Concurrency Primitives

Soham Chakraborty

11.02.2022

Outline

Recap

- Previous example

Concurrency Primitives

- Synchronization
- C/C++ primitives
- Mutex & Lock

Properties & Reasoning

Recap: Previous Example

$$Init$$

$$\begin{array}{l||l} \text{lock}(m) & \text{lock}(m) \\ S_1; & S_2; \\ \text{unlock}(m) & \text{unlock}(m) \end{array}$$

$$\mathit{Init}$$

$$\begin{array}{l|l} \text{lock}(m) & \text{lock}(m) \\ S_1; & S_2; \\ \text{unlock}(m) & \text{unlock}(m) \end{array}$$

Lock/Unlock

- How is lock/unlock defined?
- What are the properties?

$$Init$$

lock(m)	$\|\|$	lock(m)
critical section		critical section
unlock(m)		unlock(m)

Mutual Excellence

Init

lock(m)	lock(m)
critical section	critical section
unlock(m)	unlock(m)

m: lock object

critical section: code block between lock(m) and unlock(m)

Mutual execution property: critical sections can be executed by only one thread at a time

Thread Communication & Synchronization

Threads communicate to communicate information

$$*X = NULL, flag = 0;$$

$$\begin{array}{l||l} X = new\ \ Obj(); & while(flag \neq 1) \ \text{// } waiting.... \\ flag = 1; & \quad ; \\ & t = X; \ \text{// must be non-NULL} \end{array}$$

What happens if we reorder the statements in the first thread?

$$*X = NULL, flag = 0; \qquad\qquad\qquad *X = NULL, Y = 0;$$

$$\begin{array}{l} X = new\ Obj(); \\ flag = 1; \end{array} \left\| \begin{array}{l} while(flag \neq 1) \\ \quad ; \\ t = X; \end{array} \right. \quad \rightsquigarrow \quad \begin{array}{l} flag = 1; \\ X = new\ Obj(); \end{array} \left\| \begin{array}{l} while(flag \neq 1) \\ \quad ; \\ t = X; \end{array} \right.$$

t = NULL is NOT possible

t = NULL is possible

$$Init$$

$$\begin{array}{l|l} \text{lock}(m); & \text{lock}(m); \\ X = X + 1; & X = X + 1; \\ \text{unlock}(m); & \text{unlock}(m); \end{array}$$

Can we reorder instructions with lock/unlock?

- X and m are independent variables

Reordering with Lock/Unlock

Init

$$
\begin{array}{l||l}
\textcolor{blue}{\mathsf{lock}}(m); & \textcolor{blue}{\mathsf{lock}}(m); \\
\textcolor{blue}{\mathsf{unlock}}(m); & X = X + 1; \\
X = X + 1; & \textcolor{blue}{\mathsf{unlock}}(m);
\end{array}
$$

Is it problematic?

Reordering with Lock/Unlock

$$\textit{Init}$$

$$\begin{array}{l||l}
\mathsf{lock}(m); & \mathsf{lock}(m); \\
\mathsf{unlock}(m); & X = X + 1; \\
X = X + 1; & \mathsf{unlock}(m);
\end{array}$$

Is it problematic?

Safe reordering in sequential program
may be unsafe for concurrency

Roach Motel Reordering

Safe

S_1

$lock(m)$

$unlock(m)$

S_2

Unsafe

$lock(m)$

S_1

S_2

$unlock(m)$

How to restrict these reorderings?

- Concurrency primitives

C/C++ Concurrency Primitives

Introduced in 2011 C/C++ standard.

Provides platform independent abstraction

Consistency rules.

Shared Memory Accesses

Non-atomic accesses: Read (Ld), Write (St)

Atomic accesses = operation + memory order

Operations:

- Read (Ld)
- Write (St)
- Atomic update (U)
- Fence (F)

Memory orders:

- Relaxed (rlx)
- Release (rel)
- Acquire (acq)
- Acquire-Release (acq_rel)
- Sequentially consistent (sc)

Shared Memory Accesses

Non-atomic accesses: Read (Ld), Write (St)

Atomic accesses = operation + memory order

Operations:

- Read (Ld)
- Write (St)
- Atomic update (U)
- Fence (F)

Memory orders:

- Relaxed (rlx)
- Release (rel)
- Acquire (acq)
- Acquire-Release (acq_rel)
- Sequentially consistent (sc)

Example:

- X.load(memory order)
- X.store(val, memory order)
- X.CAS(oldval, nwval, success mem order, failure mem order)
- atomic_thread_fence(memory order)

Access Types

Read. $t = X_o$
where $o \in \{\text{na}, \text{rlx}, \text{acq}, \text{sc}\}$

Write. $X_o = v$
where $o \in \{\text{na}, \text{rlx}, \text{acq}, \text{sc}\}$

Update. $CAS(X, v, v', o_s, o_f)$
where $o_s, o_f \in \{\text{rlx}, \text{rel}, \text{acq}, \text{acq\_rel}, \text{U}_{\text{sc}}\}$

Fence F_o
where $o \in \{\text{rel}, \text{acq}, \text{acq\_rel}, \text{sc}\}$

For now we consider only sc accesses

Reordeing Rules

$a(\ell) \Downarrow / b(\ell') \Rightarrow$	$\mathsf{Ld}_{\mathsf{na}}/\mathsf{St}_{\mathsf{na}}$	$\mathsf{St}_{\mathsf{sc}}$	$\mathsf{Ld}_{\mathsf{sc}}$
$\mathsf{Ld}_{\mathsf{na}}/\mathsf{St}_{\mathsf{na}}$	✓	✗	✓
$\mathsf{St}_{\mathsf{sc}}$	✓	✗	✗
$\mathsf{Ld}_{\mathsf{sc}}$	✗	✗	✗

$a; b \rightsquigarrow b; a$ where $\ell \neq \ell'$ and are independent

Reordeing Rules

$a(\ell) \Downarrow / b(\ell') \Rightarrow$	$\mathsf{Ld}_{na}/\mathsf{St}_{na}$	St_{sc}	Ld_{sc}
$\mathsf{Ld}_{na}/\mathsf{St}_{na}$	✓	✗	✓
St_{sc}	✓	✗	✗
Ld_{sc}	✗	✗	✗

$a; b \rightsquigarrow b; a$ where $\ell \neq \ell'$ and are independent

Revisiting the (simplified) synchronization example:

$$
\begin{array}{l}
\mathit{int}\ X = 0, \\
\mathit{atomic\_int}\ \mathit{flag} = 0; \quad \text{// all accesses are sc}
\end{array}
$$

$$
\begin{array}{l||l}
X = 1; & \mathit{while}(\mathit{flag} \neq 1) \\
 & \quad ; \\
\mathit{flag} = 1; & t = X;
\end{array}
$$

Reordeing Rules

$a(\ell) \Downarrow / b(\ell') \Rightarrow$	$\mathsf{Ld_{na}}/\mathsf{St_{na}}$	$\mathsf{St_{sc}}$	$\mathsf{Ld_{sc}}$
$\mathsf{Ld_{na}}/\mathsf{St_{na}}$	✓	✗	✓
$\mathsf{St_{sc}}$	✓	✗	✗
$\mathsf{Ld_{sc}}$	✗	✗	✗

$a; b \rightsquigarrow b; a$ where $\ell \neq \ell'$ and are independent

Revisiting the (simplified) synchronization example:

$$\begin{array}{l} \mathit{int}\ X = 0, \\ \mathit{atomic\_int}\ \mathit{flag} = 0; \ \textit{// all accesses are sc} \end{array}$$

$$\begin{array}{l|l} X = 1; & \mathit{while}(\mathit{flag} \neq 1) \\ & \quad ; \\ \mathit{flag} = 1; & t = X; \end{array}$$

Note: the atomic accesses can be used to implement lock & unlock

```
void lock(){
   i = tid();
   j = 1 − i;
   flag[i] = true;   // I'm interested
   while(flag[j]) {}  // wait loop
}
```

```
unlock() {
   i = tid();
   flag[i] = false;   // I'm not interested
}
```

Ensures mutual execlusion (Critical sections do not overlap)

Well-formedness

A thread is well-formed if:

1. each critical section is associated with a unique lock object.
2. the thread calls lock method for that object when it is trying to enter the critical section, and
3. the thread calls the unlock method for that object when it leaves the critical section.

Reasoning

Threads are state machines: $a_0 \to a_1 \to \ldots$

Thread transitions are events

(a_0, a_1): Interval betwwen events a_0 and a_1

$I_A = (a_0, a_1)$: interval between a_0 and a_1 in thread A
$I_B = (b_0, b_1)$: interval between a_0 and a_1 in thread B

$I_A \to I_B$: interval I_A precedes I_B; when $a_1 \to b_0$

a_i^j : j^{th} occurrence of an event a_i

I_A^j : j^{th} occurrence of an interval I_A

Mutual Exclusion

Critical sections do not overlap.

Given thread A and B

Given the intervals CS_A^i and CS_B^j:

either $CS_A^i \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^i$

$$
\begin{array}{l}
void\ \ lock()\{ \\
\quad i = tid(); \\
\quad j = 1 - i; \\
\quad flag[i] = true; \\
\quad while(flag[j])\ \{\} \\
\}
\end{array}
$$

$$
\begin{array}{l}
unlock()\ \{ \\
\quad i = tid(); \\
\quad flag[i] = false; \\
\}
\end{array}
$$

Violation of Mutual Exclusion

$$CS_0^j \not\rightarrow CS_1^k \text{ and } CS_1^k \not\rightarrow CS_0^j$$

```
void lock(){
   i = tid();
   j = 1 - i;
   flag[i] = true;
   while(flag[j]) {}
}

unlock() {
   i = tid();
   flag[i] = false;
}
```

$$
\begin{aligned}
&void\ lock()\{\\
&\quad i = tid();\\
&\quad j = 1 - i;\\
&\quad flag[i] = true;\\
&\quad while(flag[j])\ \{\}\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&unlock()\ \{\\
&\quad i = tid();\\
&\quad flag[i] = false;\\
&\}
\end{aligned}
$$

$CS_0^j \not\rightarrow CS_1^k$ and $CS_1^k \not\rightarrow CS_0^j$

$$
\begin{tikzcd}
\text{a: } \mathrm{W}_0(flag[0], true) \arrow[d] \\
\text{b: } \mathrm{R}_0(flag[1], false) \arrow[d] \\
CS_0
\end{tikzcd}
$$

Violation of Mutual Exclusion

$void\ lock()\{$
$\quad i = tid();$
$\quad j = 1 - i;$
$\quad flag[i] = true;$
$\quad while(flag[j])\ \{\}$
$\}$

$unlock()\ \{$
$\quad i = tid();$
$\quad flag[i] = false;$
$\}$

$CS_0^j \not\rightarrow CS_1^k$ and $CS_1^k \not\rightarrow CS_0^j$

$$\begin{array}{ccc} \text{a: } W_0(flag[0], true) & & \text{c: } W_1(flag[1], true) \\ \downarrow & & \downarrow \\ \text{b: } R_0(flag[1], false) & & \text{d: } R_1(flag[0], false) \\ \downarrow & & \downarrow \\ CS_0 & & CS_1 \end{array}$$

Violation of Mutual Exclusion

```
void lock(){
   i = tid();
   j = 1 - i;
   flag[i] = true;
   while(flag[j]) {}
}


unlock() {
   i = tid();
   flag[i] = false;
}
```

$CS_0^j \not\rightarrow CS_1^k$ and $CS_1^k \not\rightarrow CS_0^j$

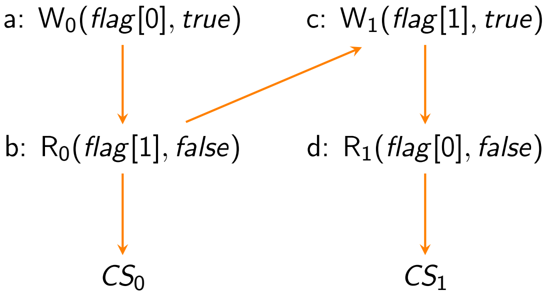
$$\begin{tikzcd}
\mathrm{a:}\ \mathrm{W}_0(\mathit{flag}[0], \mathit{true}) \arrow[d] & \mathrm{c:}\ \mathrm{W}_1(\mathit{flag}[1], \mathit{true}) \arrow[d] \\
\mathrm{b:}\ \mathrm{R}_0(\mathit{flag}[1], \mathit{false}) \arrow[d] \arrow[ur] & \mathrm{d:}\ \mathrm{R}_1(\mathit{flag}[0], \mathit{false}) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}$$

Violation of Mutual Exclusion

```
void lock(){
    i = tid();
    j = 1 − i;
    flag[i] = true;
    while(flag[j]) {}
}


unlock() {
    i = tid();
    flag[i] = false;
}
```

$CS_0^j \not\to CS_1^k$ and $CS_1^k \not\to CS_0^j$



Contradiction: $a \to d$ and no intermediate $W(flag[0], false)$ between a and b.

Alternative Lock Unlock

$$
\begin{array}{l}
\textit{void lock}()\{ \\
\quad i = \textit{tid}(); \\
\quad \textit{victim} = i; \quad \textit{// let the other go first} \\
\quad \textit{while}(\textit{victim} == i)\ \{\} \quad \textit{// wait} \\
\}
\end{array}
$$

$$\textit{unlock}()\ \{\}$$

Ensures mutual execlusion

$$
\begin{array}{l}
void\ \ lock()\{\\
\quad i = tid();\\
\quad \color{blue}{victim} = i;\\
\quad while(\color{blue}{victim} == i)\\
\quad\quad \{\}\\
\}
\end{array}
$$

$unlock()\ \{\}$

$$CS_0^j \not\to CS_1^k \text{ and } CS_1^k \not\to CS_0^j$$

$void\ lock()\{$
$\quad i = tid();$
$\quad victim = i;$
$\quad while(victim == i)$
$\qquad \{\}$
$\}$

$unlock()\ \{\}$

$CS_0^j \not\to CS_1^k$ and $CS_1^k \not\to CS_0^j$

$void\ lock()\{$
  $i = tid();$
  $victim = i;$
  $while(victim == i)$
    $\{\}$
$\}$

$unlock()\ \{\}$

$$\begin{array}{c} \mathrm{W}_0(victim, 0) \\ \downarrow \\ \mathrm{R}_0(victim, 1) \\ \downarrow \\ CS_0 \end{array}$$

Mutual Exclusion

```
void lock(){
    i = tid();
    victim = i;
    while(victim == i)
        {}
}


unlock() {}
```

$$CS_0^j \not\to CS_1^k \text{ and } CS_1^k \not\to CS_0^j$$

$$\begin{array}{ccc} \mathrm{W}_0(victim, 0) & & \mathrm{W}_1(victim, 1) \\ \downarrow & & \downarrow \\ \mathrm{R}_0(victim, 1) & & \mathrm{R}_1(victim, 0) \\ \downarrow & & \downarrow \\ CS_0 & & CS_1 \end{array}$$

$CS_0^j \not\rightarrow CS_1^k$ and $CS_1^k \not\rightarrow CS_0^j$

```
void lock(){
  i = tid();
  victim = i;
  while(victim == i)
    {}
}

unlock() {}
```

$$
\begin{tikzcd}
\mathrm{W}_0(victim, 0) \arrow[r] \arrow[d] & \mathrm{W}_1(victim, 1) \arrow[dl] \arrow[d] \\
\mathrm{R}_0(victim, 1) \arrow[d] & \mathrm{R}_1(victim, 0) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}
$$

Mutual Exclusion

```
void lock(){
    i = tid();
    victim = i;
    while(victim == i)
        {}
}


unlock() {}
```

$CS_0^j \not\to CS_1^k$ and $CS_1^k \not\to CS_0^j$

$$
\begin{tikzcd}
\mathrm{W}_0(victim, 0) \arrow[d] \arrow[dr] & \mathrm{W}_1(victim, 1) \arrow[l] \arrow[d] \\
\mathrm{R}_0(victim, 1) \arrow[d] & \mathrm{R}_1(victim, 0) \arrow[d] \\
CS_0 & CS_1
\end{tikzcd}
$$

Problem: What if the threads are not running concurrently?

```
void lock(){
  i = tid();
  j = 1 - i;
  flag[i] = true;   // I am interested
  victim = i;   // you go first
  while(flag[j] && victim == i) {}   // I am waiting
}


unlock() {
  i = tid();
  flag[i] = false;   // I am not interested
}
```

Peterson's Lock Algorithm

```
void lock(){
  i = tid();
  j = 1 − i;
  flag[i] = true;
  victim = i;
  while(flag[j] &&
        victim==i)
      {}
}

unlock() {
  i = tid();
  flag[i] = false;
}
```

w.l.o.g assume thread 0
writes to victim

Peterson's Lock Algorithm

```
void lock(){
  i = tid();
  j = 1 − i;
  flag[i] = true;
  victim = i;
  while(flag[j] &&
        victim==i)
     {}
}

unlock() {
  i = tid();
  flag[i] = false;
}
```

$$
\begin{tikzcd}
W_0(\mathit{flag}[0], \mathit{true}) \arrow[d] & \text{a:}W_1(\mathit{flag}[1], \mathit{true}) \arrow[d] \\
W_0(\mathit{victim}, 0) & W_1(\mathit{victim}, 1) \arrow[l]
\end{tikzcd}
$$

w.l.o.g assume thread 0 writes to victim

```
void lock(){
  i = tid();
  j = 1 - i;
  flag[i] = true;
  victim = i;
  while(flag[j] &&
        victim==i)
     {}
}

unlock() {
  i = tid();
  flag[i] = false;
}
```

$$
\begin{tikzcd}
W_0(\mathit{flag}[0], \mathit{true}) \arrow[d] & a{:}W_1(\mathit{flag}[1], \mathit{true}) \arrow[d] \\
W_0(\mathit{victim}, 0) \arrow[d] & W_1(\mathit{victim}, 1) \arrow[l] \\
b{:}R_0(\mathit{flag}[1], \mathit{false}) \arrow[d] & \\
R_0(\mathit{victim}, 0) \arrow[d] & \\
CS_0 &
\end{tikzcd}
$$

w.l.o.g assume thread 0 writes to victim

Peterson's Lock Algorithm

```
void lock(){
  i = tid();
  j = 1 - i;
  flag[i] = true;
  victim = i;
  while(flag[j] &&
        victim == i)
     {}
}

unlock() {
  i = tid();
  flag[i] = false;
}
```

$$\begin{tikzcd}
W_0(\mathit{flag}[0], \mathit{true}) \arrow[d] & a{:}W_1(\mathit{flag}[1], \mathit{true}) \arrow[d] \\
W_0(\mathit{victim}, 0) \arrow[d] & W_1(\mathit{victim}, 1) \arrow[l] \\
b{:}R_0(\mathit{flag}[1], \mathit{false}) \arrow[d] & \\
R_0(\mathit{victim}, 0) \arrow[d] & \\
\mathit{CS}_0 &
\end{tikzcd}$$

no intermediate write on *flag*[1] between *a* and *b*

w.l.o.g assume thread 0 writes to victim

Properties of Mutual Exclusion

(Essential) **Deadlock freedom.** If one or multiple process trying to enter critical section, then some process eventually will enter critical section

Starvation freedom. Any process/thread trying to enter critical section, will eventually enter critical section

- Requires fairness

(Essential) **Deadlock freedom.** If one or multiple process trying to enter critical section, then some process eventually will enter critical section

Starvation freedom. Any process/thread trying to enter critical section, will eventually enter critical section

- Requires fairness

Waiting.
If one process/thread delays in critical section then other processes/threads also get delayed

- What if a thread acquires a lock and crashes?
- Requires fault tolerance

```
void lock(){
   i = tid();
   j = 1 − i;
   flag[i] = true;
   while(flag[j]) {}
}


unlock() {
   i = tid();
   flag[i] = false;
}
```

$$
\begin{tikzcd}
\text{a: } W_0(\mathit{flag}[0], \mathit{true}) \arrow[d] \arrow[dr] & \text{c: } W_1(\mathit{flag}[1], \mathit{true}) \arrow[d] \arrow[dl] \\
\text{b: } R_0(\mathit{flag}[1], \mathit{true}) \arrow[d] & \text{d: } R_1(\mathit{flag}[0], \mathit{true}) \arrow[d] \\
\vdots & \vdots
\end{tikzcd}
$$

Deadlock: writes happen before reads

Alternative Lock Unlock

$$\begin{array}{l} void\ lock()\{ \\ \quad i = tid(); \\ \quad victim = i; \quad //\ let\ the\ other\ go\ first \\ \quad while(victim == i)\ \{\}\ \ //\ wait \\ \} \end{array}$$

$$unlock()\ \{\}$$

Is it deadlock-free?

It deadlocks if one thread runs completely before the other

Peterson's Lock Algorithm

$void\ lock()\{$
$\quad i = tid();$
$\quad j = 1 - i;$
$\quad flag[i] = true;$ // I am interested
$\quad victim = i;$ // you go first
$\quad while(flag[j]\ \&\&\ victim == i)\ \{\}$ // I am waiting
$\}$

$unlock()\ \{$
$\quad i = tid();$
$\quad flag[i] = false;$ // I am not interested
$\}$

It is deadlock free and starvation free.

Nested Locking

$$
\begin{array}{l||l}
lock(m_1); & lock(m_2); \\
lock(m_2); & lock(m_1); \\
\vdots & \vdots \\
unlock(m_2); & unlock(m_1); \\
unlock(m_1); & unlock(m_2);
\end{array}
$$

May lead to deadlock

Nested Locking

$$
\begin{array}{l||l}
lock(m_1); & lock(m_2); \\
lock(m_2); & lock(m_1); \\
\vdots & \vdots \\
unlock(m_2); & unlock(m_1); \\
unlock(m_1); & unlock(m_2);
\end{array}
$$

May lead to deadlock

Solution: whenever threads lock multiple mutexes, they do so in the same order

References

The Art of Multiprocessor Programming (chapter 2)
2nd Edition - September 8, 2020
Authors: Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear