CS528
Caching and Multi-threading

A Sahu
Dept of CSE, IIT Guwahati

Program Cache Behavior: Hit/Miss

Cache model

- Direct mapped 8 word per line

Program

```
int A[128];
for(i=0;i<128;i++){
    A[i]=i;
}
```

- Assume &A=000000, **Behavior of only Data**
- Scalar variable {i} mapped to register
- Data have to moved from cache/memory

Cache perf. : Data Size <= Cache Size

```
int A[128];
for(i=0;i<128;i++){
    A[i]=i;
}
```

Scalar mapped to register
Vector mapped to memory

1:7= 1miss:7hit

1:7	0	A[0]	A[1]	A[2]					A[7]
2:14	1	A[8]	A[9]						A[15]
3:21	2	A[16]	A[17]						A[23]
	14								
16:112	15								A[127]

Strided access: Reduce locality

```
for(i=0;i<N;i++){
    for (j=0;j<N;j++){
        a[i][j]=i*j
      }
}       //*(a+i*N+j), j++
```

Row major access: Stride 1, improve locality, cache hit

```
for(i=0;i<N;i++){
    for (j=0;j<N;j++){
        a[j][i]=i*j
      }
}   //*(a+j*N+i), j++
```

Column major access: Stride N, No locality, cache miss dominates

Matrix mult.c

```
int A[8][8], B[8][8], C[8][8];
for(i=0;i<8;i++){
        for(j=0;j<8;j++){
          S=0;
          for(k=0;k<8;k++)
                  S=S+B[i][k]*C[k][j];
          A[i][j]=S;
        }
}
```

Data Size > Cache Size

- (64+64+64) > 128 words
- When we get into cache it can take benefit

```
for(k=0;k<8;k++)
            S=S+B[i][k]*C[k][j];
```

- Inner loop execute for **64 times**
  - We have to get B[j] once will have 1miss/7 hit
  - C[k] have to bring every time 8miss
  - Total = **7h+9m**
- 2nd loop A have one miss in 8 access (1miss/7hit)
  - Total for A= 8m+56h
- Total program : 64*(**7h+9m**)+8m+56h=504h+584m
- ***Miss Probability = 584/(504+584)=0.5367***

Improving Locality

Matrix Multiplication example

$$[C] = [A] \times [B]$$

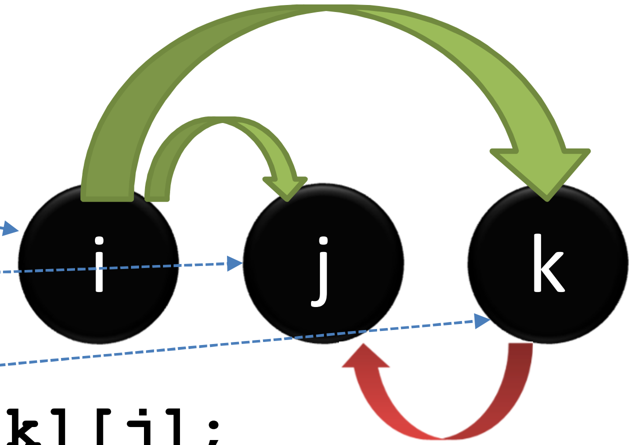
$L \times M$ $\quad$ $L \times N$ $\quad$ $N \times M$

Cache Organization for the example

- Cache line (or block) = 8 matrix elements.
- Matrices are stored row wise.
- **Cache can't accommodate a full row/column.**
  - **L, M and N are so large w.r.t. the cache size**
  - After an iteration along any of the three indices, when an element is accessed again, it results in a miss.
- Ignore misses due to conflict between matrices.
  - As if there was a **separate cache for each matrix**.

Matrix Multiplication : Code I

```
for (i = 0; i < L; i++)
  for (j = 0; j < M; j++)
    for (k = 0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
```



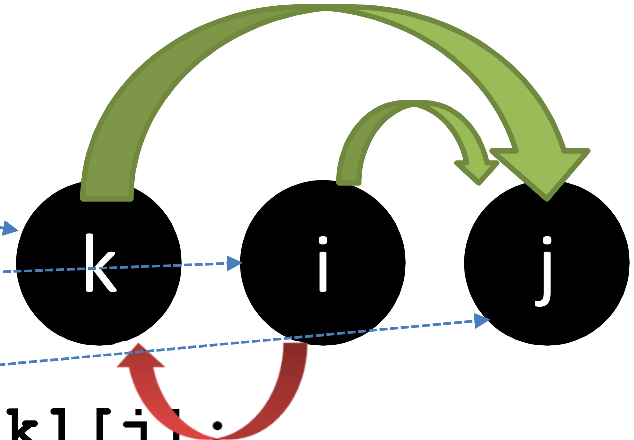
	C	A	B
accesses	LM	LMN	LMN
misses	LM/8	LMN/8	LMN

Total misses = LM(9N+1)/8

L=M=N=100; miss=100*100*901/8=1,126,250

Matrix Multiplication : Code II

```
for (k = 0; k < N; k++)
  for (i = 0; i < L; i++)
    for (j = 0; j < M; j++)
      C[i][j] += A[i][k] * B[k][j];
```

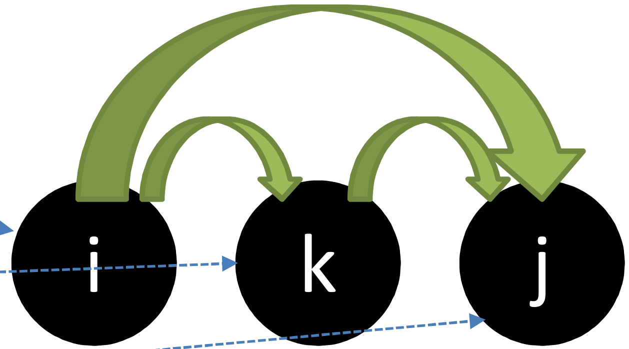


	C	A	B
accesses	LMN	LN	LMN
misses	LMN/8	LN	LMN/8

```
Total misses = LN(2M+8)/8
L=M=N=100; miss=100*100*208/8=260,000
```

Matrix Multiplication : Code III

```
for (i = 0; i < L; i++)
  for (k = 0; k < N; k++)
    for (j = 0; j < M; j++)
      C[i][j] += A[i][k] * B[k][j];
```



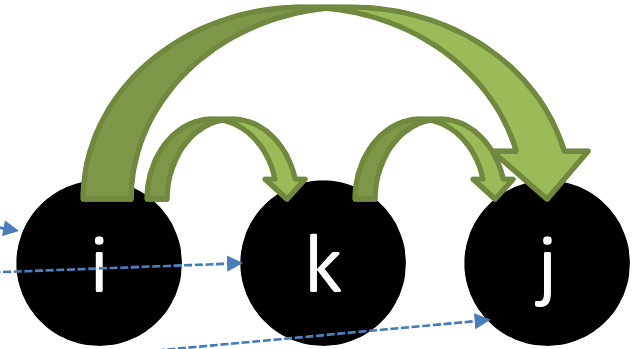
	C	A	B
accesses	LMN	LN	LMN
misses	LMN/8	LN/8	LMN/8

Total misses = LN(2M+1)/8

L=M=N=100; miss=100*100*201/8=251,250

Matrix Multiplication : Code III

```
for (i = 0; i < L; i++)
  for (k = 0; k < N; k++)
    for (j = 0; j < M; j++)
       C[i][j] += A[i][k] * B[k][j];
```



All most all modern processor uses

- Cache block pre-fetch
- When ith block is getting used i+1 block prefetched
- **Perfect overlap : only three cache miss**
  - **Each for A, B, C**

See the cachegrind demo of matmul code

Case Study : Jacobi-Algorithm Stencil based iterative method

```
double D[2][kmax+1][kmax+1];
int t0=0,t1=1;
for(time=0; time<itertime; time++){
   for(i=1;i<kmax,i++) {
      for(j=1;j<kmax;k++){
        //Sum of neighbours
        S=  D[t0][i+1][j]+D[t0][i-1][j]
           + D[t0][i][j-1]+ D[t0][i][j+1];
        D[t1][i][j]=S*0.25;
       }
   tmp=t0; t0=t1; t1=tmp;  //swap array
}
```

Inner loop : Bc= 5Words/4F=1.25 W/F

WriteNot Allocate: Bc=4W/4F=1.0F

Jacobi-Algorithm

```
double D[2][kmax+1][kmax+1];
int t0=0,t1=1;
for(time=0; time<itertime; time++){
   for(i=1;i<kmax,i++) {
      for(j=1;j<kmax;k++){
        //Sum of neighbours
        S=  D[t0][i+1][j]+D[t0][i-1][j]
           + D[t0][i][j-1]+ D[t0][i][j+1];
        D[t1][i][j]=S*0.25;
      }
   tmp=t0; t0=t1; t1=tmp;  //swap array
}
```

Assume Row i and i-1 with no cost : if cache is capable to holding two rows

Bc=1W/4F=0.25W/F

Algorithm Classification and Access Optimization

- O(N)/O(N) : If the # of arithmetic Ops and data transfer (LD/ST) are proportional to Loop Length N
  - Optimization potential is limited
  - Example Scalar Product, vector add, sparse MVM
- Memory bound for large N
- Compiler generated code achieve good perf.
  - Using software pipelining and loop nests

Loop fusion for O(N)/O(N)

```
for(i=0;i<N;i++)
    A[i]=B[i]+C[i];   //Bc=3W/1F
for(i=0;i<N;i++)
    Z[i]=B[i]+E[i];   //Bc=3W/1F
```

```
for(i=0;i<N;i++){
    A[i]=B[i]+C[i];
    Z[i]=B[i]+E[i];
}
```

B_C=5W/2F

No need to B[i]

$O(N^2)/O(N^2)$: OPS/DataTransfer

- Typical two loop nests with loop strip count N
  - $O(N^2)$ operation for $O(N^2)$ loads and stores
- Example: dense MVM, Mat add, MatTrans
- MVM : -> Covert both access to row access

```
for(i=0;i<N;i++) {
      tmp=C[i]
      for(j=0;j<N;j++)   tmp=A[i][j]*B[j]
      C[i]=tmp
}
```

  - Row I of A and vector B
  - Original Bc=2W/2F but → 2W*m/2F
  - m is miss rate of cache for Row access

$O(N^3)/O(N^2)$: OPS/DataTransfer

- Typical three loop nests
  - $O(N^3)$ operation for $O(N^2)$ loads and stores
- Example: dense Matrix Mulltiplication
- Implementation of cache Bound
  - Already studied : loop interchange
  - **Blocking : Strassen multiplication, will be discussed later**

Threading

Threading Language and Support

- Pthread: POSIX thread
  - Popular, Initial and Basic one
- Improved Constructs for threading
  - c++ thread : available in c++11, c++14
  - Java thread : very good memory model
    - Atomic function, Mutex
- Thread Pooling and higher level management
  - OpenMP (loop based)
  - Cilk (dynamic DAG based)

Programming with Threads

- Threads
- Shared variables
- The need for synchronization
- Synchronizing with semaphores
- Thread safety and reentrancy
- Races and deadlocks

Traditional View of a Process

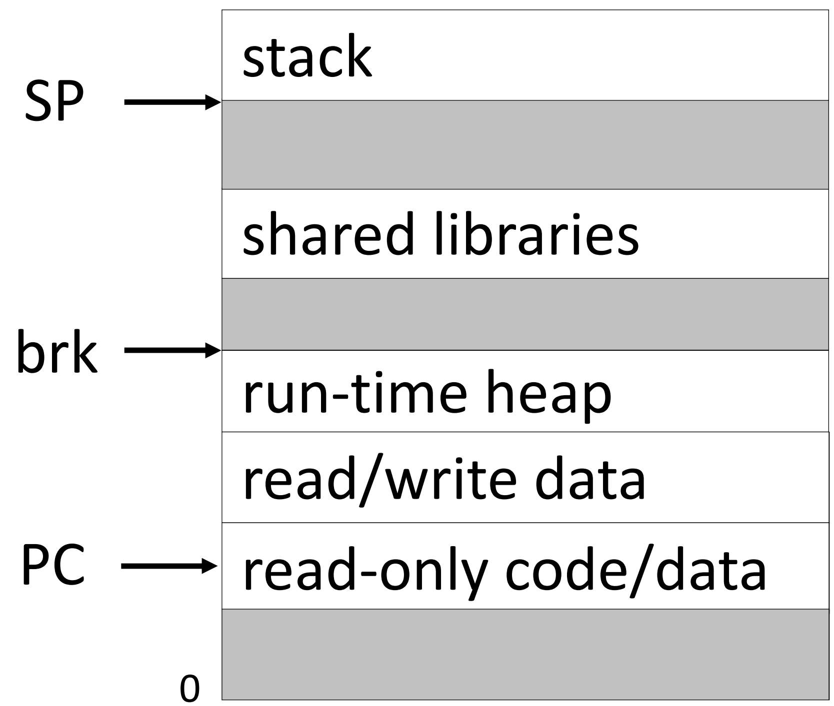
- Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernel context:
- VM structures (VMem)
- Descriptor table
- brk pointer

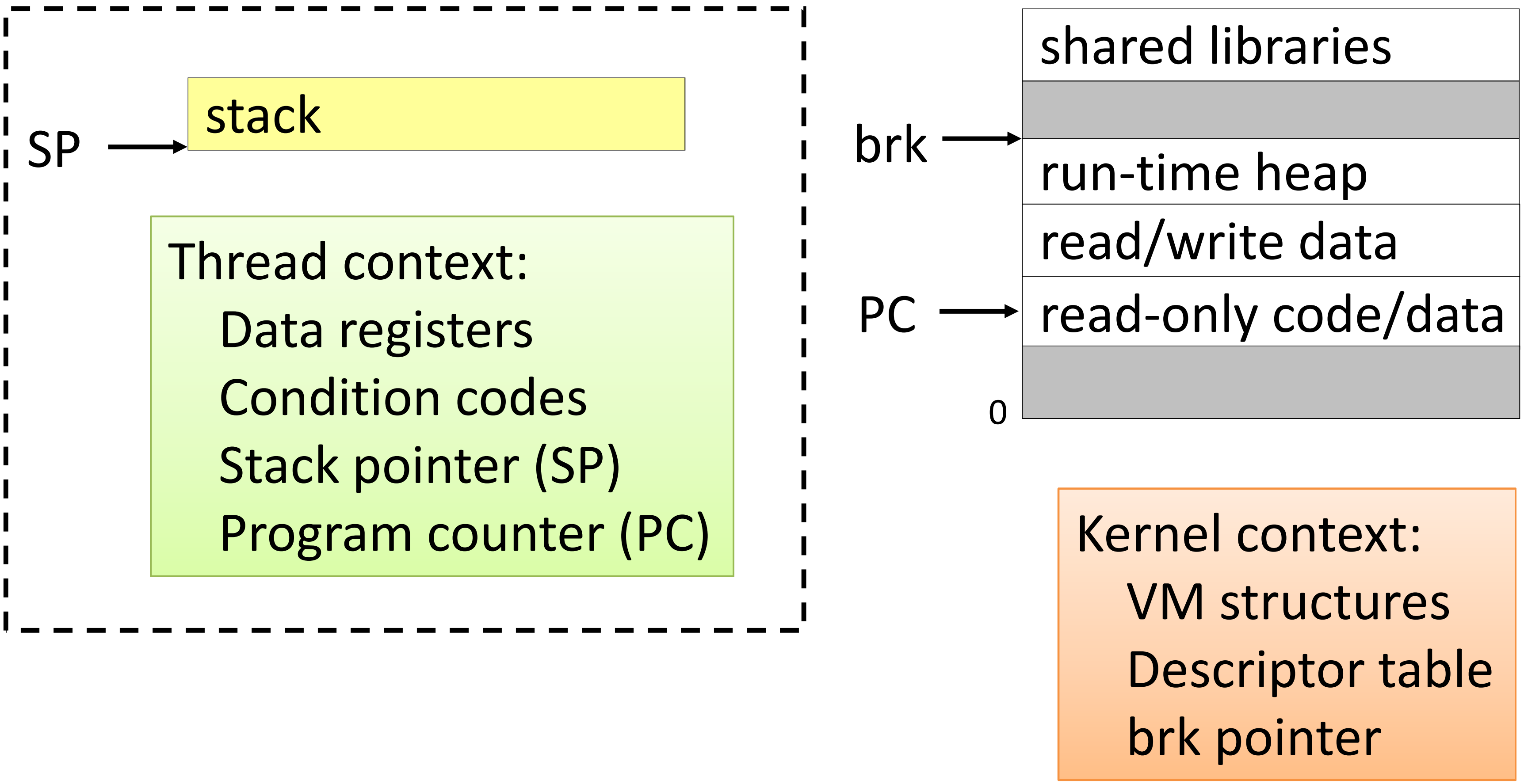
Code, data, and stack



Alternate View of a Process

- Process = thread+ code, data & kernel context

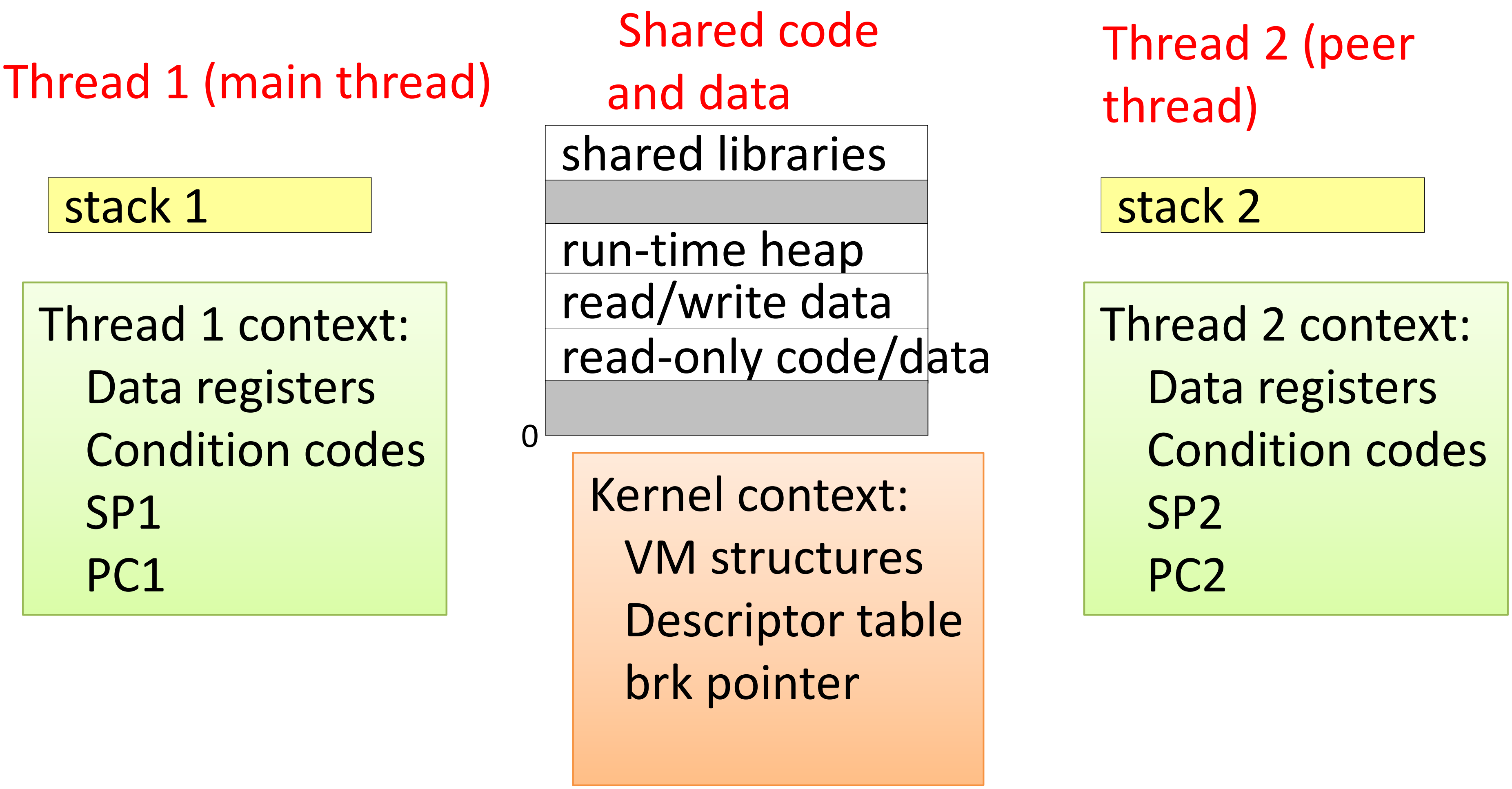
Thread (main thread) Code and Data



A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its *own* logical control flow (sequence of PC values)
  - Each thread *shares* the same code, data, and kernel context
  - Each thread has its own thread id (TID)

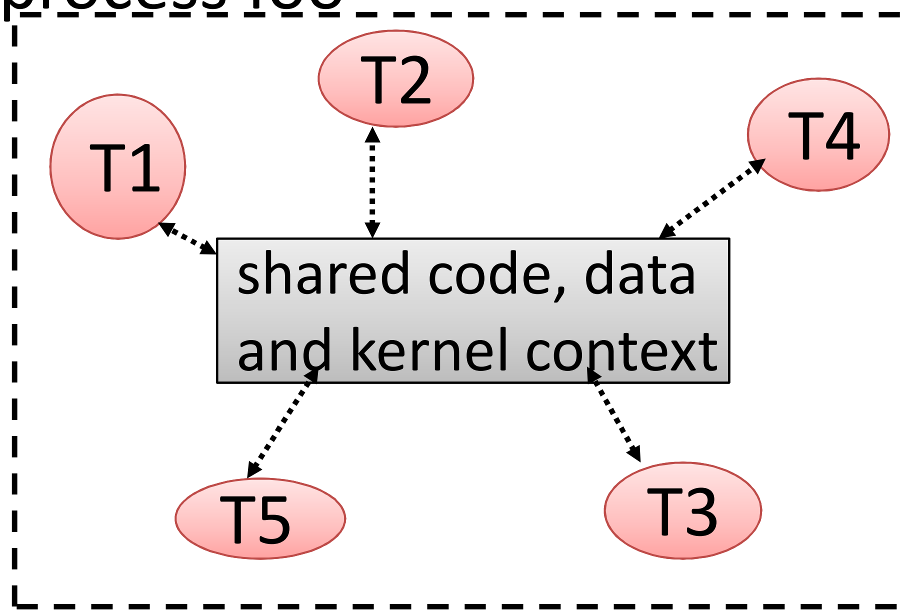
A Process With Multiple Threads



Logical View of Threads

- Threads associated with a process form a pool of peers
  - Unlike processes, which form a tree hierarchy

Threads associated with process foo



Process hierarchy