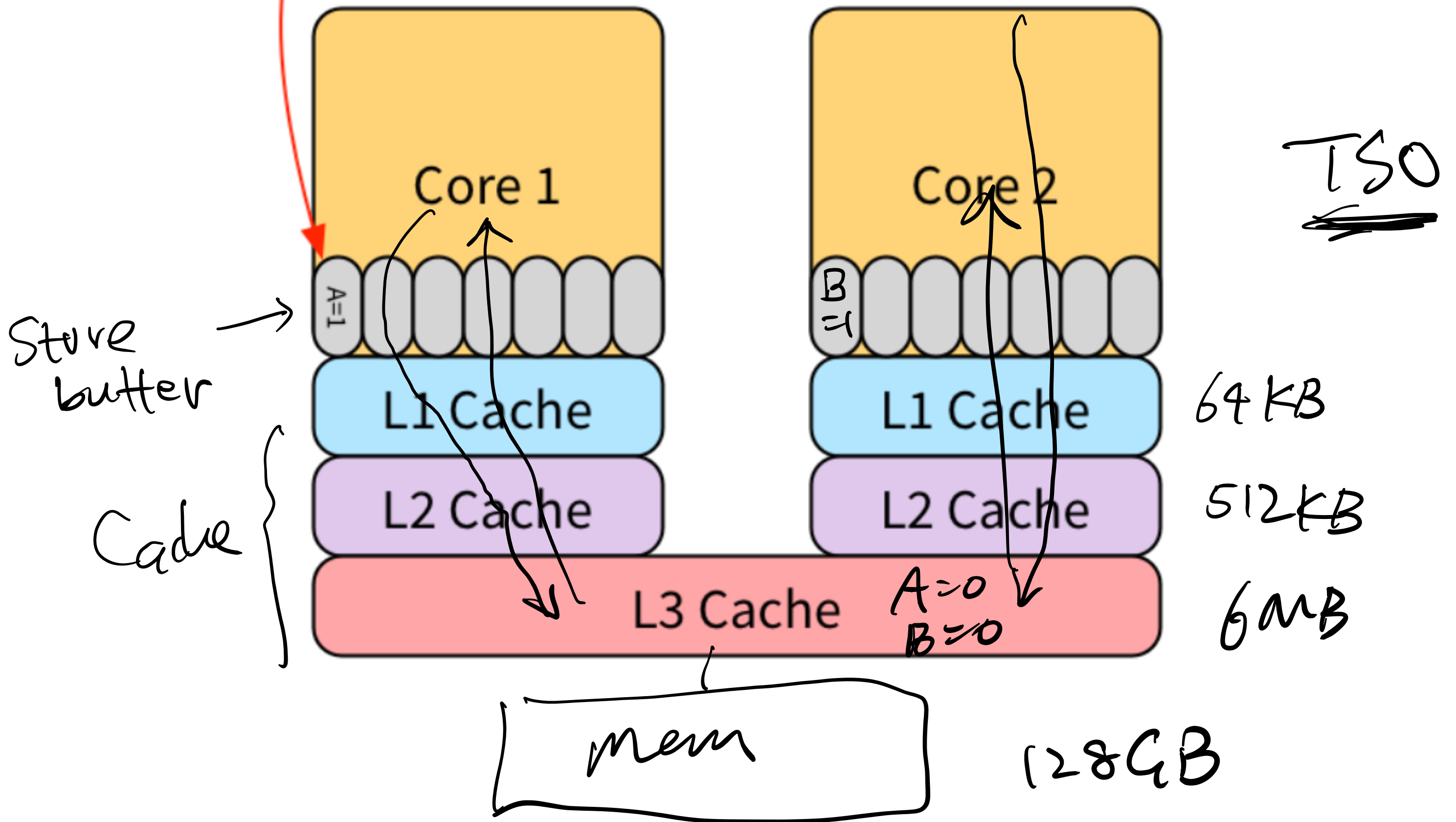
```
begin integer j;
L1: choosing [i] := 1;
    number[i] := 1 + maximum (number[1], . . . , number[N]);
    choosing[i] := 0;
    for j = 1 step 1 until N do
        begin
        L2: if choosing[j] ≠ 0 then goto L2;
        L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                i) then goto L3;
        end;
    critical section;
    number[i] := 0;
    noncritical section;
    goto L1;
end
```

Borrowed from Lamport, Leslie. "A new solution of Dijkstra's concurrent programming problem." ~~*Concurrency: the Works of Leslie Lamport*. 2019. 171-1~~78.

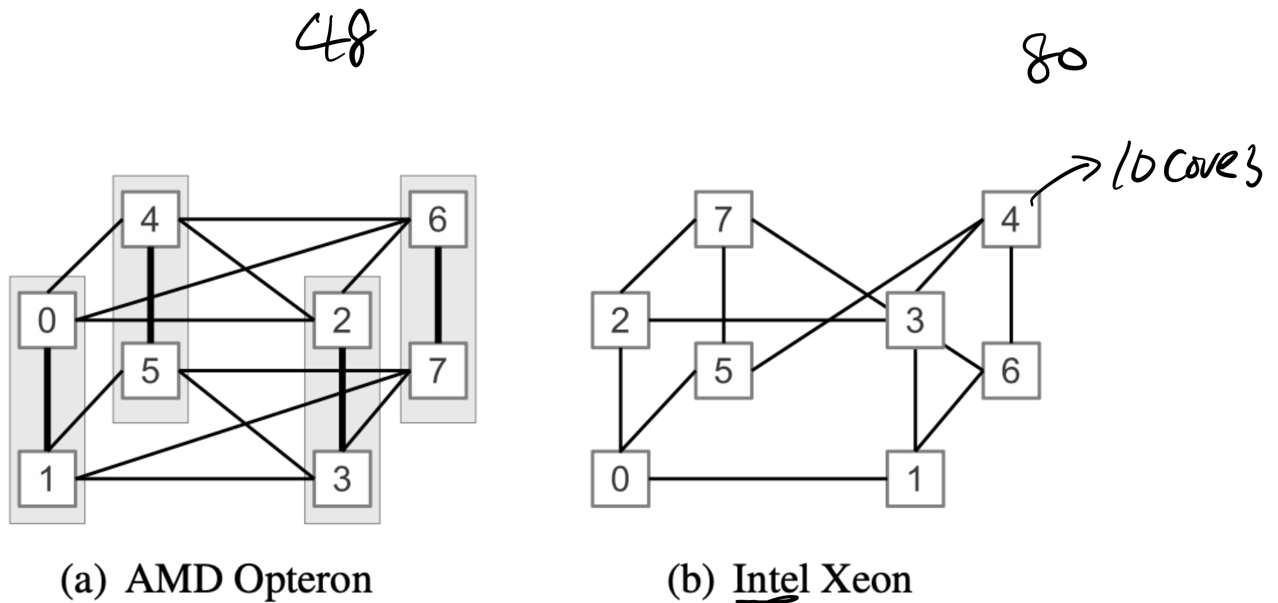


Borrowed from blog "Memory Consistency Model: A Tutorial", James Bornholt.
https://www.cs.utexas.edu/~bornholt/post/memory-models.html

multi-core CPU

Name	Opteron	Xeon	Niagara	Tilera
System	AMD Magny Cours	Intel Westmere-EX	SUN SPARC-T5120	Tilera TILE-Gx36
Processors	4× AMD Opteron 6172	8× Intel Xeon E7-8867L	SUN UltraSPARC-T2	TILE-Gx CPU
# Cores	48	80 (no hyper-threading)	8 (64 hardware threads)	36
Core clock	2.1 GHz	2.13 GHz	1.2 GHz	1.2 GHz
L1 Cache	64/64 KiB I/D	32/32 KiB I/D	16/8 KiB I/D	32/32 KiB I/D
L2 Cache	512 KiB	256 KiB		256 KiB
Last-level Cache	2×6 MiB (shared per die)	30 MiB (shared)	4 MiB (shared)	9 MiB Distributed
Interconnect	6.4 GT/s HyperTransport (HT) 3.0	6.4 GT/s QuickPath Interconnect (QPI)	Niagara2 Crossbar	Tilera iMesh
Memory #Channels / #Nodes	128 GiB DDR3-1333 4 per socket / 8	192 GiB Sync DDR3-1067 4 per socket / 8	32 GiB FB-DIMM-400 8 / 1	16 GiB DDR3-800 4 / 2
OS	Ubuntu 12.04.2 / 3.4.2	Red Hat EL 6.3 / 2.6.32	Solaris 10 u7	Tilera EL 6.3 / 2.6.40

Table 1: The hardware and the OS characteristics of the target platforms.



(a) AMD Opteron (b) Intel Xeon

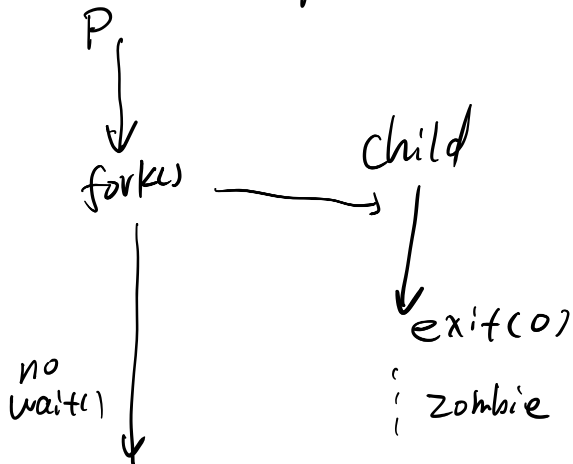
Figure 2: The system topologies.

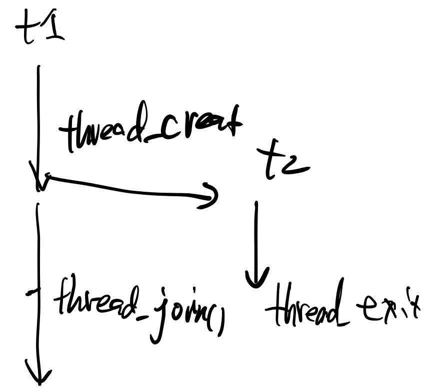
Borrowed from "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask", SOSP'13
https://sigops.org/s/conferences/sosp/2013/papers/p33-david.pdf

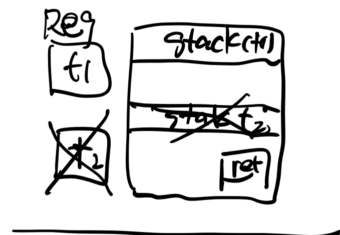
1. Last time
2. Critical section
3. Bakery algorithm
4. Mutexes
5. Condition variables
6. Semaphores

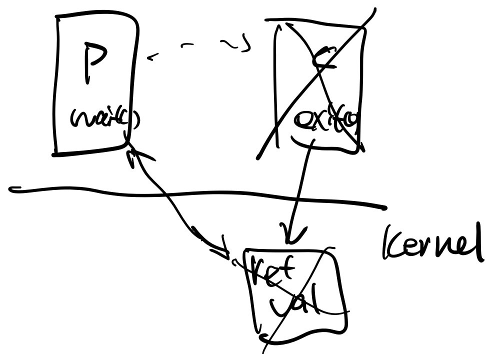
---

zombie process ⟶ zombie thread?









Process

process vs. threads

```
void *f(void *xx) {        t2
    sleep(1);
    printf("this is f\n");
    exit(0);
}

int main(){                t1
    pthread_t tid;
    pthread_create(&tid, NULL, f, NULL);
    pthread_join(tid, NULL); // Line X
    printf("this is main\n");
}   exit(0);
```

Answer: "this is main"

Q? output? "this is f"

Q? output? ["this is main" / "this is f"] "depends" 2 poss / 3 possible

SC:

→ W_1, W_2, R_1, R_2

```
T1: W1(x)=1, R1(y)=2
T2: W2(y)=2, R2(x)=1
```

$R_1 R_2 W_1 W_2$

x,y=0

```
T1: W1(x)=1, R1(y)=0
T2: W2(y)=2, R2(x)=0
```

TSO

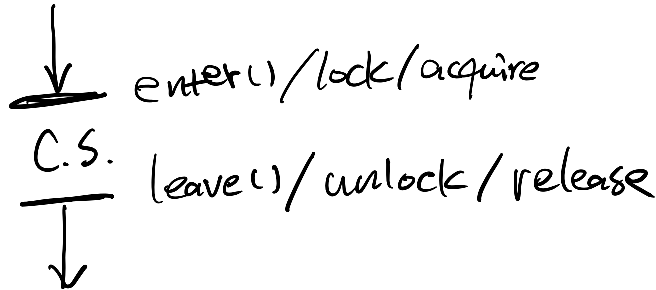
Yes.

$W_1 R_1 W_2 R_2$

Critical section

1. mutual exclusion
2. progress
3. bounded waiting

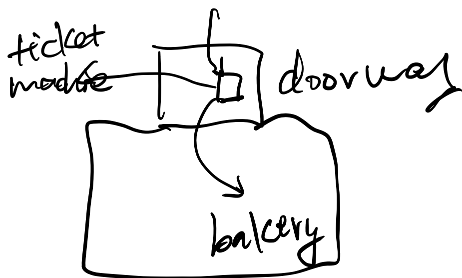
(① and ② ← scheduling constraints)

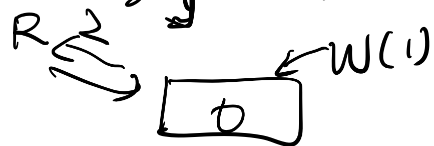


implement C.S.

→ atomic memory op

bakery algo. ← safe register





1. no concurrent W, return normal val
2. ∃ concurrent W, return **anything**

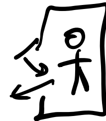
Mutex.

Usage:

```
mutex_t m
mutex_init(mutex_t* m)
acquire(mutex_t* m)   <- enter()
release(mutex_t* m)   <- leave()
```

pthread

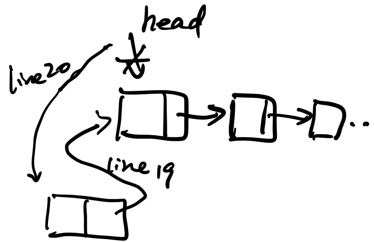
```
1   1. Example to illustrate interleavings: say that thread A executes f()
2   and thread B executes g(). (Here, we are using the term "thread"
3   abstractly. This example applies to any of the approaches that fall
4   under the word "thread".)
5
6       a. [this is pseudocode]
7
8           int x;
9
10          int main(int argc, char** argv) {
11
12              tid tid1 = thread_create(f, NULL);
13              tid tid2 = thread_create(g, NULL);
14
15              thread_join(tid1);
16              thread_join(tid2);
17
18              printf("%d\n", x);
19
20          }
21
22          void f() {
23              x = 1;
24              thread_exit();
25          }
26
27          void g() {
28              x = 2;
29              thread_exit();
30          }
31
32          What are possible values of x after A has executed f() and B has
33          executed g()? In other words, what are possible outputs of the
34          program above?
35
36
37      b. Same question as above, but f() and g() are now defined as
38      follows
39
40          int y = 12;
41
42          f() { x = y + 1; }
43          g() { y = y * 2; }
44
45          What are the possible values of x?
46
47      c. Same question as above, but f() and g() are now defined as
48      follows:
49
50          int x = 0;
51
52          f() { x = x + 1; }
53          g() { x = x + 2; }
54
55          What are the possible values of x?
56
57
```

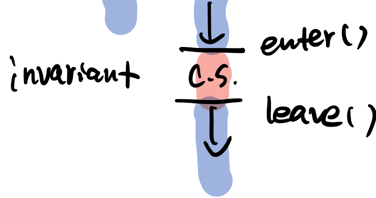


enter()
leave()
3.
1.2.3


```
58
59
60  2. Linked list example
61
62      struct List_elem {
63          int data;
64          struct List_elem* next;
65      };
66
67      List_elem* head = 0;
68
69      insert(int data) {
70          List_elem* l = new List_elem;
71          l->data = data;
72          l->next = head;
73          head = l;
74      }
75
76      What happens if two threads execute insert() at once and we get the
77      following interleaving?
78
79      thread 1: l->next = head
80      thread 2: l->next = head
81      thread 2: head = l;
82      thread 1: head = l;
83
84
85
```

```
1   CS5600 Week5.b
2
3   The handout from the last class gave examples of race conditions.
4   The following panels demonstrate the use of concurrency primitives
5   (mutexes, etc.). We are using concurrency primitives to eliminate
6   race conditions (see items 1 and 2a) and improve scheduling (see item 2b).
7
8
9   1. Protecting the linked list......
10
11      Mutex list_mutex;
12
13      insert(int data) {
14        List_elem* l = new List_elem;
15        l->data = data;
16
17        acquire(&list_mutex);
18
19        l->next = head;
20        head = l;
21
22        release(&list_mutex);
23      }
24
```






```
25  2. Producer/consumer revisited [also known as bounded buffer]
26
27  2a. Producer/consumer [bounded buffer] with mutexes
28
29      Mutex mutex;
30
31      void producer (void *ignored) {
32        for (;;) {
33          /* next line produces an item and puts it in nextProduced */
34          nextProduced = means_of_production();
35
36          acquire(&mutex);
37          while (count == BUFFER_SIZE) {
38            release(&mutex);
39            yield(); /* or schedule() */
40            acquire(&mutex);
41          }
42
43          buffer [in] = nextProduced;
44          in = (in + 1) % BUFFER_SIZE;
45          count++;
46          release(&mutex);
47        }
48      }
49
50      void consumer (void *ignored) {
51        for (;;) {
52
53          acquire(&mutex);
54          while (count == 0) {
55            release(&mutex);
56            yield(); /* or schedule() */
57            acquire(&mutex);
58          }
59
60          nextConsumed = buffer[out];
61          out = (out + 1) % BUFFER_SIZE;
62          count--;
63          release(&mutex);
64
65          /* next line abstractly consumes the item */
66          consume_item(nextConsumed);
67        }
68      }
69
```

```
 70
 71   2b. Producer/consumer [bounded buffer] with mutexes and condition variables
 72
 73       Mutex mutex;
 74       Cond nonempty;
 75       Cond nonfull;
 76
 77       void producer (void *ignored) {
 78         for (;;) {
 79           /* next line produces an item and puts it in nextProduced */
 80           nextProduced = means_of_production();
 81
 82           acquire(&mutex);
 83           while (count == BUFFER_SIZE)
 84             cond_wait(&nonfull, &mutex);
 85
 86           buffer [in] = nextProduced;
 87           in = (in + 1) % BUFFER_SIZE;
 88           count++;
 89           cond_signal(&nonempty, &mutex);
 90           release(&mutex);
 91         }
 92       }
 93
 94       void consumer (void *ignored) {
 95         for (;;) {
 96
 97           acquire(&mutex);
 98           while (count == 0)
 99             cond_wait(&nonempty, &mutex);
100
101           nextConsumed = buffer[out];
102           out = (out + 1) % BUFFER_SIZE;
103           count--;
104           cond_signal(&nonfull, &mutex);
105           release(&mutex);
106
107           /* next line abstractly consumes the item */
108           consume_item(nextConsumed);
109         }
110       }
111
112
113   Question: why does cond_wait need to both release the mutex and
114   sleep? Why not:
115
116       while (count == BUFFER_SIZE) {
117         release(&mutex);
118         cond_wait(&nonfull);
119         acquire(&mutex);
120       }
121
```