G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Mutexes and Critical Sections II

Dr Jason Atkin

Previous lectures

- Creating processes and threads
  - Creating windows programs
  - Event loops and windows messages
  - Sharing memory between processes
- Process traces
  - Tracing the possible orders of execution
    - Do all possibilities work?
- Atomic Operations
- Spin-locks

Round-robin algorithm

Variable: *turn*: integer variable, initialised to 1, **volatile**

- **Thread 1:**

init

Entry protocol:

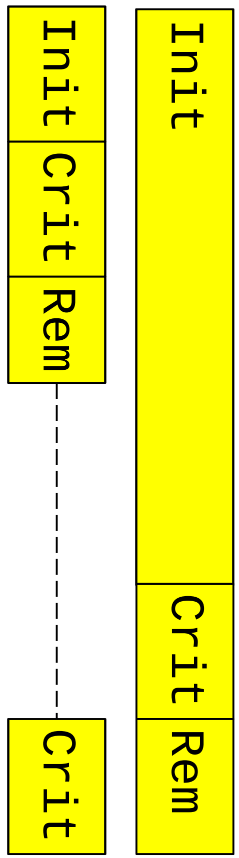
while (*turn* != 1) ;

crit

Exit protocol:

turn = 2;

rem



- **Thread 2:**

init

Entry protocol:

while (*turn* != 2) ;

crit

Exit protocol:

turn = 1;

rem

Shared turn variable. Works but can have unnecessary delays – have to wait for the other one to act (take it in turns). Problem is init or rem for the other one is too long

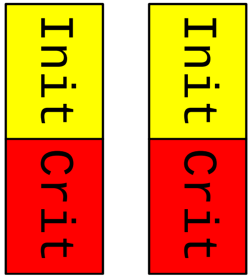
A simple spin lock

```
bool lock = false;  // shared lock variable

// Process i
init_i;
while(true) {
  while(lock) {};   // entry protocol
  lock = true;      // entry protocol
  crit_i;
  lock = false;     // exit protocol
  rem_i;
}
```

Shared variable for the lock.
Does not ensure mutual exclusion – they can both look at the variable before either checks it (see next slide)

Spin-lock example trace

```
// Process 1

init1;
while(true) {
    while(lock)

}
```




```
// Process 2

init2;
while(true) {
    while(lock)

}
```

`lock == false`

Test-and-Set instruction

The Test-and-Set (atomic) instruction effectively executes the function

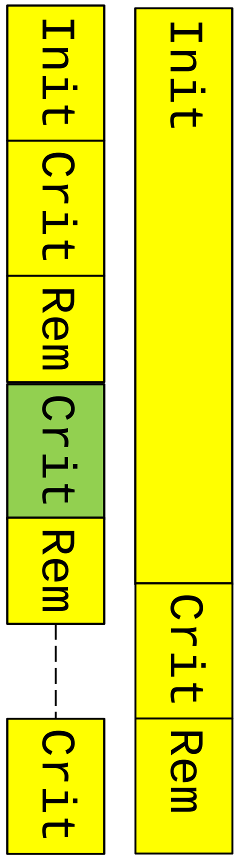
```
bool TS(bool lock)
{
    bool v = lock;
    lock = true; // Set true
    return v; // Old lock value
}
```

Stops the other thread looking at it before we set it

See InterlockedExchange: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683590%28v=vs.85%29.aspx

Example of Test-And-Set spin-lock

```
// Process 1

init1;
while(true) {
    while(TS(lock)) ;
    crit1;
    lock = false;
    rem1;
}
```




```
// Process 2

init2;
while(true) {
    while(TS(lock)) ;
    crit2;
}
```

Use two variables?

```
// Process 1
init1;

while(true)
{
    c1 = 0; // entry protocol
    while (c2 == 0)
        ;
    crit1;
    c1 = 1; // exit protocol
    rem1;
}
```

```
// Process 2
init2;

while(true)
{
    c2 = 0; // entry protocol
    while (c1 == 0)
        ;
    crit2;
    c2 = 1; // exit protocol
    rem2;
}
```

`c1 == 0`   `c2 == 0`

Aim: Avoid the need for a special atomic operation.
2 variables, c1 and c2. Each thread has its own variable and is the only thread to alter that variable. Sets it to 0 when wanting to enter.

Dekker's algorithm

```
// Process 1
init1;
while(true) {
    c1 = 0;   // entry protocol
    while (c2 == 0) {
        if (turn == 2) {
            c1 = 1;
            while (turn == 2) {};
            c1 = 0;
        }
    }
    crit1;
    turn = 2; // exit protocol
    c1 = 1;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    c2 = 0;   // entry protocol
    while (c1 == 0) {
        if (turn == 1) {
            c2 = 1;
            while (turn == 1) {};
            c2 = 0;
        }
    }
    crit2;
    turn = 1; // exit protocol
    c2 = 1;
    rem2;
}
```

c1 == 1 c2 == 1 turn == 1

This lecture

- Improved (entry and exit) protocols

- Today:
  - Peterson’s algorithm
  - Operating system support
    - Mutex and CriticalSection objects
  - Disadvantages of critical sections

Peterson's algorithm

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
// shared variables
bool c1 = c2 = false;
integer turn = 1;
```

Peterson's algorithm: trace (1)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = false;    integer turn = 1;    bool c2 = false;
```

Peterson's algorithm: trace (2)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = false;     integer turn = 1;     bool c2 = false;
```

Peterson's algorithm: trace (3)

```
// Process 1
init1;
while(true) {
    // entry protocol
➔   c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
➔ init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;      integer turn = 1;      bool c2 = false;
```

Peterson’s algorithm: trace (4)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;     integer turn = 2;     bool c2 = false;
```

Peterson's algorithm: trace (5)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;    integer turn = 2;    bool c2 = false;
```

Peterson's algorithm: trace (6)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;      integer turn = 2;      bool c2 = false;
```

Peterson's algorithm: trace (7)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;     integer turn = 2;     bool c2 = true;
```

Peterson's algorithm: trace (8)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;      integer turn = 1;      bool c2 = true;
```

Peterson’s algorithm: trace (9)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
→   crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
→   while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;      integer turn = 1;      bool c2 = true;
```

Peterson's algorithm: trace (10)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = false;    integer turn = 1;    bool c2 = true;
```

Peterson’s algorithm: trace (11)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = false;    integer turn = 1;    bool c2 = true;
```

Peterson's algorithm: trace (12)

```
// Process 1
init1;
while(true) {
    // entry protocol
→   c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
→   crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;     integer turn = 1;     bool c2 = true;
```

Peterson’s algorithm: trace (13)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
➔   turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
➔   crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;     integer turn = 2;     bool c2 = true;
```

Peterson's algorithm: trace (14)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
➔   while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
➔   crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;     integer turn = 2;     bool c2 = true;
```

Peterson's algorithm: trace (2)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn
    while
    {};
    crit1
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
                        n == 1)
    // exit protocol
    c2 = false;
    rem2;
}
```

Assume process 1 is swapped out or for some reason not active for a bit…

```
bool c1 = true;      integer turn = 2;      bool c2 = true;
```

Peterson's algorithm: trace (15)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
➔   while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
➔   c2 = false;
    rem2;
}
```

```
bool c1 = true;      integer turn = 2;      bool c2 = false;
```

Peterson's algorithm: trace (16)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
➔   while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
➔   rem2;
}
```

```
bool c1 = true;     integer turn = 2;     bool c2 = false;
```

Peterson’s algorithm: trace (17)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
➔   while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
➔   c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;        integer turn = 2;        bool c2 = true;
```

Peterson's algorithm: trace (18)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit protocol
    c1 = false;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    crit2;
    // exit protocol
    c2 = false;
    rem2;
}
```

```
bool c1 = true;     integer turn = 1;     bool c2 = true;
```

Peterson's algorithm: trace (19)

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = true;
    turn = 2;
    while (c2 && turn == 2)
    {};
    crit1;
    // exit
    c1 = fa
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = true;
    turn = 1;
    while (c1 && turn == 1)
    {};
    ol
```

Process 2 was the last one to enter the critical section entry protocol, so process 1 will get the option to go next

```
bool c1 = true;      integer turn = 1;      bool c2 = true;
```

Properties of Peterson's algorithm

- The solution based on Peterson's algorithm has the following properties:
  - **Mutual Exclusion:** yes
  - **Absence of Livelock:** yes
  - **Absence of Unnecessary Delay:** yes
  - **Eventual Entry:** is guaranteed even if scheduling policy is only *weakly fair*.
- A weakly fair scheduling policy guarantees that if a process requests to enter its critical section (**and does not withdraw the request**), the process will *eventually* enter its critical section

Operating system support

- Operating systems often provide support for:
  - Mutual exclusion
  - Semaphores (lock counting mechanism, later lectures)
- Advantages:
  - Can suspend thread – no spinning
  - Many allow a thread to ‘lock’ multiple times without problems
  - Operating system can detect if the process dies and free resource
- Windows Mutual exclusion:
  - Mutex: Windows objects sharable between processes
  - CriticalSection: light-weight lock, available to threads **within a single** process. Designed for speed, allowing some spinning too
- Linux:
  - Uses the mutex support within pthreads
  - Easy for threads in a single process
  - Can be used cross-process by putting mutex in shared memory

CriticalSection objects

- Will only work within the same process
- Fast – only switch to kernel mode and wait if there is contention, otherwise see that it is available and continue
- The thread owns the critical section once it gets it
  - Further requests by the same thread automatically succeed – it already has it
  - Must match number of ‘leave’ and ‘enter’ to release it

1. Process creates object: CRITICAL_SECTION MyCriticalSection;
2. Process initialises it: InitializeCriticalSection()
3. Thread requests entry: EnterCriticalSection(), TryEnterCriticalSection()
4. Thread leaves critical section: LeaveCriticalSection()
5. Process deletes it: DeleteCriticalSection()

- Parameter will usually be the address of the critical section object which you created, e.g. &MyCriticalSection

CriticalSection objects : more info

More information: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682530%28v=vs.85%29.aspx

- For speed efficiency, you can tell it to 'spin' first, continuously checking for a while to see whether lock becomes available almost immediately, see:
  - InitializeCriticalSectionAndSpinCount, SetCriticalSectionSpinCount
- Warnings:
  - "If a thread terminates while it has ownership of a critical section, the state of the critical section is undefined"
  - "If a critical section is deleted while it is still owned, the state of the threads waiting for ownership of the deleted critical section is undefined"
  - Source https://msdn.microsoft.com/en-us/library/windows/desktop/ms682608%28v=vs.85%29.aspx

Sample code : variable and using it

```
// Global variable for locking
CRITICAL_SECTION MyCriticalSection;

// Function which wants to use it
DWORD WINAPI ThreadProc( LPVOID lpParm )
{
      …
      EnterCriticalSection( &MyCriticalSection );
      …
      // Do something within critical section
      …
      LeaveCriticalSection( &MyCriticalSection );
      …
}
```

Sample : initialisation and deletion

```
int main()
{
   // Initialize the critical section one time only.
   if ( !InitializeCriticalSectionAndSpinCount(
          &MyCriticalSection, 1024 ) )
   { … Handle error happened … }

   // Start threads and do things?
   …
   // Wait for completion of all operations

   // Release resources used by CriticalSection object
   DeleteCriticalSection( &MyCriticalSection );
}
```

Mutex objects

- A windows object whose state is set to signaled when NOT owned by a thread
  - Often slower than a CRITICAL_SECTION function because it has to ask the kernel every time
  - Treated like any other handle to a kernel object
- Again there is a lot of information on the MSDN:
  - Mutex Objects: https://msdn.microsoft.com/en-gb/library/windows/desktop/ms684266%28v=vs.85%29.aspx
- We already know how to see whether the object associated with a handle is signalled:
  - WaitForSingleObject() or WaitForMultipleObjects()
  - Thread handles are signalled when thread dies
  - Process handles are signalled when process dies
- Create/release using CreateMutex and ReleaseMutex

Mutex example : getting a lock

```
// Global variable for locking
HANDLE MutexA;

DWORD WINAPI ThreadProc( LPVOID lpParm )
{
   // Request ownership of the mutex
   switch( WaitForSingleObject( MutexA, INFINITE ) )
      // handle to mutex, timeout interval
   {
      // The thread got ownership of the mutex
      case WAIT_OBJECT_0:
        // Access the shared resource.
        … Do something here …
        // Release ownership of the mutex object
        if ( !ReleaseMutex( MutexA ) )
        { /* Handle error. */ }
        break;
```

Mutex example : error values

```
    // The thread got ownership of an abandoned mutex
    case WAIT_ABANDONED:
       /* Handle the problem */
       return FALSE;
    // Timed out - if we had added time limit
    case WAIT_TIMEOUT:
       /* Handle the problem */
       return FALSE;
    // Function failed for some reason
    case WAIT_FAILED:
       /* Handle the problem */
       return FALSE;
  } // End of switch on return value

  // Remainder - any other stuff we need to do
}
```

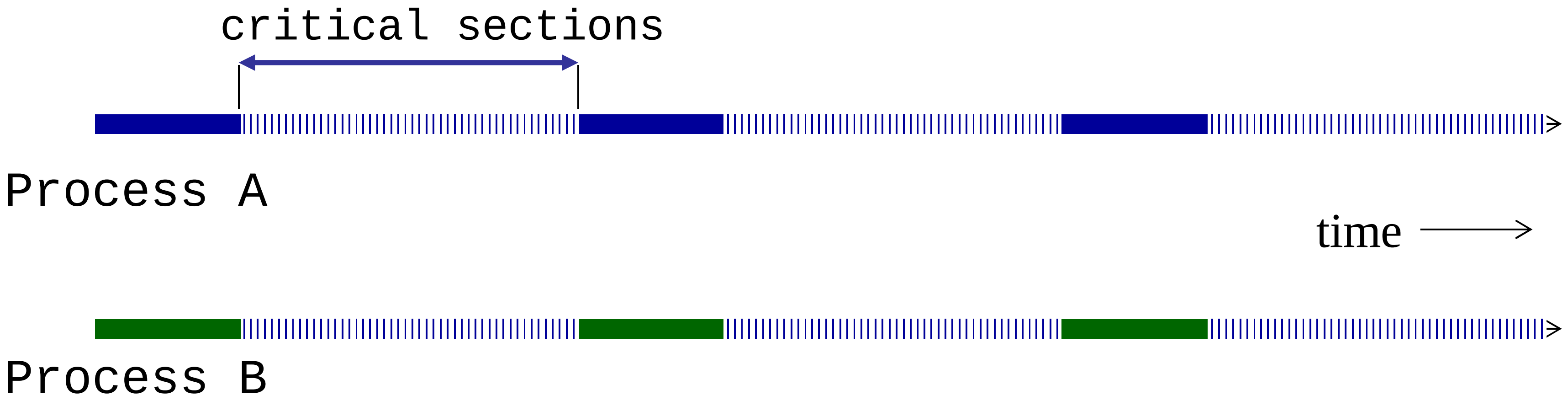
Mutex example

```
int main()
{
   // Create the mutex – one time only
   MutexA = CreateMutex(
            NULL,    // default security attributes
            FALSE,   // initially not owned
            "JasonsMutexA" ); // Name – your choice

   // Start threads and do things?
   …
   // Wait for completion of all operations

   // Release the mutex
   ReleaseMutex( MutexA );
}
```

Extra Mutex information

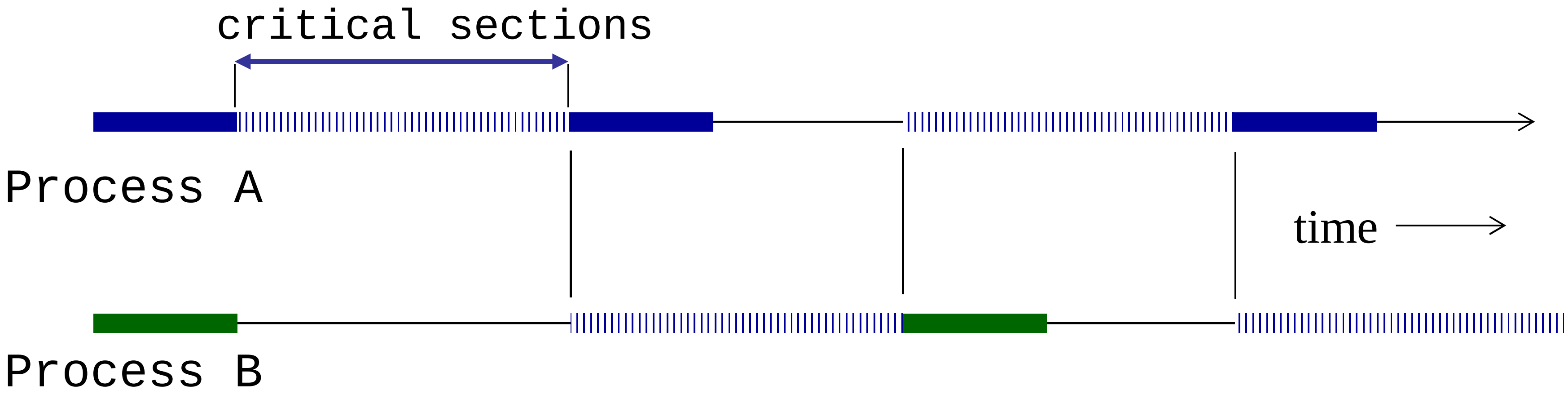
- A thread owns a mutex
  - Further calls to lock it will succeed rather than deadlocking it
  - Needs to release it multiple times then though
- A random waiting thread is selected when multiple threads are waiting – not FIFO!!!
- If a thread terminates without releasing its ownership of a mutex object, the mutex object is considered to be abandoned
  - https://msdn.microsoft.com/en-gb/library/windows/desktop/ms684266%28v=vs.85%29.aspx
  - Releasing the mutex will return it to normal

The problems of critical sections



- Consider two processes which have long critical sections
- When you enforce mutual exclusion on the critical sections the time taken may increase…

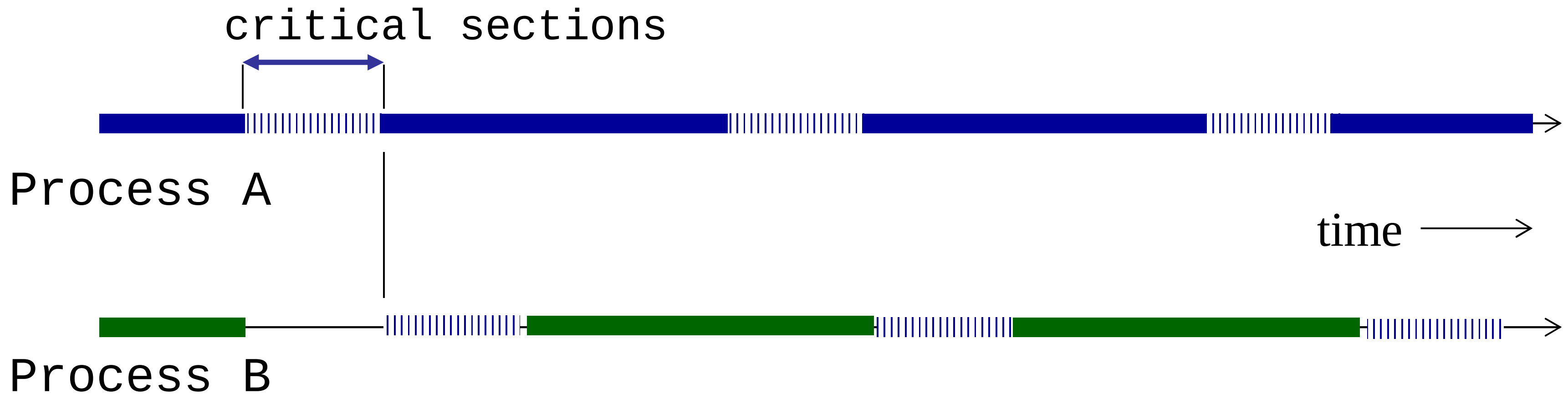
The problems of critical sections



- Higher proportions of critical sections compared with other code will increase the duration further – wasting time for one or more threads/processes

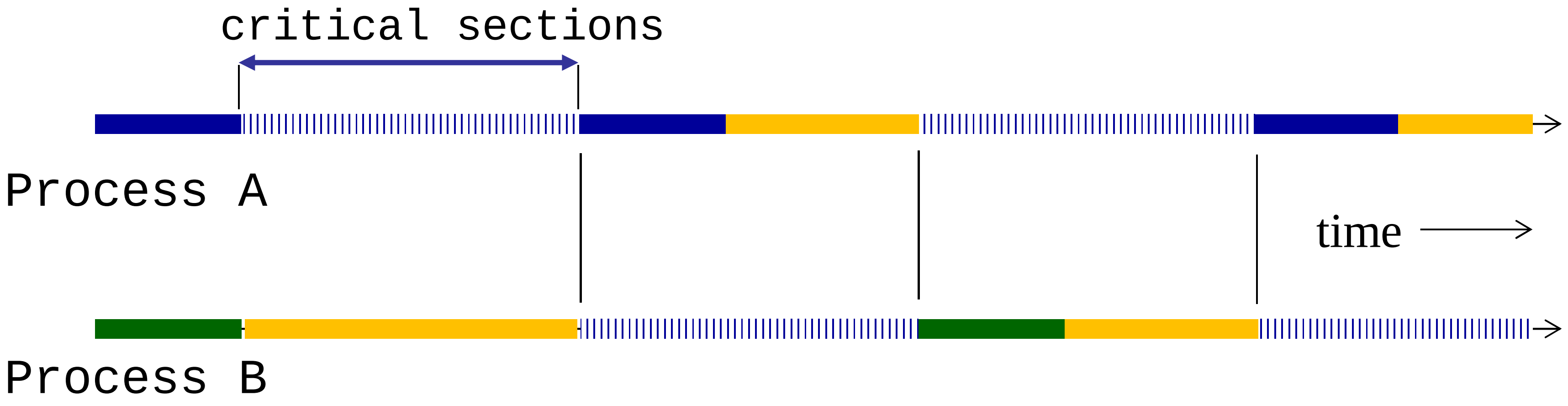
- When you enforce mutual exclusion on the critical sections the time taken may increase…

The problems of critical sections



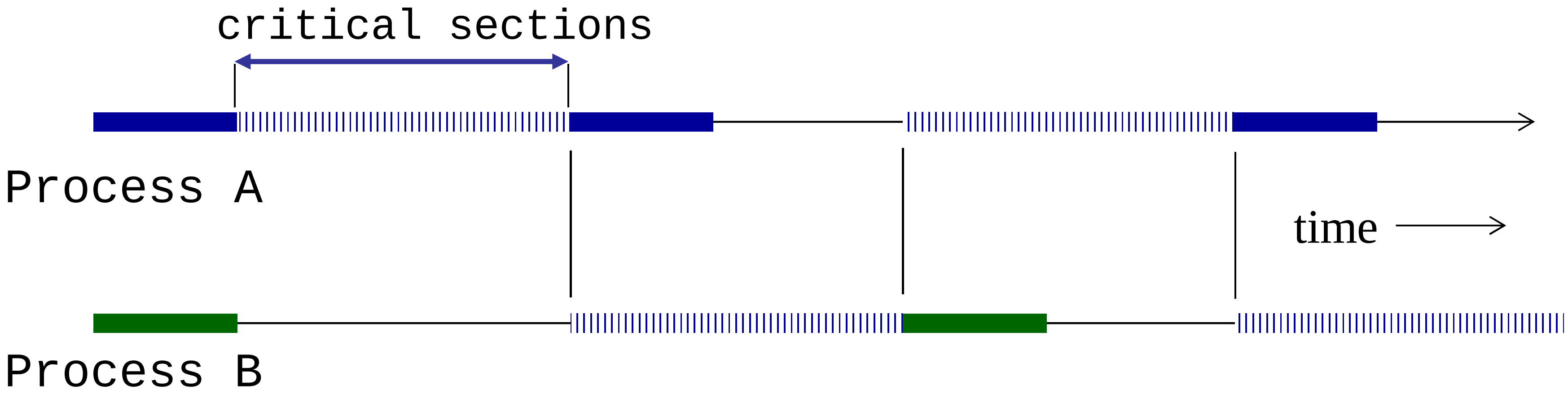
- With short critical sections, the time to execute may be affected very little (or not at all?)

Spin locks are even worse



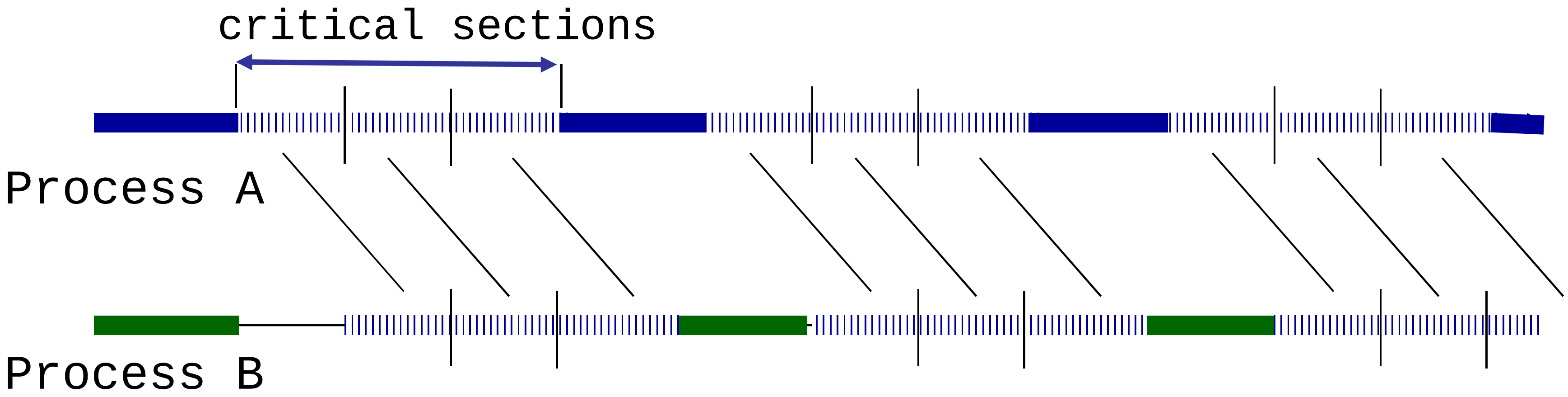
- With spin-locks it is even worse!
- Scheduler does not know that the thread cannot continue so it will be assigned CPU time:
- At least critical section and mutex objects allow the thread to ‘sleep’ until the critical section becomes available

The problems of critical sections



- Assume that there are three shared variables, set one after another, by both processes, e.g.
  - **`var1++`**
  - **`var2++`**
  - **`var3++`**
- A single long critical section may not be the best thing to use – you could potentially use three classes…

The problems of critical sections



- Each critical section only ensures mutual exclusion against the critical sections of the same type
  - i.e. same mutex or critical_section object
- Operations in different types of critical sections can take place simultaneously
- The system **may** execute a **lot** faster as a result of this

Example: 3 critical sections

```
// Global variables for locking
// Assume these are initialised and released appropriately
CRITICAL_SECTION CriticalSection1;
CRITICAL_SECTION CriticalSection2;
CRITICAL_SECTION CriticalSection3;

// Code to use the critical sections:
EnterCriticalSection( &CriticalSection1 );
++dwValue1; // Access the shared resource.
LeaveCriticalSection( &CriticalSection1 );
EnterCriticalSection( &CriticalSection2 );
++dwValue2; // Access the shared resource.
LeaveCriticalSection( &CriticalSection2 );
EnterCriticalSection( &CriticalSection3 );
++dwValue3; // Access the shared resource.
LeaveCriticalSection( &CriticalSection3 );
```

Next Lecture

- Semaphores
- Shared queues (as arrays)
- Using sockets for inter-process communication