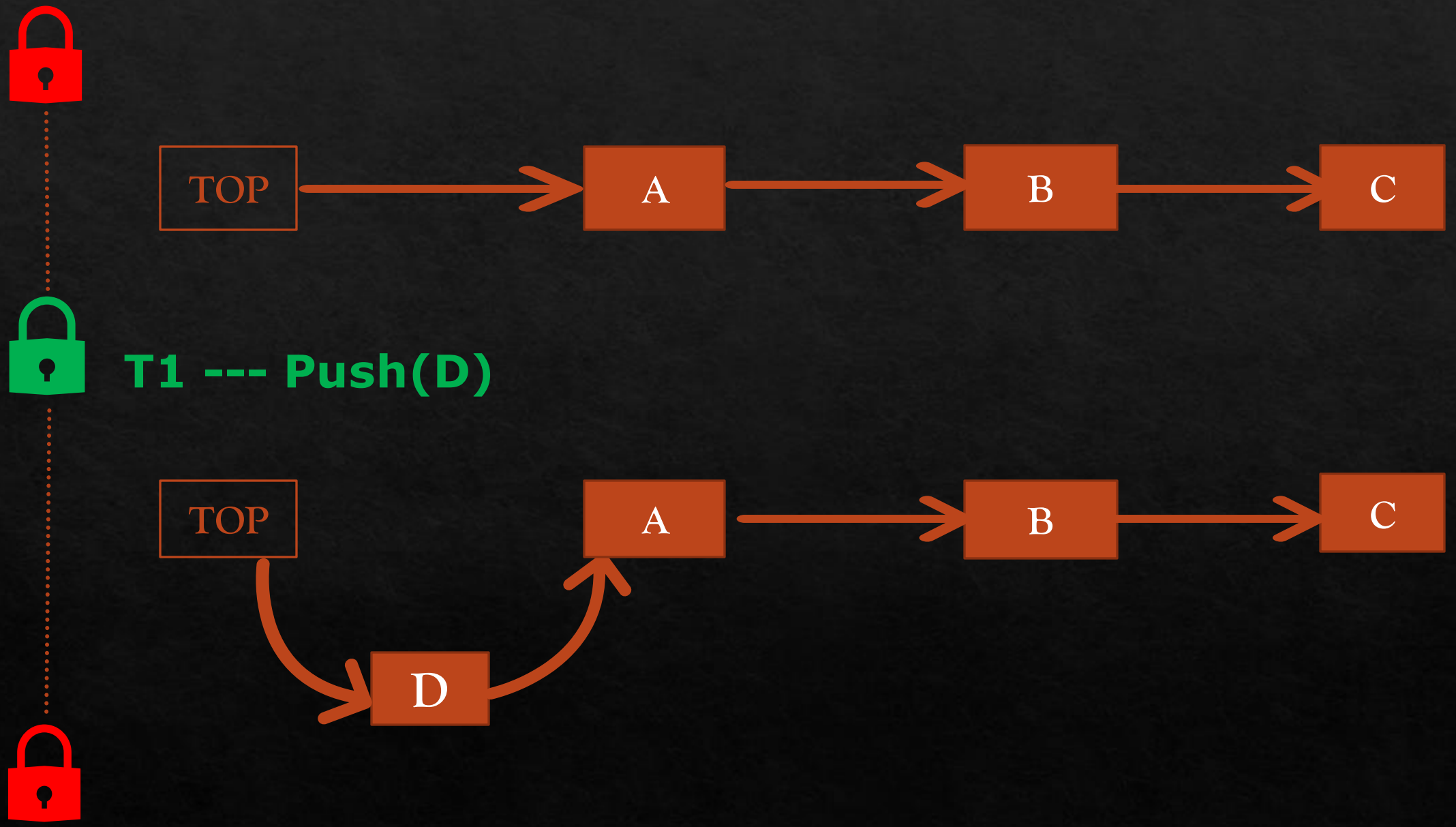
Elimination stack

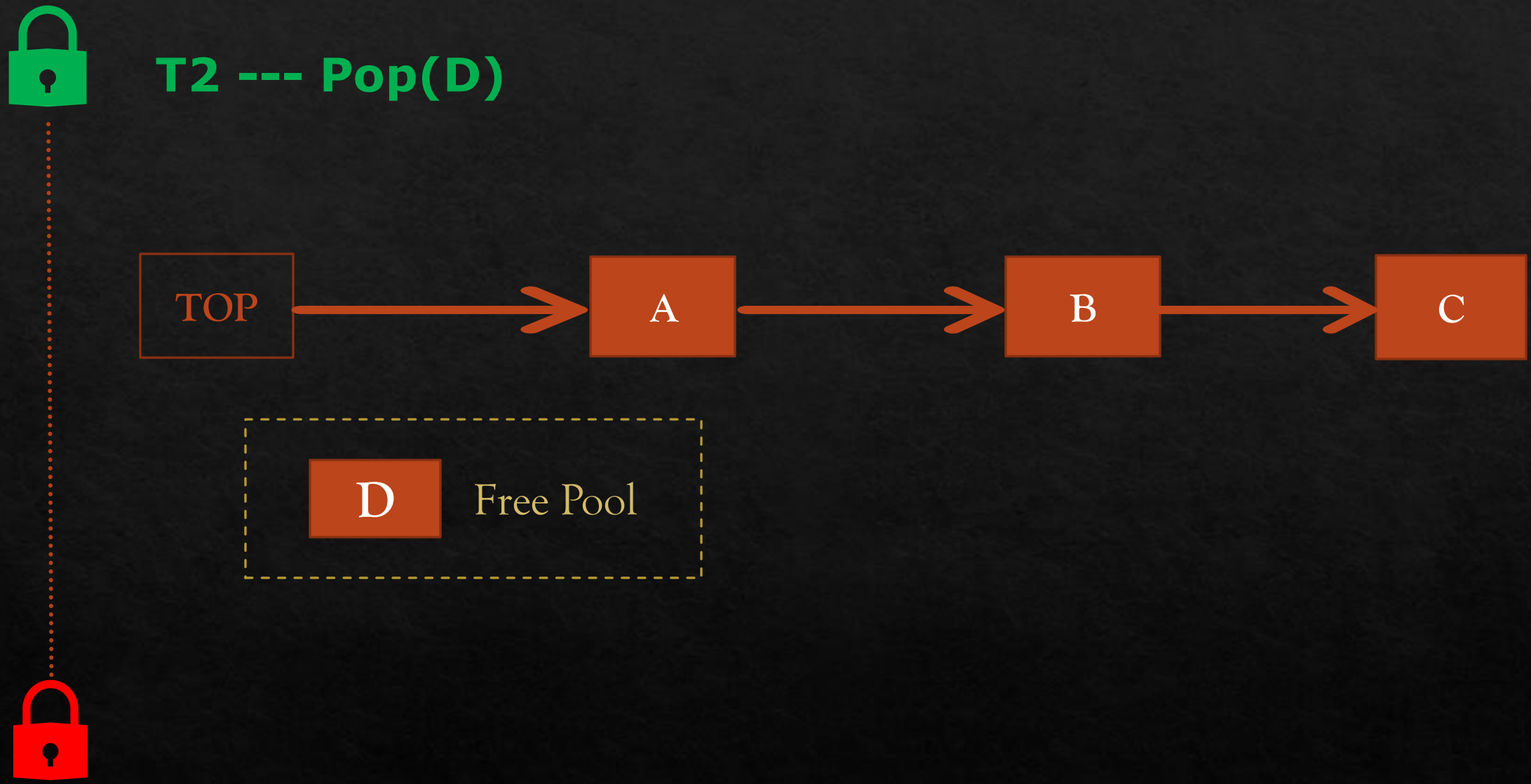
Aditya

203050079

Introduction

- Shared stacks are widely used in parallel applications and operating systems.A concurrent shared stack is a data structure that supports the usual push and pop operations .
- The algorithm is linearizable and thus easy to modularly combine with other algorithms, it is lock-free and hence robust, it is parallel and hence scalable, and it utilizes its parallelization construct adaptively, which allows it to perform well at low loads.


TOP
A
B
C
T1 --- Push(D)
TOP
A
B
C
D

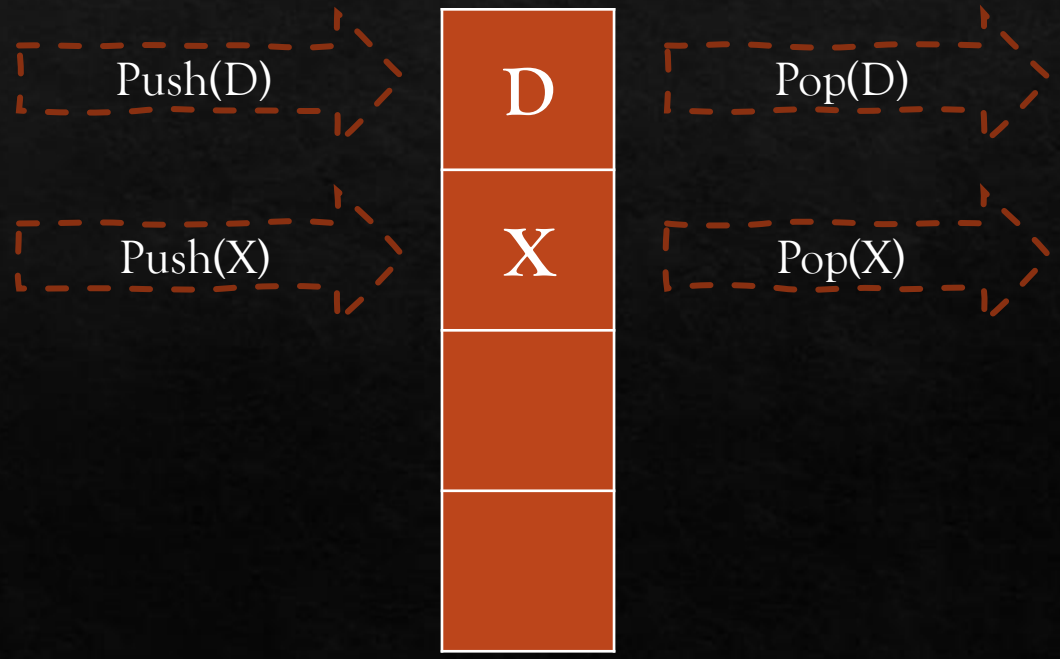

T2 --- Pop(D)
TOP
A
B
C
D
Free Pool

```
void StackOp(ThreadInfo* pInfo) {
P1: if(TryPerformStackOp(p)==FALSE)
P2:   LesOP(p);
P3: return;
}
void LesOP(ThreadInfo *p) {
S1: while (1) {
S2:   location[mypid]=p;
S3:   pos=GetPosition(p);
S4:   him=collision[pos];
S5:   while(!CAS(&collision[pos],him,mypid))
S6:     him=collision[pos];
S7:   if (him!=EMPTY) {
S8:     q=location[him];
S9:     if(q!=NULL&&q->id==him&&q->op!=p->op) {
S10:      if(CAS(&location[mypid],p,NULL)) {
S11:        if(TryCollision(p,q)==TRUE)
S12:          return;
S13:        else
S14:          goto stack;
          }
S15:      else {
S16:        FinishCollision(p);
S17:        return;
           }
        }
      }
S18:  delay(p->spin);
S19:  if (!CAS(&location[mypid],p,NULL)) {
S20:      FinishCollision(p);
S21:      return;
      }
stack:
S22:  if (TryPerformStackOp(p)==TRUE)
          return;
      }
}
```

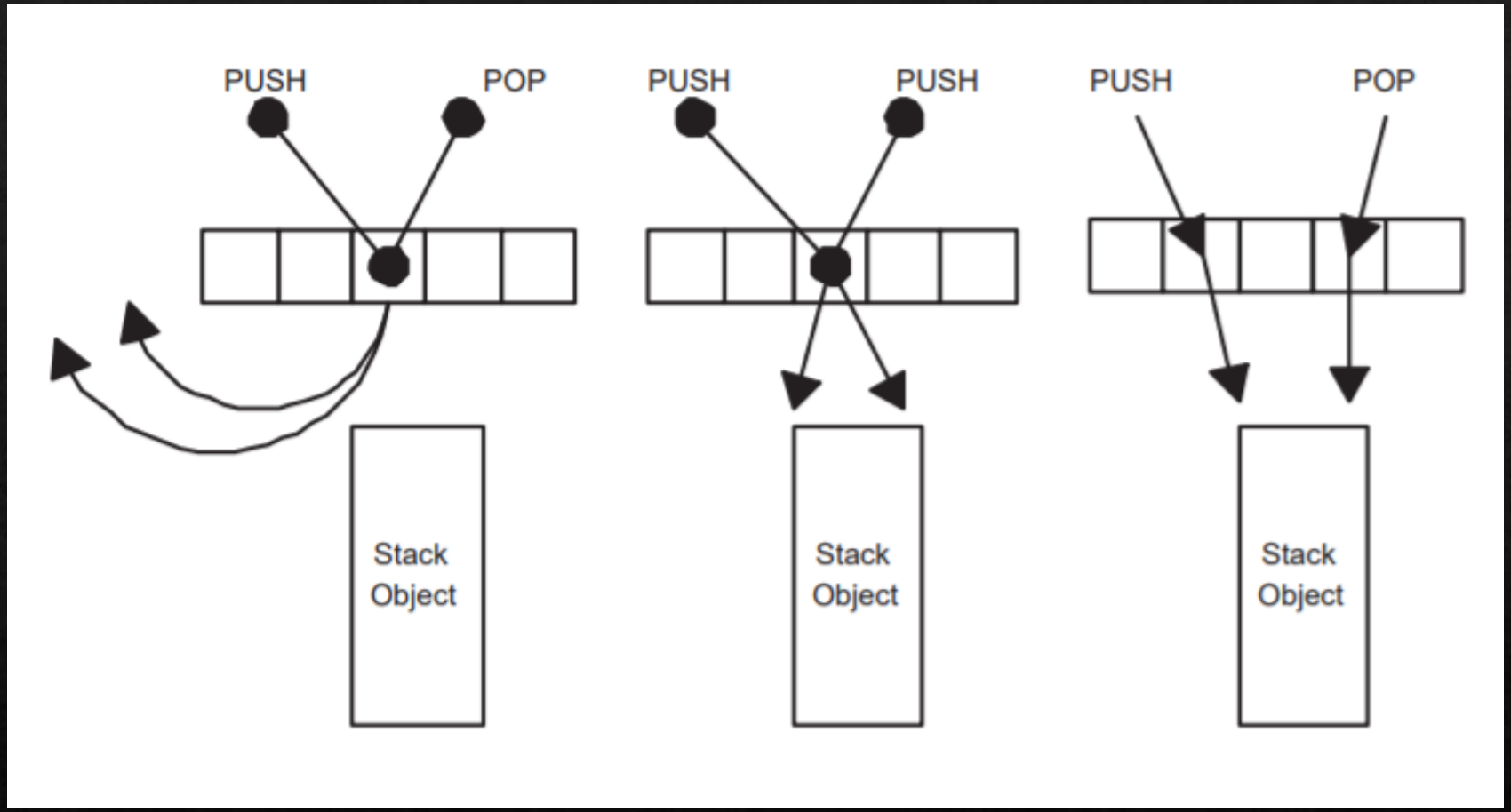
```
boolean TryPerformStackOp(ThreadInfo*p){
    Cell *phead,*pnext;
T1: if(p->op==PUSH) {
T2:    phead=S.ptop;
T3:    p->cell.pnext=phead;
T4:    if(CAS(&S.ptop,phead,&p->cell))
T5:      return TRUE;
T6:    else
T7:      return FALSE;
   }
T8: if(p->op==POP) {
T9:    phead=S.ptop;
T10:   if(phead==NULL) {
T11:     p->cell=EMPTY;
T12:     return TRUE;
       }
T13:   pnext=phead->pnext;
T14:   if(CAS(&S.ptop,phead,pnext)) {
T15:      p->cell=*phead;
T16:      return TRUE;
      }
T17:   else {
T18:      p->cell=EMPTY;
T19:     return FALSE;
       }
}
void FinishCollision(ProcessInfo *p) {
F1: if (p->op==POP_OP) {
F2:     p->pcell=location[mypid]->pcell;
F3:     location[mypid]=NULL;
    }
}
```

```
void TryCollision(ThreadInfo*p,ThreadInfo *q) {
C1:  if(p->op==PUSH) {
C2:    if(CAS(&location[him],q,p))
C3:      return TRUE;
C4:    else
C5:      return FALSE;
     }
C6:  if(p->op==POP) {
C7:    if(CAS(&location[him],q,NULL)){
C8:      p->cell=q->cell;
C9:      location[mypid]=NULL;
C10:     return TRUE
       }
C11:   else
C12:     return FALSE;
     }
}
```

Elimination
Backoff
Stack Codes



Collision Array

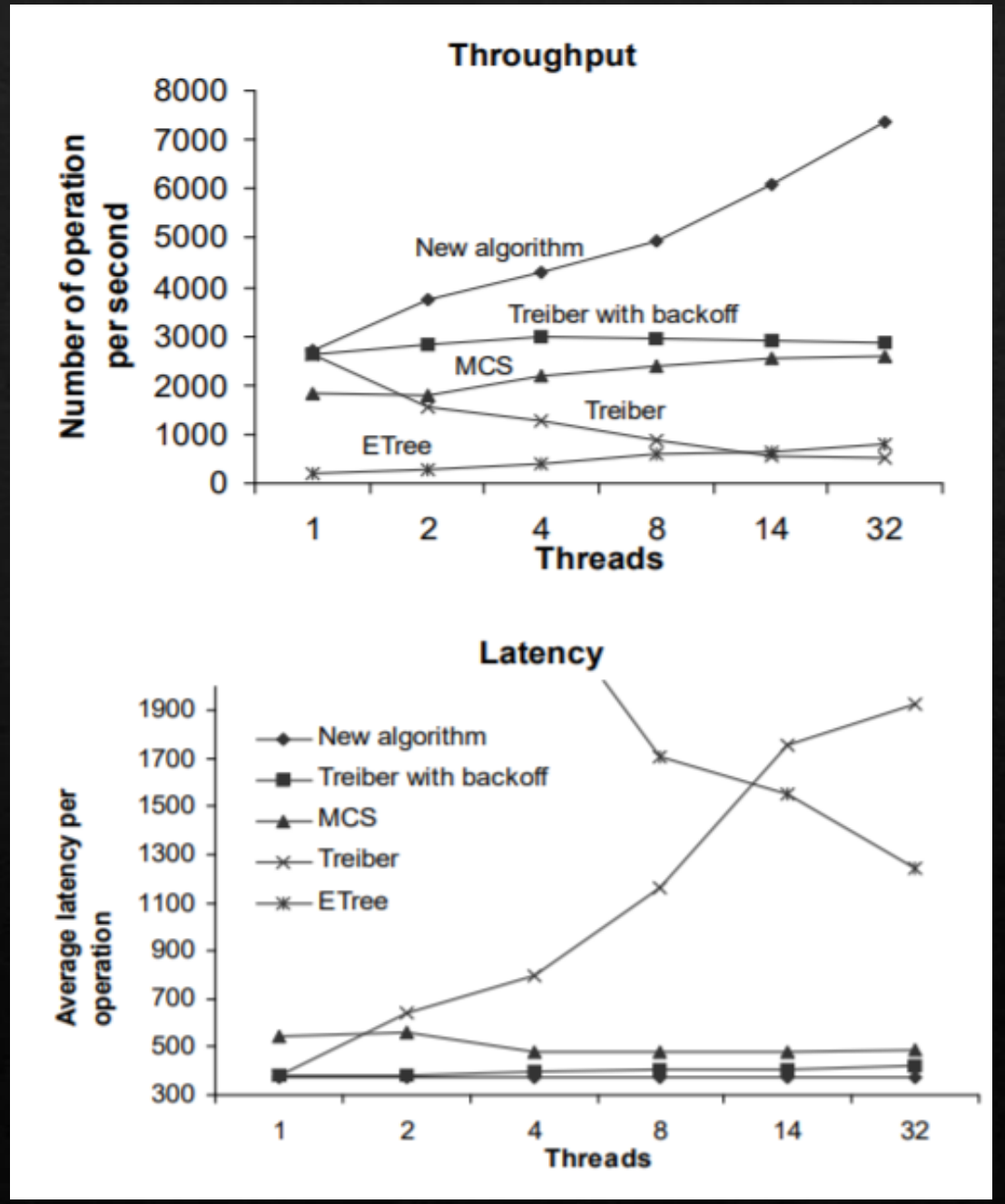


Collision Scenarios

Correctness

- Operations can only collide with operations of the opposite type
- •TryCollision can succeed only in case of a successful CAS in line C2 (for a push operation) or in line C7 (for a pop operation). Such a CAS changes the value of the other thread's cell in the location array, thus exchanging values with it and returns without modifying the central stack object. From the code, before calling TryCollision op has to execute line S9, thus verifying that it collides with an operation of the opposite type.
- If op is a passive colliding-operation, then op performs FinishCollision, which implies that op failed in resetting its entry in the location array (in line S10 or s19). Let op1 be the operation that has caused op's failure by writing to its entry. From the code, op1 must have succeeded in TryCollision , thus, it has verified in line S9 that its type is opposite to that of op.

Throughput and latency of different stack implementations with varying number of threads. Each thread performs 50% pushs, 50% pops.



Lock Freedom

Algorithm is lock free ,if system as a whole makes progress . Let op be some operation. If op manages to collide, then op's linearization has occurred , and op does not iterate anymore before returning. Otherwise, op calls TryPerformStackOp . if TryPerformStackOp returns TRUE, op immediately returns, and its linearization has occurred ; if, on the other hand, TryPerformStackOp returns FALSE, this implies that the CAS performed by it has failed, and the only possible reason for the failure of the CAS by op is the success of a CAS on phead by some other operation, thus whenever op completes a full iteration, some operation is linearized.

Adaptive

- Algorithm first tries to access the central stack and only if that fails to back off to the elimination array. This allows us, in case of low loads, to avoid the collision array altogether, thus achieving the latency of a simple stack .
- Each thread t keeps a value spin which holds the amount of time that t should delay while waiting to be collided. The spin value may change within a predetermined range. When t successfully collides, it increments a local counter. When the counter exceeds some limit, t doubles spin. If t fails to collide, it decrements the local counter. When the counter decreases bellow some limit, spin is halved.

Reference

- https://people.csail.mit.edu/shanir/publications/Lock_Free.pdf

Thank You.