Synchronization

Disclaimer: some slides are adopted from the book authors' slides with permission

Recap: Thread

- What is it?
  - Independent flow of control
- What does it need (thread private)?
  - Stack
- What for?
  - Lightweight programming construct for concurrent activities
- How to implement?
  - Kernel thread vs. user thread

Recap: Process vs. Thread

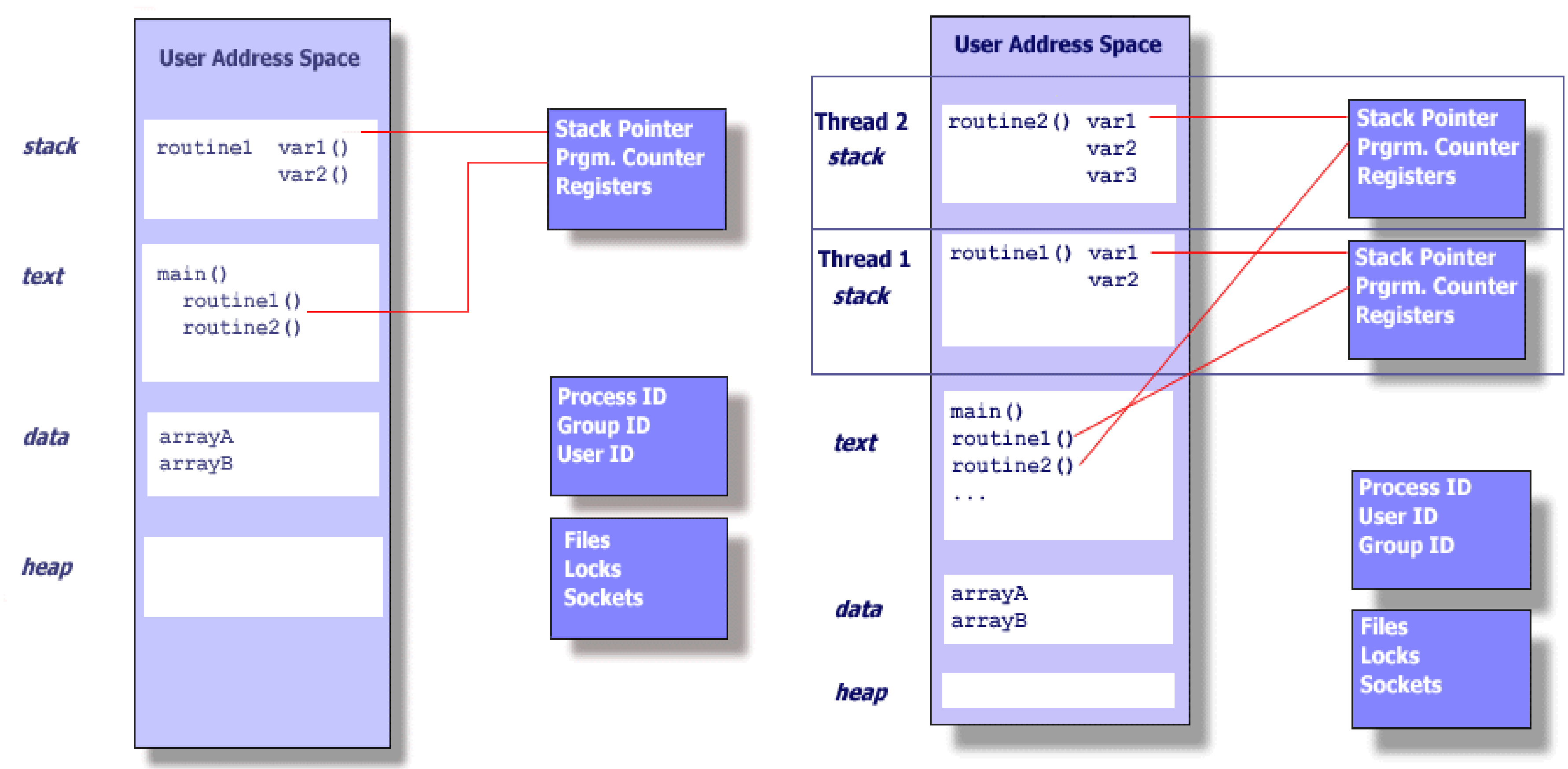


Figure source: https://computing.llnl.gov/tutorials/pthreads/

Recap: Multi-threads vs. Multi-processes

- Multi-processes
  - (+) protection
  - (-) performance (?)

Process-per-tab

- Multi-threads
  - (+) performance
  - (-) protection

Single-process multi-threads

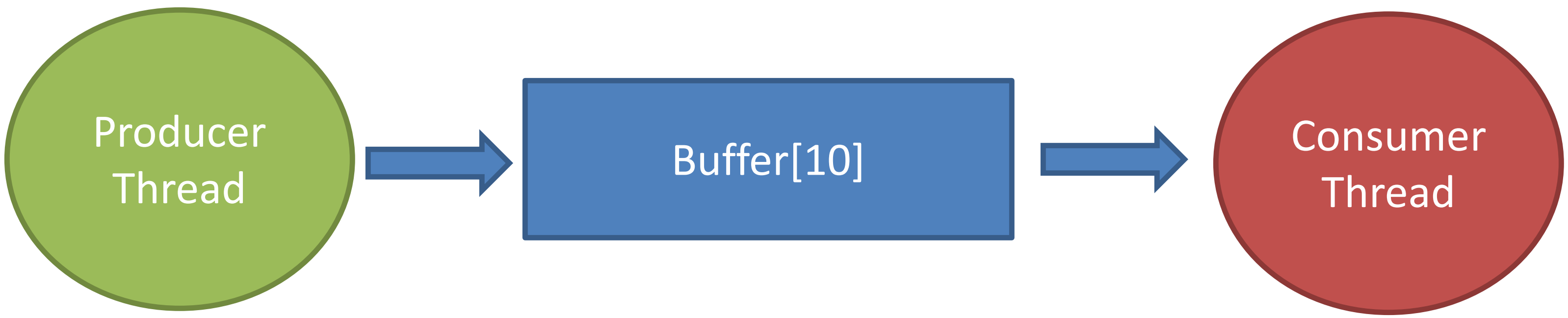
Agenda

- **Mutual exclusion**
  - **Peterson's algorithm (Software)**
  - Synchronization instructions (Hardware)
    - Spinlock 自旋锁
- High-level synchronization mechanisms
  - Mutex
  - Semaphore
  - Monitor

Producer/Consumer

线程不安全

计算机在执行程序时，每条指令都是在CPU中执行的，而执行指令过程中，势必涉及到数据的读取和写入



Producer/Consumer

Producer

```
while (true){

   /* wait if buffer full */
   while (counter == 10);

   /* produce data */
   buffer[in] = sdata;
   in = (in + 1) % 10;

   /* update number of
      items in buffer */
   counter++;
}
```

Problem:当同步工作的时候，有时候counter的值还没有完全被确定就被另一边更改了（Race condition）

Consumer

```
while (true){

   /* wait if buffer empty */
   while (counter == 0);

   /* consume data */
   sdata = buffer[out];
   out = (out + 1) % 10;

   /* update number of
      items in buffer */
   counter--;
}
```

Producer/Consumer

Producer

```
while (true){

   /* wait if buffer full */
   while (counter == 10);

   /* produce data */
   buffer[in] = sdata;
   in = (in + 1) % 10;

   /* update number of
      items in buffer */
   R1 = load (counter);
   R1 = R1 + 1;
   counter = store (R1);
}
```

Consumer

```
while (true){

   /* wait if buffer empty */
   while (counter == 0);

   /* consume data */
   sdata = buffer[out];
   out = (out + 1) % 10;

   /* update number of
      items in buffer */
   R2 = load (counter);
   R2 = R2 – 1;
   counter = store (R2);
}
```

Check Yourself

```
int count = 0;
int main()
{
        count = count + 1;
        return count;
}

$ gcc –O2 –S sync.c
```

```
…
movl    count(%rip), %eax
addl    $1, %eax
movl    %eax, count(%rip)
…
```

Race Condition

Initial condition: *counter = 5*

Thread 1

```
R1 = load (counter);
R1 = R1 + 1;
counter = store (R1);
```

Thread 2

```
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
```

- What are the possible outcome?

Race Condition

Initial condition: *counter = 5*

R1 = load (counter);
R1 = R1 + 1;
counter = store (R1);
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
	R1 = load (counter);
	R1 = R1 + 1;
	R2 = load (counter);
	R2 = R2 – 1;
	counter = store (R1);
	counter = store (R2);
		R1 = load (counter);
		R1 = R1 + 1;
		R2 = load (counter);
		R2 = R2 – 1;
		counter = store (R2);
		counter = store (R1);

counter = 5	*counter = 4*	*counter = 6*

- Why this happens?

Race Condition

- A situation **when two or more threads read and write shared data at the same time**
- Correctness depends on the execution order

Thread 1

```
R1 = load (counter);
R1 = R1 + 1;
counter = store (R1);
```

read

write

Thread 2

```
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
```

- How to prevent race conditions?

Critical Section

- Code sections of potential race conditions

Thread 1	Thread 2
Do something	Do something
..	..
R1 = load (**counter**);	R2 = load (**counter**);
R1 = R1 + 1;	R2 = R2 – 1;
counter = store (R1);	**counter** = store (R2);
...	..
Do something	Do something

Critical sections

Solution Requirements

- Mutual Exclusion
  - If a thread executes its critical section, *no other threads* can enter their critical sections
- Progress
  - If no one executes a critical section, someone can enter its critical section
- Bounded waiting
  - Waiting (time/number) must be bounded

Simple Solution (?): Use a Flag

```
// wait
while (in_cs)
    ;
// enter critical section
in_cs = true;

Do something

// exit critical section
in_cs = false;
```

T1	T2
while(in_cs){};	
	while(in_cs){};
	in_cs = true;
in_cs = true;	
//enter	//enter

- Mutual exclusion is not guaranteed

Peterson's Solution

- Software solution (no h/w support)
- Two process solution
  - Multi-process extension exists
- The two processes share two variables:
  - int turn;
    - The variable turn indicates whose turn it is to enter the critical section
  - Boolean flag[2]
    - The flag array is used to indicate if a process is ready to enter the critical section.

KU THE UNIVERSITY OF KANSAS

Peterson's Solution

Thread 1

```
do {
   flag[0] = TRUE;
   turn = 1;
   while (flag[1]  &&  turn==1)
   {};
   // critical section

   flag[0] = FALSE;

   // remainder section
} while (TRUE)
```

Thread 2

```
do {
   flag[1] = TRUE;
   turn = 0;
   while (flag[0]  &&  turn==0)
   {};
   // critical section

   flag[1] = FALSE;

   // remainder section
} while (TRUE)
```

- Solution meets all three requirements
  - Mutual exclusion: P0 and P1 cannot be in the critical section at the same time
  - Progress: if P0 does not want to enter critical region, P1 does no waiting
  - Bounded waiting: process waits for at most one turn

Peterson’s Solution

- Limitations
  - Only supports two processes
    - generalizing for more than two processes has been achieved, but not very efficient
  - Assumes LOAD and STORE instructions are atomic
    - In reality, no guarantees
  - Assumes memory accesses are not reordered
    - compiler re-orders instructions (gcc –O2, -O3, …)
    - Out-of-order processor re-orders instructions

并不重新排序

Atomic: Reading and writing this data type is guaranteed to happen in a single instruction, so there’s no way for a handler to run “in the middle” of an access.
程序的原子性指：整个程序中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节

Reordering by the CPU

Initially X = Y = 0

注意读x和写回x是原子操作，两个线程
不能同时执行，无法在同一行

Thread 0	Thread 1
X = 1	Y = 1
R1 = Y	R2 = X

Thread 0	Thread 1
R1 = Y	
	R2 = X
X = 1	
	Y = 1

- Possible values of R1 and R2?
  - 0,1
  - 1,0
  - 1,1
  - **0,0 ← possible on PC** ？

Summary

- Peterson's algorithm
  - Software-only solution
    - Turn based
  - Pros
    - No hardware support
    - Satisfy all requirements
      - Mutual exclusion, progress, bounded waiting
  - Cons
    - Complicated
    - Assume program order
    - May not work on out-of-order processors

Recap

- Race condition
  - A situation when two or more threads **read and write** shared data at the same time
  - Correctness depends on the execution order
- Critical section
  - Code sections of potential race conditions
- Mutual exclusion
  - If a thread executes its critical section, *no other threads* can enter their critical sections

Recap

- Peterson's algorithm
  - Software-only solution
    - Turn based
  - Pros
    - No hardware support
    - Satisfy all requirements
      - Mutual exclusion, progress, bounded waiting
  - Cons
    - Complicated
    - Assume program order
    - May not work on out-of-order processors

Today

- Hardware support
  - Synchronization instructions
- Lock
  - Spinlock
  - Mutex

Lock

spin lock是一种死等的锁机制。当发生访问资源冲突的时候，可以有两个选择：一个是死等，一个是挂起当前进程，调度其他进程执行。spin lock是一种死等的机制，当前的执行thread会不断的重新尝试直到获取锁进入临界区。
保护共享资源

- General solution
  - Protect critical section via a lock
  - Acquire on enter, release on exit

```
do {
    acquire lock;

        critical section

    release lock;

        remainder section

} while(TRUE);
```

How to Implement a Lock?

- Unicore processor
  - No true concurrency

    one thread at a time
  - Threads are *interrupted* by the OS
    - scheduling events: timer interrupt, device interrupts
- Disabling interrupt
  - Threads can't be interrupted

```
do {

    disable interrupts;
        critical section
    enable interrupts;

        remainder section
} while(TRUE);
```

How to Implement a Lock?

- Multicore processor
  - True concurrency
    - More than one active threads sharing memory
  - Disabling interrupts don't solve the problem
    - More than one threads are executing at a time
- Hardware support
  - Synchronization instructions
    - **Atomic** *test&set* instruction 原子性操作：不可被中断的一个或一系列操作
    - **Atomic** *compare&swap* instruction
- What do we mean by **atomic**?
  - All or nothing

THE UNIVERSITY OF KANSAS

TestAndSet Instruction

- Pseudo code

之所以叫硬同步，是因为TestAndSet()函数中的三条语句是由硬件保证同步的，即硬件保证这三条语句必须原子运行，中间不发生任何中断，如同一条语句

```
boolean TestAndSet (boolean *target)
{
        boolean rv = *target;
         *target = TRUE;
         return rv:
}
```

https://zhidao.baidu.com/question/2117056499260142027.html

如果criticalsection没有被锁住，即lock值为false，那么传入TestAndSet()的参数为false，【【在函数体内：rv先被赋为false，然后*target被赋为true，然后返回rv（值为false）。运行的结果为：返回值为false，使线程跳出while循环，得以进入criticalsection；同时参数（即lock）被改为true，表示criticalsection被锁住】】，其他线程在当前线程没有出criticalsection之前（即运行lock=false这句之前），不能进入criticalsection。为什么呢，下面说

Spinlock using *TestAndSet*

spinlock 在 Linux底下是以 spinlock_t 来表示的。使用spinlock
必须包含#include <linux/spinlock>

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
    *mutex = 0;
}

void lock (int *mutex)
{
    while(TestAndSet(mutex))
        ;
}

void unlock (int *mutex)
{
    *mutex = 0;
}
```

while的死循环代表
另一个线程被锁住了

CAS (Compare & Swap) Instruction

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。

如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。

否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。
https://www.cnblogs.com/shangdawei/p/3917117.html

- Pseudo code

```
int CAS(int *value, int oldval, int newval)
{
        int temp = *value;
        if (*value == oldval)
           *value = newval;
        return temp;
}
```

KU THE UNIVERSITY OF KANSAS

Spinlock using *CAS*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    while(CAS(&mutex, 0, 1) != 0);
}

void unlock (int *mutex) {
    *mutex = 0;
}
```

What's Wrong With **Spin**locks?

- Very wasteful
  - Waiting thread continues to use CPU cycles
  - While doing absolutely nothing but wait
  - 100% CPU utilization, but no useful work done
  - Power consumption, fan noise, …
- Useful when
  - You hold the lock only *briefly*
- Otherwise
  - A better solution is needed