CPSC 457

Concurrent programming

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

- threads and fork()
- 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

Thread cancellation

- scenario: parallelizing database search
- multiple threads searching different parts of the database
- one thread finds the result
- how do we notify the other threads to stop searching?

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

Asynchronous thread cancellation

- one thread manually terminates the target thread
  - `pthread_kill(threadid, SIGUSR1)`
  - target thread is killed instantly*
- common problem: what happens to the data currently updated by the thread that is killed?
  - killed thread has no chance to "clean up"
  - this can (likely) lead to leaving data in undefined state
- usually a much better solution is to use synchronous thread cancellation

Deferred / Synchronous thread cancellation

- controlling thread somehow *indicates* it wishes to cancel a thread
  - many options
  - eg. by setting some shared global flag
  - or use `pthread_cancel()` and related mechanisms (`man pthread_cancel` for details)
- target thread periodically checks whether it should terminate
  - checking done only at **cancellation points** at which it can be canceled safely
- some issues:
  - less performance
    - checking the cancellation flag requires at least 1 instruction…
  - target thread will not react immediately
    - may run for a while after cancellation requested
    - eg. could continue to report results for a while after cancellation request issued
- more flexible than asynchronous cancellation, but also more complex to implement

Deferred cancellation example


```
void * thread_print(void * tid) {
  while ( 1) {
    printf("thread %ld running\n", tid);
    sleep( 1);
  }
  pthread_exit(0); // not necessary
}
int main() {
  pthread_t threads[NUMBER_OF_THREADS];
  for (long i = 0; i < NUMBER_OF_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
  ???
  for (long i = 0; i < NUMBER_OF_THREADS; i++)
    pthread_join(threads[i], NULL);
  exit(0);
}
```

keep printing message forever

here we need to cancel the threads, otherwise this program will also run forever

Deferred cancellation example

```c
volatile int cancel_flag = 0;
void * thread_print(void * tid) {
  while(1) {
    printf("thread %ld running\n", tid);
    sleep(1);
    if(cancel_flag ) return;
  }
}
int main() {
  pthread_t threads[NUMBER_OF_THREADS];
  for (long i = 0; i < NUMBER_OF_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 somethi
  cancel_flag = 1;
  for (long i = 0; i < NUMBER_OF_THREADS; i++)
    pthread_join(threads[i], NULL);
  exit(0);
}
```

global flag

works fine on x86, but should use an atomic type / operations eg. std::atomic<bool> for portability

periodically check the global flag (cancellation point)

set global flag to request cancellation

Race conditions

```c
// global variable counter
int counter;

void incr() {
  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
int counter;

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

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

Thread 1:	Thread 2:

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

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

Race conditions


```
// global variable counter
int counter;

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

int main() {
  counter = 0;
  pthread_create(..., incr);
  pthread_create(..., incr);
  pthread_join ...
  printf( "%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

possible **execution sequence**
leading to counter = 2

Race conditions

```
// global variable counter
int counter;

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

int main() {
  counter = 0;
  pthread_create(..., incr);
  pthread_create(..., incr);
  pthread_join ...
  printf( "%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 leading to **counter = 1 !!!**

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?

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

Race conditions

```c
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
```

```c
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 instructions, you can try:

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

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

Race conditions in software

- **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)
- often a result of multiple processes/threads operating on a shared state/resource, eg.:
  - modifying shared memory
  - reading/writing to files
  - modifying filesystems
  - reading/writing to databases
- debugging race conditions is not fun
  - many test runs may produce the same output, often correct
    https://repl.it/@pavolfederl/thread-counter-race-condition
  - then, in a rare situation the output might be different, eg. when system was less/more busy
- we want to avoid race conditions
  - how?

Avoiding race conditions

- we need to prevent more than one process/thread from accessing the 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

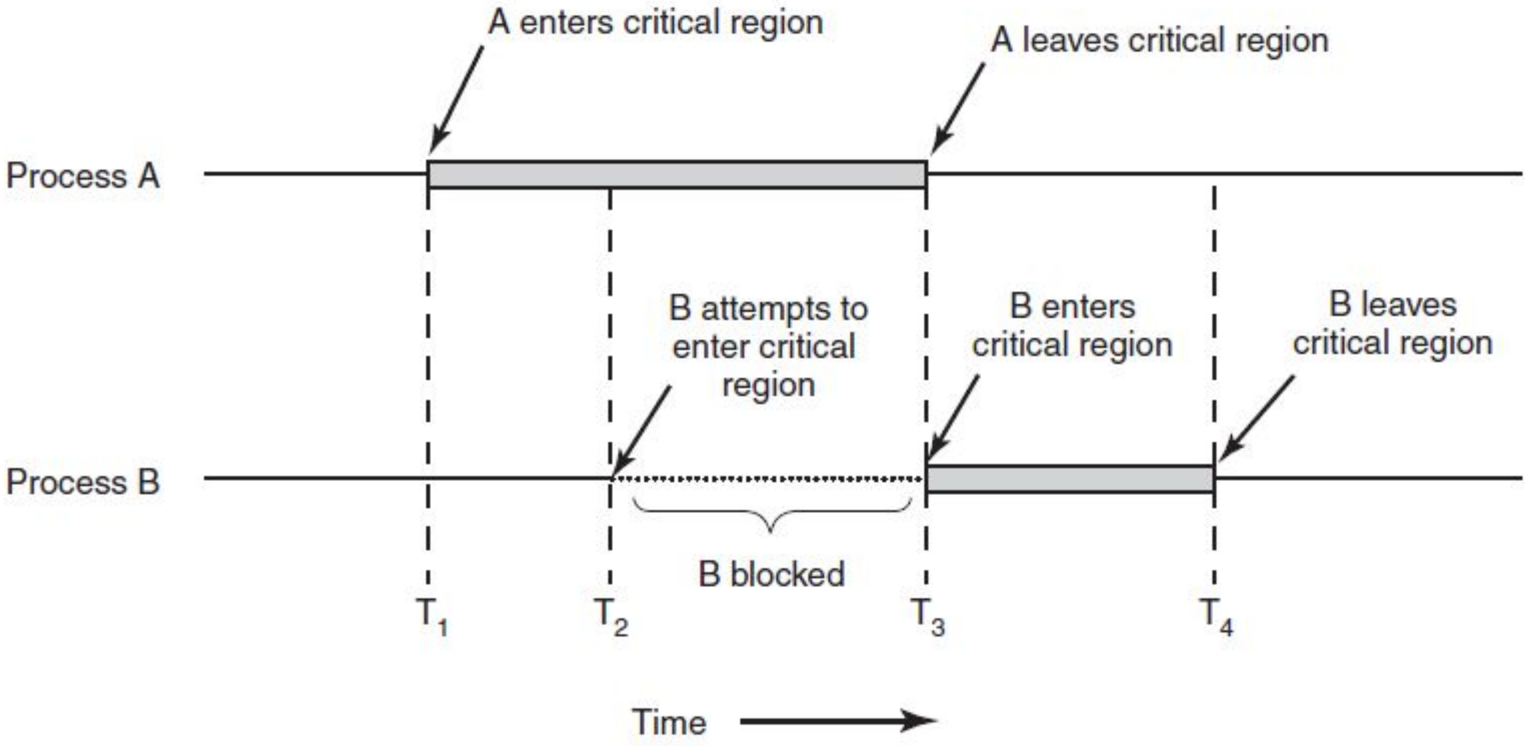
Critical sections and mutual exclusion

- **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



Critical sections

- requirements to avoid race conditions:
  - no two processes may be simultaneously inside their critical sections
  - no assumptions may be made about the speeds or the number of CPUs
  - no process running outside its critical region may block other processes
  - no process should have to wait forever to enter its critical section

Questions?