  printf("All threads done.\n");
}
```

global flag

cancellation point: periodically checks global flag when it's safe

request cancellation by setting global flag

!!! non-portable code !!!

works on x86, but for portability we should use atomic operation e.g. `std::atomic<bool>`

UNIVERSITY OF CALGARY

Deferred cancellation example (C++, portable)

```
std::atomic_bool cancel_flag { false };

void thread_print(int tid) {
  while(1) {
    std::cout << "thread " << tid << " running\n";
    sleep(1);
    if(cancel_flag.load()) return;
  }
}

int main() {
  std::vector<std::thread> threads;
  for (long i = 0; i < N_THREADS; i++)
    threads.push_back( std::thread(thread_print,i));
  sleep(5); // pretend to do something
  cancel_flag.store(true);
  for( auto & t : threads )
    t.join();
  return 0;
}
```

global flag

`std::atomic<bool>` is portable, and will work on all architectures

cancellation point: periodically checks global flag when it's safe

request cancellation by setting global flag

https://repl.it/@pfederl/c-threads-with-cancellation

Race Conditions

UNIVERSITY OF CALGARY

Race conditions

- **race condition** is a behavior where the output is dependent on the sequence or timing of other uncontrollable events (eg. context switching, scheduling on multiple CPUs)
- race condition is a bug
- often a result of multiple processes/threads operating on a shared state/resource, eg.:
  - modifying shared memory
  - reading/writing to files
  - reading/writing to databases
- but not specific to multi-threaded applications
  - race conditions can exist among processes on the same computer; or even
  - among different computers using shared filesystems, databases, etc.

UNIVERSITY OF CALGARY

Race conditions

```
// global variable counter
int counter;

void incr() {
  // local variable x
  int x = counter;
  x = x + 1;
  counter = x;
}

int main() {
  counter = 0;
  incr();
  incr();

  printf( "%d\n", counter);
}
```

Output:

2

… every time

Race conditions

```
// global variable "counter" is shared
int counter;

void incr() {
  // local variable "x" is not shared
  int x = counter;
  x = x + 1;
  counter = x;
}

int main() {
  counter = 0;
  pthread_create(..., incr);
  pthread_create(..., incr);
  pthread_join ...
  printf( "counter = %d\n", counter);
}
```

Thread 1:

```
void incr() {
  int x = counter;
  x = x + 1;
  counter = x;
}
```

Thread 2:

```
void incr() {
  int x = counter;
  x = x + 1;
  counter = x;
}
```

What is the value in **counter** after both threads finish executing `incr()`?

UNIVERSITY OF CALGARY

Race conditions

```
// global variable "counter" is shared
int counter;

void incr() {
  // local variable "x" is not shared
  int x = counter;
  x = x + 1;
  counter = x;
}

int main() {
  counter = 0;
  pthread_create(..., incr);
  pthread_create(..., incr);
  pthread_join ...
  printf( "counter = %d\n", counter);
}
```

Thread 1	Thread 2	counter
		0
x = counter;		0
x = x + 1;		0
counter = x;		1
	x = counter;	1
	x = x + 1;	1
	counter = x;	2

one possible **execution sequence** resulting in

counter = 2

Race conditions

```
// global variable "counter" is shared
int counter;

void incr() {
  // local variable "x" is not shared
  int x = counter;
  x = x + 1;
  counter = x;
}

int main() {
  counter = 0;
  pthread_create(..., incr);
  pthread_create(..., incr);
  pthread_join ...
  printf( "counter = %d\n", counter);
}
```

Thread 1	Thread 2	counter
		0
x = counter;		0
	x = counter;	0
	x = x + 1;	0
	counter = x;	1
x = x + 1;		1
counter = x;		1

another possible **execution sequence** resulting in

counter = 1 !!!

This program has a race condition.

UNIVERSITY OF CALGARY

Race conditions

```
// global variable counter
int counter;

    void incr() {
      int x = counter;
      x = x + 1;
      counter = x;
    }
```

```
void incr() {
  counter ++;
}
```

Would this get rid of the race condition?

Can a single line of code be 'interrupted' by another thread?

```
int main() {
  counter = 0;
  pthread_create(..., incr);
  pthread_create(..., incr);
  pthread_join ...
  printf( "%d\n", counter);
}
```


UNIVERSITY OF CALGARY

Race conditions

```
int counter;

int incr1() {
    int x = counter;
    x = x + 1;
    counter = x;
}
```

```
mov eax, DWORD PTR counter[rip]
mov DWORD PTR [rbp-4], eax
add DWORD PTR [rbp-4], 1
mov eax, DWORD PTR [rbp-4]
mov DWORD PTR counter[rip], eax
```

```
int incr2() {
    counter ++;
}
```

```
mov eax, DWORD PTR counter[rip]
add eax, 1
mov DWORD PTR counter[rip], eax
```

To see how GCC compiles your code into assembly, you can try:

```
$ gcc -S -fverbose-asm test.c
```

Or use an online tool, eg: https://godbolt.org/z/WTPzC2 (full)

Race conditions

```
// global variable counter
int counter;

    void incr() {
      int x = counter;
      x = x + 1;
      counter = x;
    }
```

```
void incr() {
  counter ++;
}
```

```
int main() {
  counter = 0;
  pthread_create(..., incr);
  pthread_create(..., incr);
  pthread_join ...
  printf( "%d\n", counter);
}
```

Q: Would this get rid of the race condition?

A: No, because compiler might translate this into multiple instructions...

Q: But what if we compiled with **-O2** and compiler reduced this to a single instruction?

A: Race condition could still happen... multicore systems

UNIVERSITY OF CALGARY

Race conditions

- debugging race conditions is not fun
  - many test runs may produce the same output, often correct
  - then, in a rare situation the output might be different, e.g. when system was less/more busy
  - C example: https://repl.it/@pfederl/counter-race-condition
  - C++ example: https://repl.it/@pfederl/c-threads-with-race-condition

- we want to avoid race conditions
  - but how?

Concurrent programming

Avoiding race conditions

- we need to prevent more than one process/thread from accessing a shared resource at any given time
- approach:
  - identify **critical sections** in code where this could happen
  - enforce **mutual exclusion** to make sure it does not happen

UNIVERSITY OF CALGARY

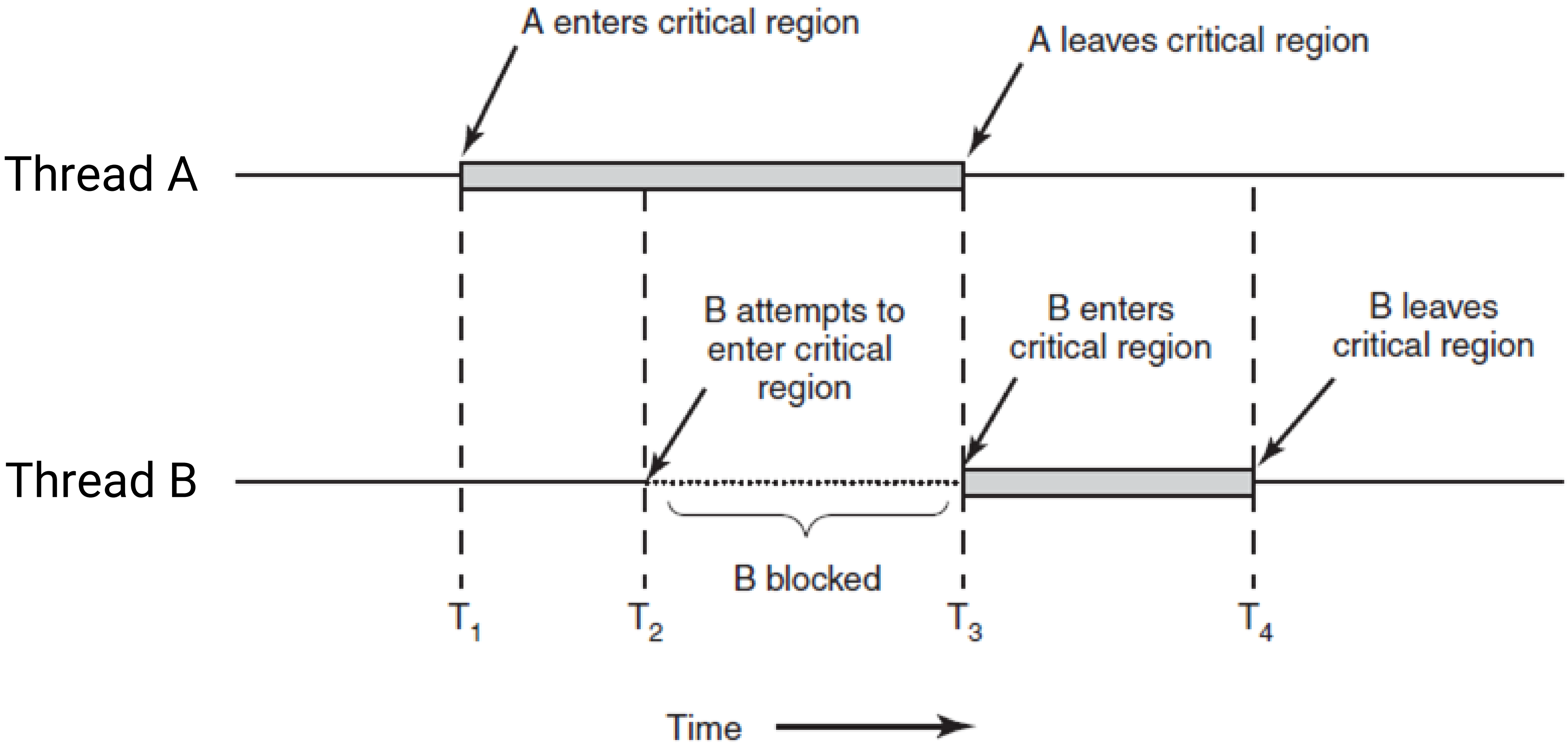
Critical sections and mutual exclusion

- **critical section** / **critical region**: part of the program that accesses the shared resource in a way that could lead to races or other undefined/unpredictable/unwanted behaviour

```
int counter;                      ← shared resource
void incr() {
  int x = counter;  ┐
  x = x + 1;        ├             ← critical section
  counter = x;      ┘
}
```

- if we can arrange tasks such that no two processes or threads will ever be in their critical sections at the same time, we could avoid the race condition (achieving **mutual exclusion**)

Critical sections and mutual exclusion



UNIVERSITY OF CALGARY

Requirements for good race-free solution

```
General structure:

while (1) {
  CS entry code
  critical section
  CS exit code
  non-critical section
}
```

1. **Mutual exclusion**: No two processes/threads may be simultaneously inside their critical sections (CS).
2. **Progress**: No process/threads running outside its CS may block other processes/threads.
3. **Bounded waiting**: No process/thread should have to wait forever to enter its CS.
4. **Speed**: No assumptions may be made about the speed or the number of CPUs.

Review

UNIVERSITY OF CALGARY

Summary

- thread cancellation
- race conditions
- critical sections

Threads and fork()

- is it ok to call fork() in a program with multiple threads?
  - what should happen?
  - what does happen?
- what actually happens:
  - only the calling thread survives, other threads are not duplicated
  - this creates a problem if synchronization mechanisms were used
  - it's possible to register a callback in case fork() is called using pthread_atfork()
- general advice: avoid using fork() in programs with multiple threads
- some usages are safe, eg.:
  - fork() is immediately followed by execve() to execute external program, or
  - fork() is executed before creating any threads

UNIVERSITY OF CALGARY

Onward to … locks, mutexes, dining philosophers

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY