CS343: Operating System

Threading and Synchronization

Lect16 : 04th Sept 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Threading
- Threading Examples
- Thread mappings
  - Pthread/Uthread, Kthread, Hthread
- Synchronization

Pthread, C++ Thread, Cilk and OpenMP

Pthread

```
pthread_t tid1, tid2;
pthread_create(&tid1,NULL,Fun1, NULL);
pthread_create(&tid2,NULL,Fun2, NULL);
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
```

C++ thread

```
  thread t1(Fun1);
  thread t1(Fun2, 0, 1, 2);
    // 0, 1,2 param to Fun2
  t1.join();
  t2.join();
```

Posix Threads (Pthreads) Interface

- **Creating and reaping threads**
  - `pthread_create`, `pthread_join`
- **Determining your thread ID :** `pthread_self`
- **Terminating threads**
  - `pthread_cancel`, `pthread_exit`
  - `exit` [terminates all threads] , `return` [terminates current thread]
- **Synchronizing access to shared variables**
  - `pthread_mutex_init`, `pthread_mutex_[un]lock`
  - `pthread_cond_init`, `pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

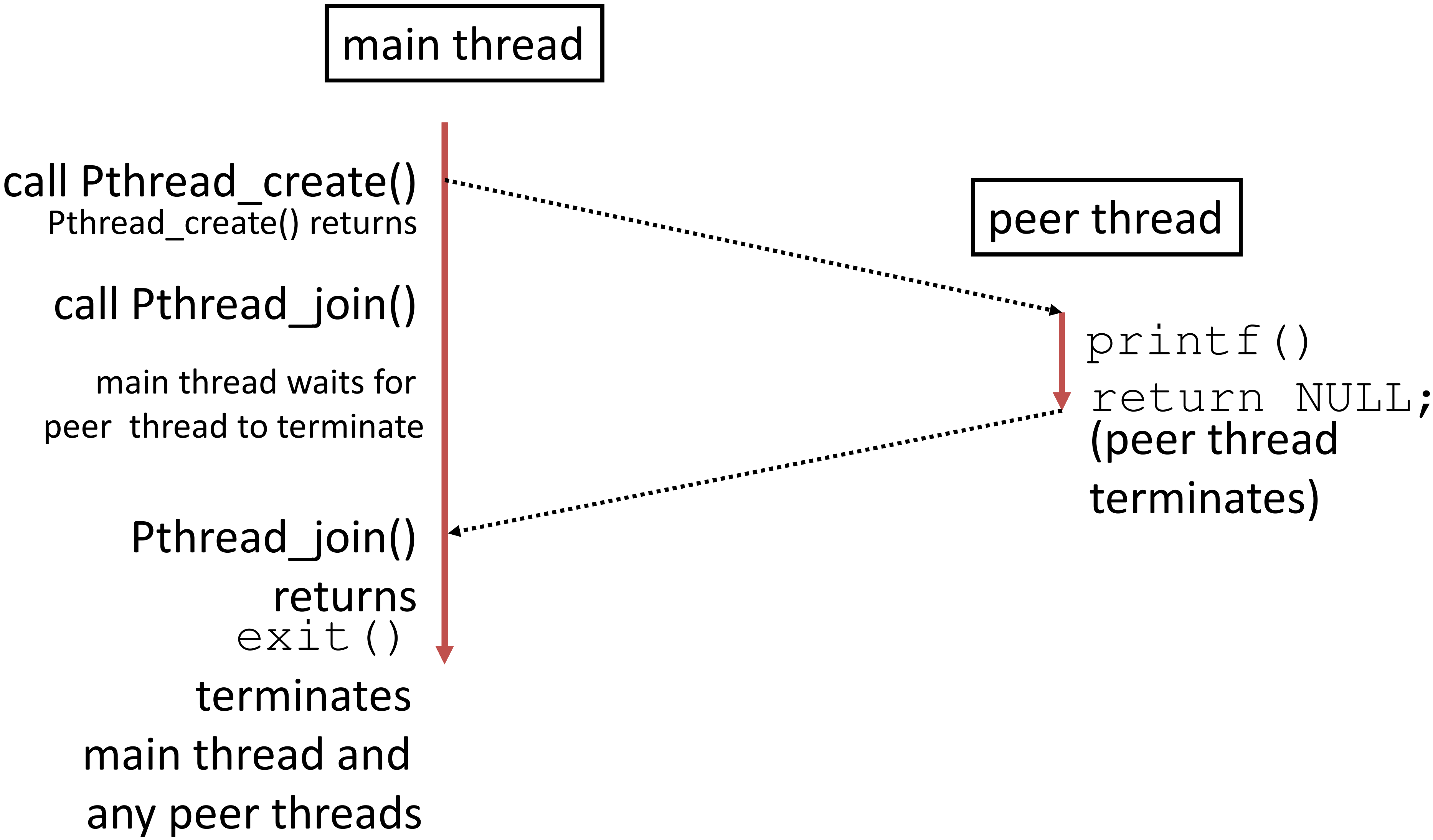
```
/* thread routine */
void *HelloW(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}

int main() {
  pthread_t tid;
  pthread_create(&tid, NULL, Hellow, NULL);
  pthread_join(tid, NULL);
  return 0;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Execution of Threaded “hello, world”



Pros and Cons: Thread-Based Designs

- **+ Easy to share data structures between threads**
  - E.g., logging information, file cache
- **+ Threads are more efficient than processes**
- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - Ease of data sharing is greatest strength of threads
  - Also greatest weakness!

VectorSum Serial

```
int A[VSize], B[VSize], C[VSize];


void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

Suppose Size=1000

0-249	**250-499**	**500-749**	**750-999**
T1	**T2**	**T3**	**T4**

VectorSum Serial

```
int A[VSize], B[VSize], C[VSize];


void VectorSumSerial(){
   for( int j=0;j<SIZE;j++)
     A[j]=B[j]+C[j];
}
```

- Independent
- Divide work into equal for each thread
- Work per thread: Size/numThread

VectorSum Parallel

```
void *DoVectorSum(void *tid){
    int j, SzPerthrd, LB, UB, TID;
     TID= *((int *)tid);
     SzPerthrd=(VSize/NUM_THREADS);
     LB= SzPerthrd*TID;UB=LB+SzPerthrd;


    for(j=LB;j<UB;j++)
      A[j]=B[j]+C[j];
}
```

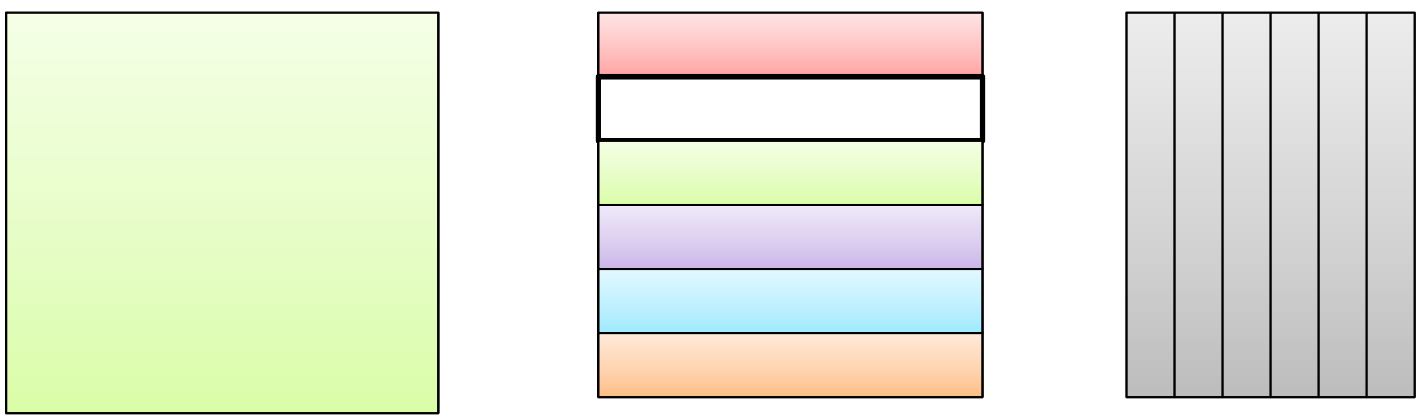
VectorSum Parallel

```
int main(){
    int i;
    pthread_t thread[NUM_THREADS];
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&thread[i],
        NULL, DoVectorSum, (void*)&i);
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(thread[i], NULL);

    return 0;
}
```

Matrix multiply and threaded matrix multiply

- Matrix multiply: C = A × B

$$C[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j]$$



Matrix multiply and threaded matrix multiply

- Matrix multiply: C = A × B

$$C[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j]$$

- Divide the whole rows to T chunks
  - Each chunk contains : N/T rows, Assume N%T=0

Matrix multiply Serial

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

Matrix Pthreaded: RowWise

```
void * DoMatMulThread(void *arg){
     int i,j,k,S,LB,UB, TID, ThrdSz;
     TID=*((int *)arg);ThrdSz=Size/NumThrd;
     LB=TID*ThrdSz;UB=LB+ThrdSz;
     for(i=LB;i<UB;i++)
          for(j=0;j<Size;j++){
          S=0;
          for(k=0;k<Size;k++)
            S=S+A[i][k]*B[k][j];
          C[i][j]=S;
          }
 }
```

Matrix Pthreaded: RowWise

```
int main(){
    pthread_t thread[NumThread];
    int t;
    Initialize();
     for(t=0; t<NumThread; t++)
          pthread_create(&thread[t], NULL,
          DoMatMulThread, &t);
     for(t=0; t<NumThread; t++)
          pthread_join(thread[t], NULL);

     TestResult();
     return 0;
}
```

Estimating π using Monte Carlo

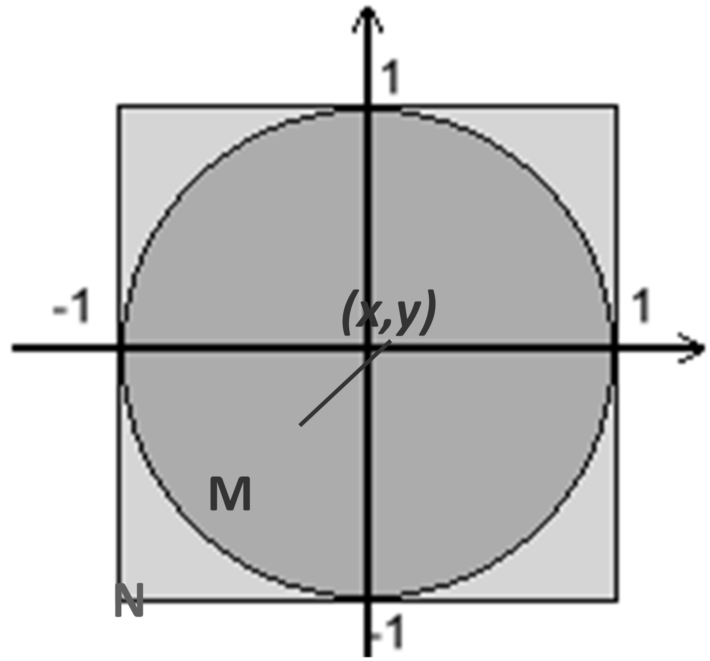
- The probability of a random point lying inside the unit circle:

$$\mathbf{P}\left(x^2 + y^2 < 1\right) = \frac{A_{circle}}{A_{square}} = \frac{\pi}{4}$$

- If pick a random point *N* times and *M* of those times the point lies inside the unit circle:

$$\mathbf{P}^{\circ}\left(x^2 + y^2 < 1\right) = \frac{M}{N}$$

- If *N* becomes very large, $P=P^0$



$$\pi = \frac{4 \cdot M}{N}$$

Value of PI: Monte-Carlo Method

```
void MontePI(){
   int count=0,i;
    double x,y,z;
    for ( i=0; i<niter; i++) {
        x = (double)rand()/RAND_MAX;
        y = (double)rand()/RAND_MAX;
        z = x*x+y*y;
        if (z<=1) count++;
        }
    pi=(double)count/niter*4;
}
```

PI- Multi-threaded

- 1 thread you are able to generate N points
  - Suppose M points fall under unit circle
  - PI=4M/N
- With 10 thread generate 10XN points and calculate more accurately
  - Each thread calculate own value of PI (or M)
  - Average later on (or recalculate PI from collective M)

Value of PI: Pthreaded

```
int main(){
   pthread_t thread[NumThread]; double pi;
   int t, at[NumThread], count, TotalIter;
    for(t=0; t<NumThread; t++)
      pthread_create(&thread[t], NULL,
          DoLocalMC_PI, &t);
    for(t=0; t<NumThread; t++)
      pthread_join(thread[t], NULL);
    for(t=0;t<NumThread;t++) count+=LCount[t];
    TotalIter=niter*NumThread;
    pi=((double)count/TotalIter)*4;
    return 0;
}
```

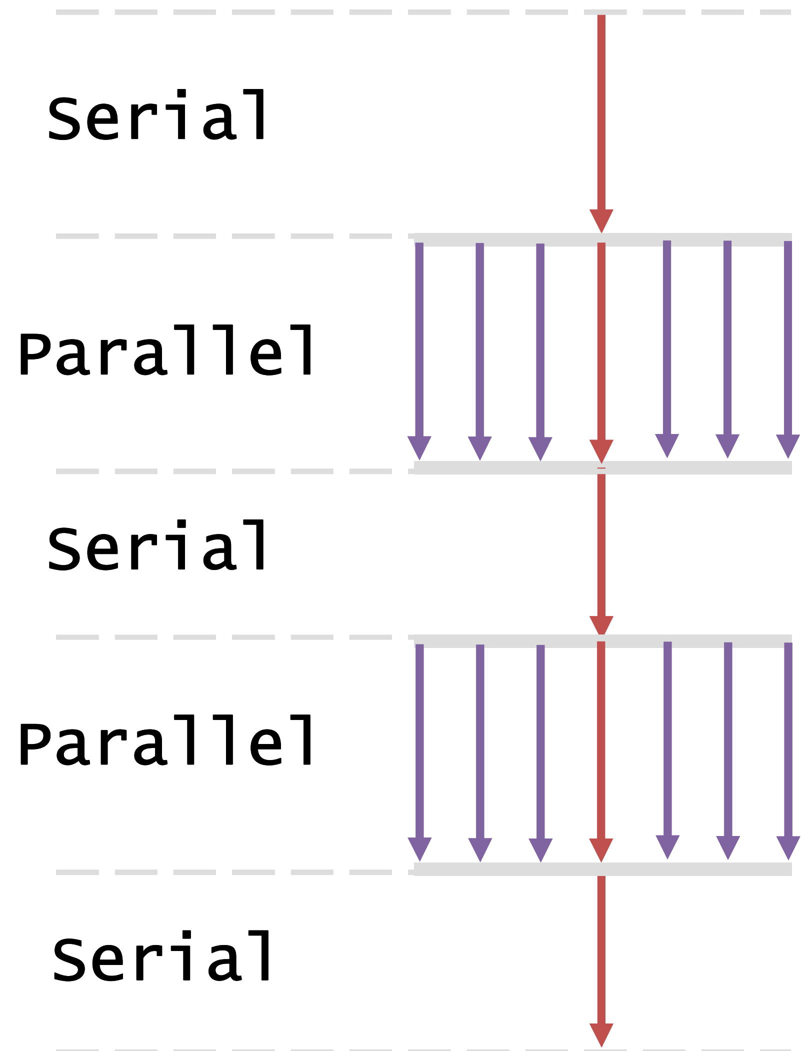
Value of PI: Pthreaded

```
int LCount[NumThread];
void *DoLocalMC_PI(void *aTid){
   int tid, count, i; double x,y,z;
    tid= *((int *)aTid);
    count=0;LCount[tid]=0;
    for ( i=0; i<niter; i++) {
        x = (double)rand()/RAND_MAX;
        y = (double)rand()/RAND_MAX;
        z = x*x+y*y; if (z<=1) count++;
        }
    LCount[tid]=count;
}
```

Performance of Parallel Program
(Amdahl's Law)

Example: OpenMP Parallel Program

```
printf("begin\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("done\n");
```



Speed up and efficiency

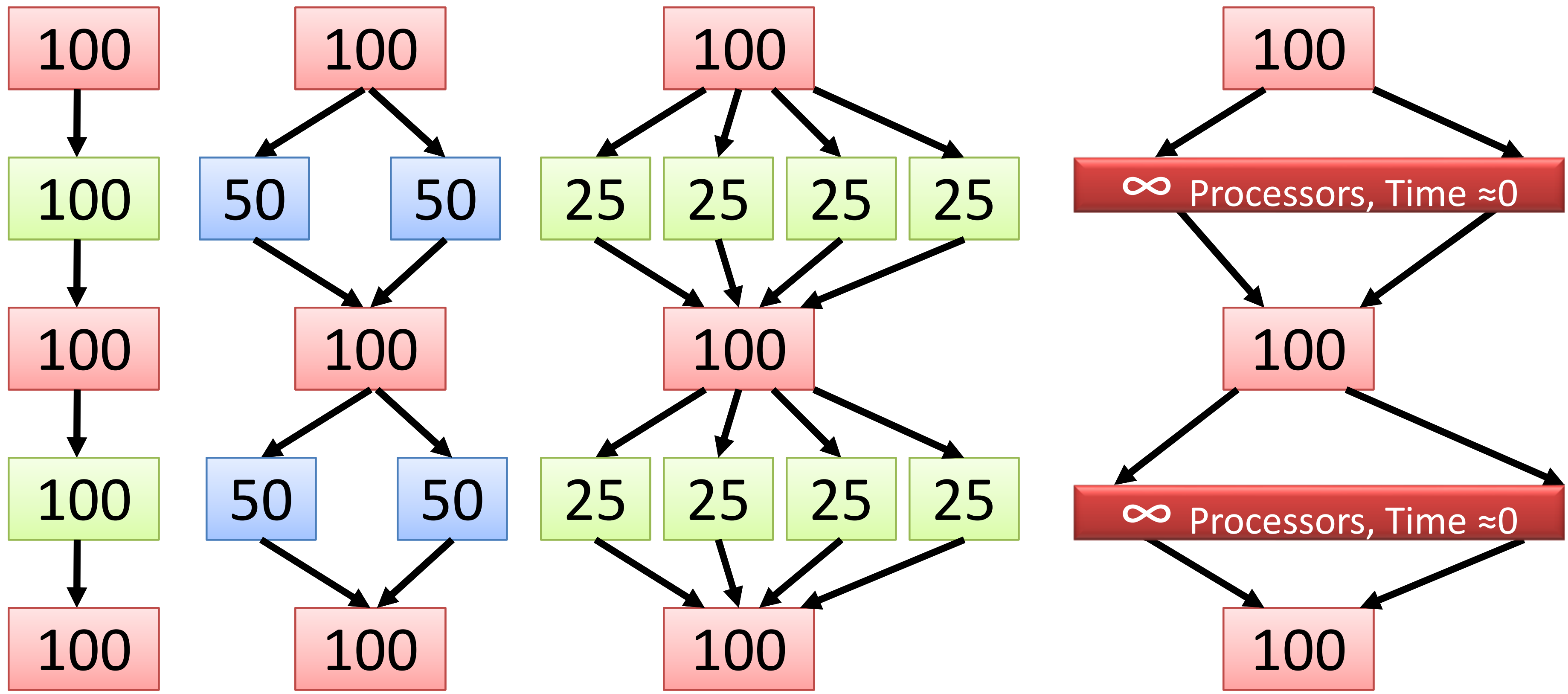
- **Notion :** T_1 =Time on Uni-processor, T_p= Time on p Processors

$$\text{Speed up} = S_p = T_1/T_p \leq p$$

$$\text{Efficiency} = E_p = T_1/(p.T_p)$$

- Usually $S_p < p$ or $E_p < 1$ due to overhead
- Some time superliner speed up reported ($S_p > p$ or $E_p > 1$)
  - Failure to use the best sequential algorithm
  - Advantage due to larger memory

Amdahl's law



Work 500,
Time 500
Sp=1X

Work 500,
Time 400
Sp=1.25X

Work 500,
Time 350
Sp=1.4X

Work 500,
Time 300
Sp=1.7X

Amdahl's Law

$$\text{Serial fraction} = s = \frac{T_s}{T_1}$$

$$T_p = T_s + \frac{T_1 - T_s}{p}$$

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_s + \frac{T_1 - T_s}{p}} = \frac{T_1}{T_s(1 - \frac{1}{p}) + \frac{T_1}{p}}$$

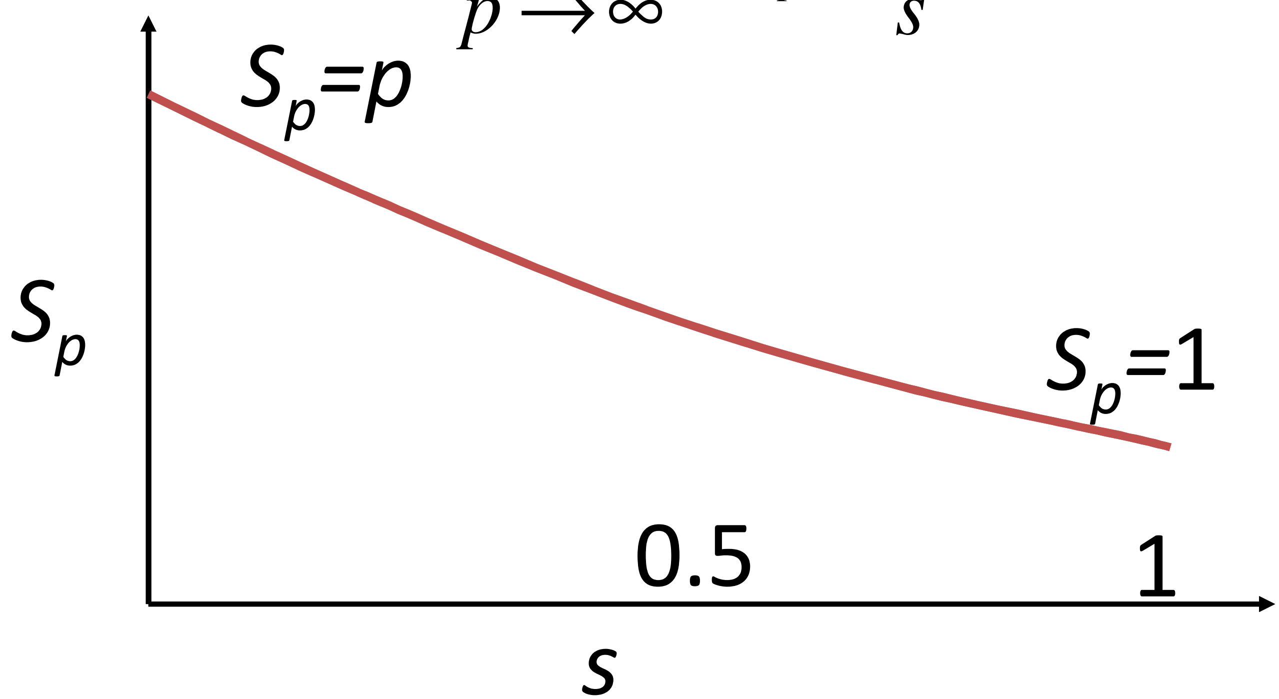
Amdahl's Law

$$S_p = \frac{T_1}{T_s(1-\frac{1}{p}) + \frac{T_1}{p}}$$

$$Sp = \frac{1}{s(1-\frac{1}{p}) + \frac{1}{p}} = \frac{p}{s(p-1)+1}$$

$$s = \frac{T_s}{T_1}$$

$$\underset{p \to \infty}{Lt} \quad S_p = \frac{1}{s}$$



Assumption behind Amdahl's Law

- All the processors are homogeneous
- All the communication costs are zero
- All the memory accesses takes unit time (PRAM)
- All the parallel section are purely parallel: Divisible load

$$S_p = \frac{1}{s(1 - \frac{1}{p}) + \frac{1}{p}} =$$

$$\underset{p \to \infty}{Lt} \quad S_p = \frac{1}{s}$$

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**, Windows threads, Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows , Solaris, Linux

User Threads
and
Kernel Threads

Kernel Thread

- Kernels are generally multithreaded (kthread)
- Pthread : user level thread
- To make concurrency cheaper the execution aspect of process is separated out into threads.
- OS manages/schedule threads and processes
- All thread operations are implemented in the kernel
- OS managed threads are called kernel-level threads or light weight processes
  - Window NT: Thread, Solaris: LWP

Kernel Thread

- The kernel knows about and manages the threads
- No runtime system is needed in this case.
- Kernel
  - A thread table that keeps track of all threads in the system.
  - In addition, process table to keep track of processes.
- OS kernel provides system call to create and manage threads

Advantage of Kernel Thread

- **ADV: As kernel has full knowledge of all threads**
  - Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- **ADV: kthreads are especially good for apps that frequently block.**

DisAdv of Kernel Thread

- **DISADV: The kthreads are slow and inefficient.**
  - threads operations are hundreds of times slower than that of user-level threads.
- **DISADV: Since kernel must manage and schedule threads as well as processes.**
  - It require a full thread control block (TCB) for each thread to maintain information about threads.
  - As a result there is significant overhead and increased in kernel complexity.

User level thread

- Kthreads make concurrency much cheaper than process
  - because, much less state to allocate and initialize.
- However, for fine-grained concurrency kthreads still suffer from too much overhead.
  - Thread operations still require system calls.
  - Ideally, we require thread operations to be as fast as a procedure call.
- For fine grained concurrency we need "cheaper" threads.
- To make threads cheap and fast, they need to be **implemented at user level.**

User level thread

- User-Level threads are managed
  - Entirely by the run-time system (user-level library).
- Kernel knows nothing about user-level threads
  - Manages them as if they were single-threaded processes.
- User-Level threads are small and fast

User level thread : pthread

- Each **Uthread** is represented by a PC,register,stack, and small thread control block.
- Done via procedure call. i.e **no kernel involvement**
  - Creating a new thread, switiching between threads, and synchronizing threads are
- Uthreads are **100x** faster than Kthreads.

Uthread : Advantage

- **The most obvious advantage of this technique**
  - uthread package can be implemented on an OS that does not support kthreads.
  - uthreads does not require modification to OS
- **Simple Representation**
  - Each thread is represented simply by a PC, registers, stack and a small control block TCB
  - all stored in the user process address space.
- **Simple Management**
  - Creating a thread, thread switching and synch threads
  - All be done without intervention of the kernel.
- **Fast and Efficient**
  - Thread switching is not much more expensive than a procedure call

Uthread: Disadvantage

- Uthreads are not a perfect solution as with everything else, they are a trade off.
  - Since, Uthreads are invisible to the OS they are not well integrated with the OS, As a result, Os can make poor decisions
  - Scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock.
  - Solving this requires communication between between kernel and user-level thread manager.

Uthread: Disadvantage

- Lack of coordination between threads and OS
  - Process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within.
- Uthreads requires ***non-blocking systems call*** i.e., a multithreaded kernel.
  - Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes.
  - For example, if one thread causes a page fault, the process blocks.