Lecture 21: Introduction to Multithreading

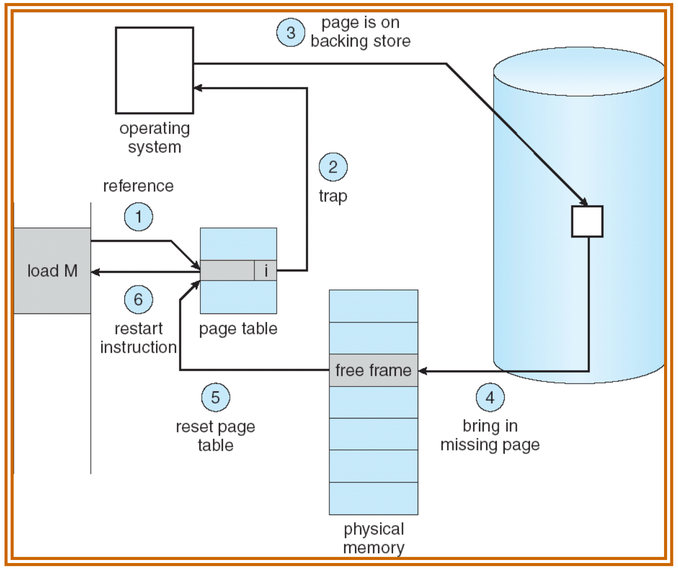
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture



Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

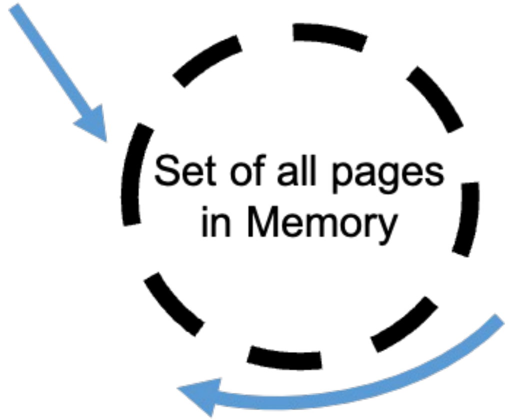
Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

Single Clock Hand in Clock Algorithm:

Advances only on page fault!

Check for pages not used recently

Mark pages as not used recently

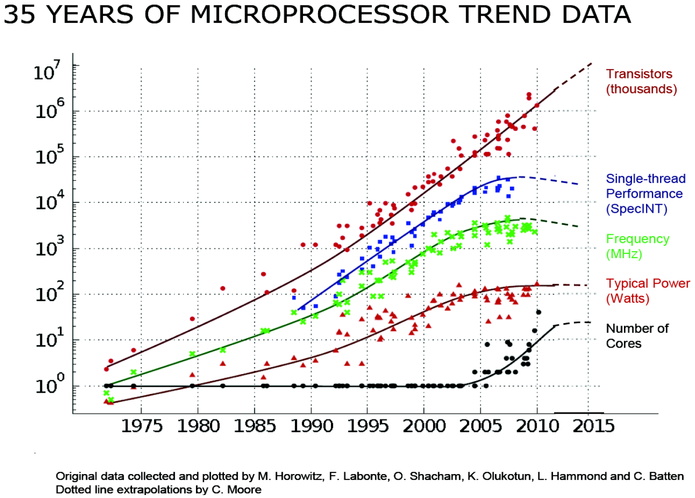


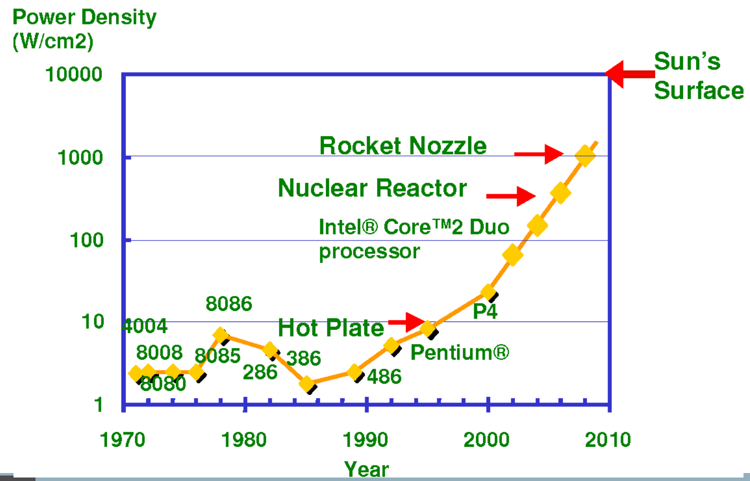
- Demand paging
  - Use DRAM as a cache for recently used pages and move the unused pages to disk
  - Bring back the page from disk to ram on demand (upon page fault)
- Page replacement policies (Suppose we have the following page reference stream: A B C A B D A D B C B)
  - FIFO
  - MIN / LRU
- Approximating LRU using clock algorithm (each page has a “use” bit)

Today's Class

- Why multicore processors
- Motivation for multithreading
- Thread creation and termination

Processor Technology Trend

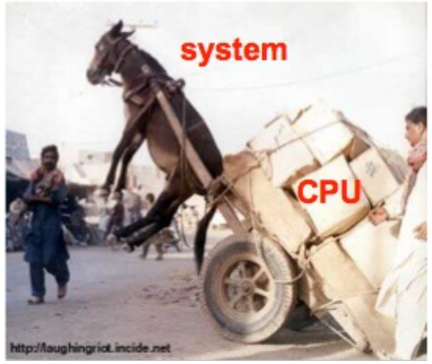


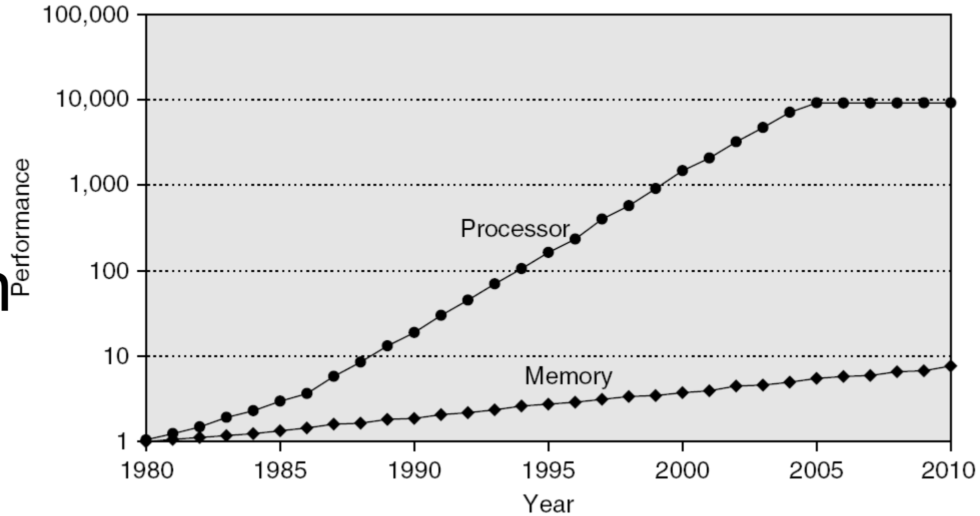


- Moore's law (1964)
  - Area of transistors halves roughly every two years
    - I.e., Total transistors on processor chip gets doubled roughly every two years
- Dennard scaling (1974)
  - Power for fixed chip are remains almost constant as transistors become smaller
- No more free lunch!
  - Thermal wall hit around 2004
  - Power is proportional to cube of frequency
    - It restricts frequency growth, but opens up the multicore era

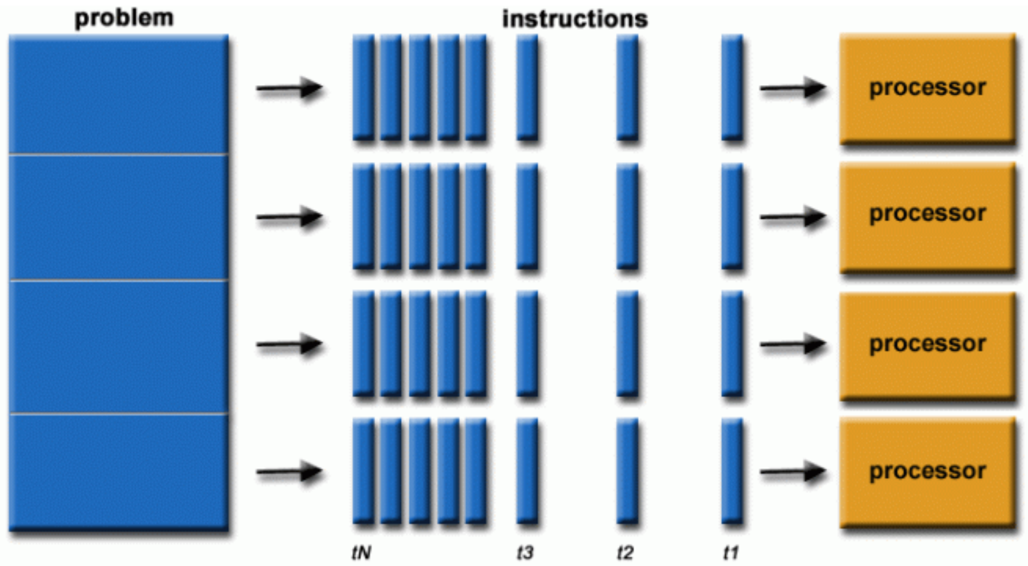
Adding More Cores Improves performance?

- Computation is just part of the picture
- Memory latency and bandwidth
  - CPU rates have increased 4x as fast as memory over last decade
  - Bridge speed gap using memory hierarchy
  - Multicore exacerbates demand
- Inter-processor communication
- Input/Output





Free Lunch is Over!



- Industrial and commercial users of parallel computing
  - BigData, Databases, Data mining
  - Artificial intelligence
  - Oil exploration
  - Web search engines, web based business services (Facebook, Twitter, etc.)
  - Medical imaging and diagnosis
  - Financial and economic modelling
  - Advanced graphics and virtual reality
  - Collaborative work environment

Let's Revisit Array Sum Program (Lecture #9)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

- Note that it is an MPI program that is aimed to be run on a distributed memory machine (cluster / supercomputer)
- However, we can still run it on a single machine (laptop/desktop) with multicore processors
    - Total process equals to total cores

IPC Using MPI Within a Multicore Processor

- Inter-process communication in shared memory (only)
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Separate heap, stack, .text segment, etc. for each process
  - Same copy of .text segment in each process
- Separate page table for each process
- Cost of IPC may exceed the cost of actual computation!

Let's Revisit Array Sum Program (Lecture #8)

```
int main() {
    shm_t* shm = setup();
    sem_init(&shm->mutex, 1, 1);
    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int j=start; j<end; j++) local += shm->array[j];
            sem_wait(&shm->mutex);
            shm->sum += local;
            sem_post(&shm->mutex);
            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum[NPROCS];
    sem_t mutex;
} shm_t;
```

- Recall, we also parallelized the array sum program using normal IPC mechanisms
  - Inter-process communication using shared memory and semaphores

Same Program but without Semaphore

```
int main() {
    shm_t* shm = setup();

    int chunks = SIZE/NPROCS;
    for(int i=0; i<NPROCS; i++) {
        if(fork()==0) {
            int local=0;
            int start = i*chunks;
            int end = start+chunk;
            for(int j=start; j<end; j++) local += shm->array[j];

            shm->sum[i] = local;

            cleanup_and_exit();
        }
    }
    for(int i=0; i<NPROCS; i++) wait(NULL);
    int total=0; for(int j=0; j<NPROCS; j++) total += sum[j];
    cleanup();
    return 0;
}
```

```
typedef struct shm_t {
    int A[SIZE];
    int sum[NPROCS];

} shm_t;
```

- It is another version of the same program that doesn’t use semaphores
  - Removing it just to simplify the motivation for today’s topic

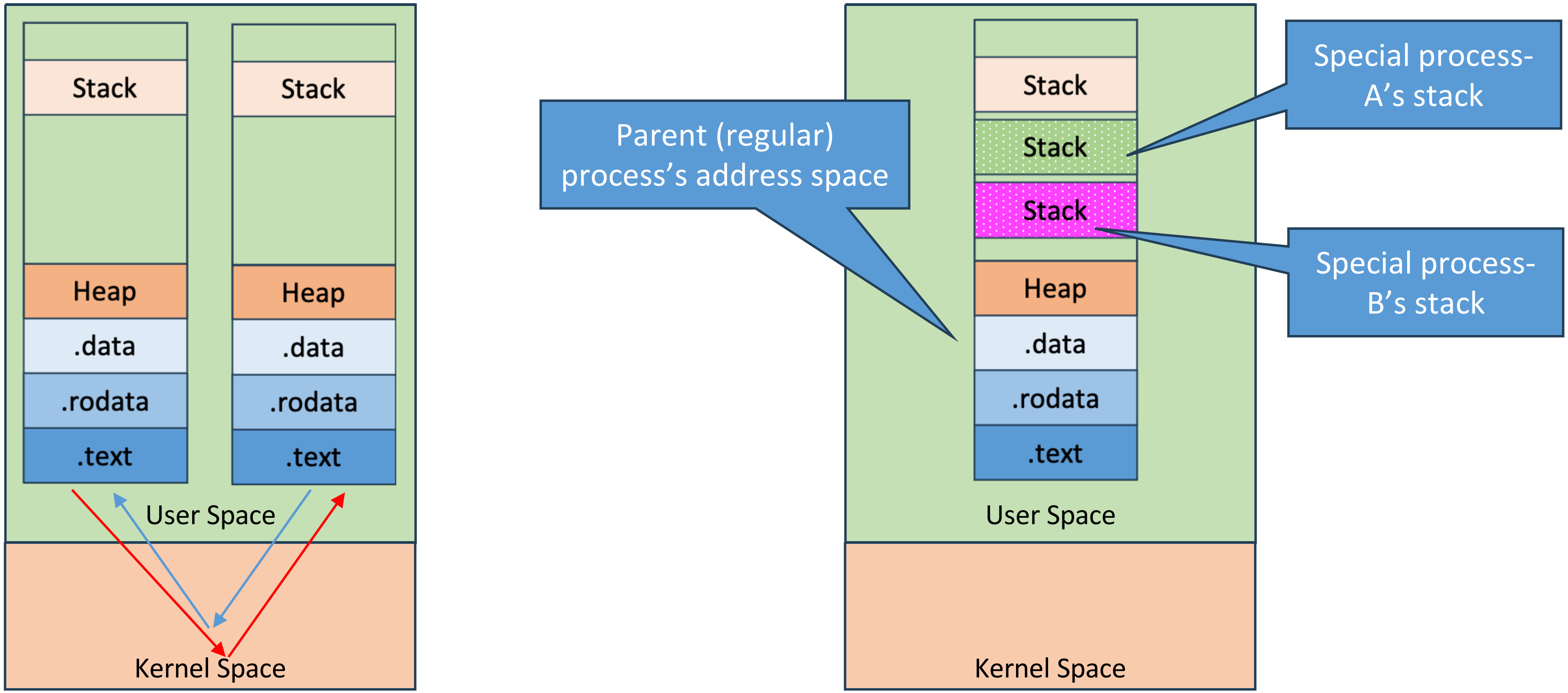
IPC Using SHM Within a Multicore Processor

- Inter-process communication in shared memory (only)
  - Transfer of control from user space to kernel space and vice-versa
- Complicated IPC mechanism for communication
- OS has to reserve extra memory / resources
  - Same copy of .text segment in each process
- Separate page table for each process
- ~~Cost of IPC may exceed the cost of actual computation!~~
  - We can neglect it as the processes communicate in shared memory without using any explicit IPC calls after the setup of SHM

How to Fix the Issues we Highlighted?

- Asking the OS to allow creation of "**special**" processes that has **special** powers
- These **special** processes can be created by the parent process, and they all live in harmony like an **ideal** Indian joint family, where entire resources in the house are shared within the family members
    - Sharing of page table of the parent process
    - Sharing of parent's process address space
        - Shared heap, .text, .data segment, etc.
            - Stack cannot be shared as we need each of these "**special**" processes to execute a different method call chain inside the same program
            - Likewise, PC, registers also cannot be shared. Hence, they go through the same set of steps during context switch similar to **regular** processes
    - They can communicate with each other without using any special APIs or without going into the kernel space (zero overheads in communication!)

2 Regular Process v/s 2 Special Process



These “**special**” processes are called as **Threads**!

Thread Creation in Linux

```
//Asynchronously invoke func in a new thread
int pthread_create(
              //returned identifier for the new thread
              pthread_t *thread,

              //specifies the size of thread's stack and
              //how the thread should be scheduled by OS
              const pthread_attr_t *attr,

              //routine executed after creation
              void *(*func)(void *),

              //a single argument passed to func
              void *arg
) //returns error status
```

Waiting for Thread Termination in Linux

```
//Suspend execution of calling thread until thread
//terminates
int pthread_join(
  //identifier of thread to wait for
              pthread_t thread,

  //terminating thread's status (NULL to ignore)
              void **status
) //returns error status
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
      args[i].low=i*chunk; args[i].high=(i+1)*chunk;
      pthread_create(&tid[i],
                     NULL,
                     thread_func,
                     (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
      pthread_join(tid[i] , NULL);
      result += args[i].sum;
    }
  }
  printf(”Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Original code

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf(”Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Structure for thread arguments

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                        NULL,
                        thread_func,
                        (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Function called when thread is created

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

No point in creating thread if there isn't enough to do

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Marshal input argument to thread

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Create threads to execute `array_sum`

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

Main program blocks until threads terminate

Array Sum using Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE]; // Initialization code elided
int array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
        args[i].low=i*chunk; args[i].high=(i+1)*chunk;
        pthread_create(&tid[i],
                       NULL,
                       thread_func,
                       (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
        pthread_join(tid[i] , NULL);
        result += args[i].sum;
    }
  }
  printf(”Total Sum is %d\n", result);
  return 0;
}
```

Add the results together to produce the final output

Advantages of Multithreading

- Responsiveness
  - Even if part of program is blocked or performing lengthy operation, multithreading allows the program to continue
- Economical resource sharing
  - Threads share memory and resources of their parent process which allows multiple tasks to be performed simultaneously inside the process
- Utilization of multicores
  - Easily scale on modern multicore processors

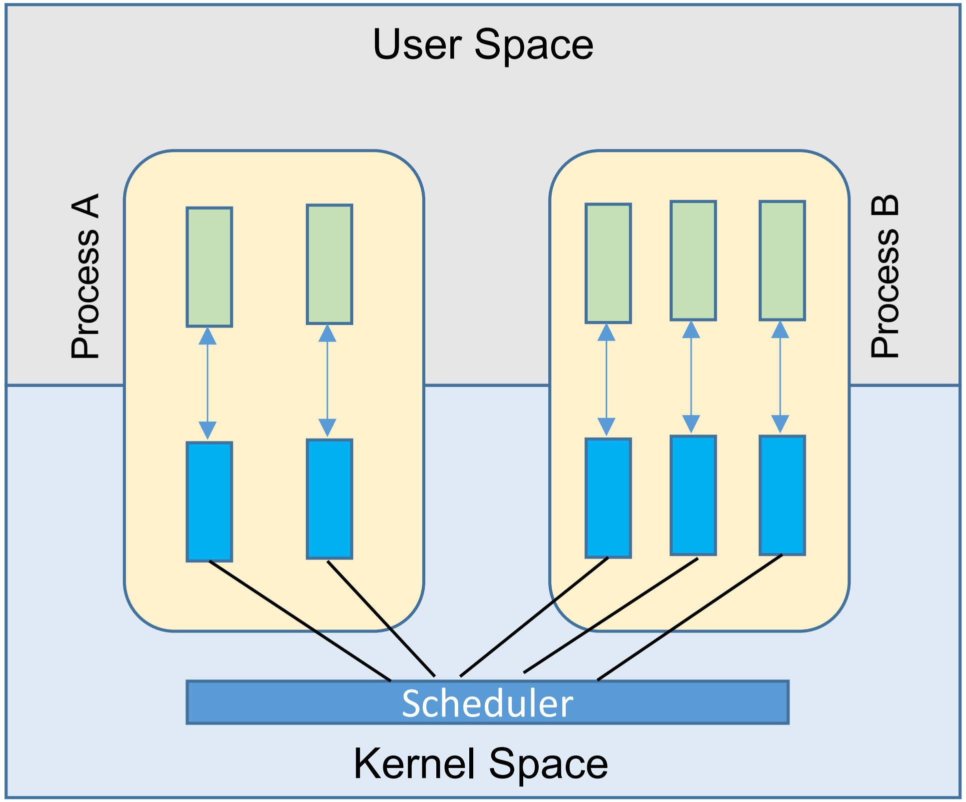
Thread Scheduling

- OS schedules threads that are ready to run independently, much like processes
- The context of a thread (PC, registers) is saved into/restored from thread control block (TCB)
  - Every PCB has one or more linked TCBs

Types of Threads

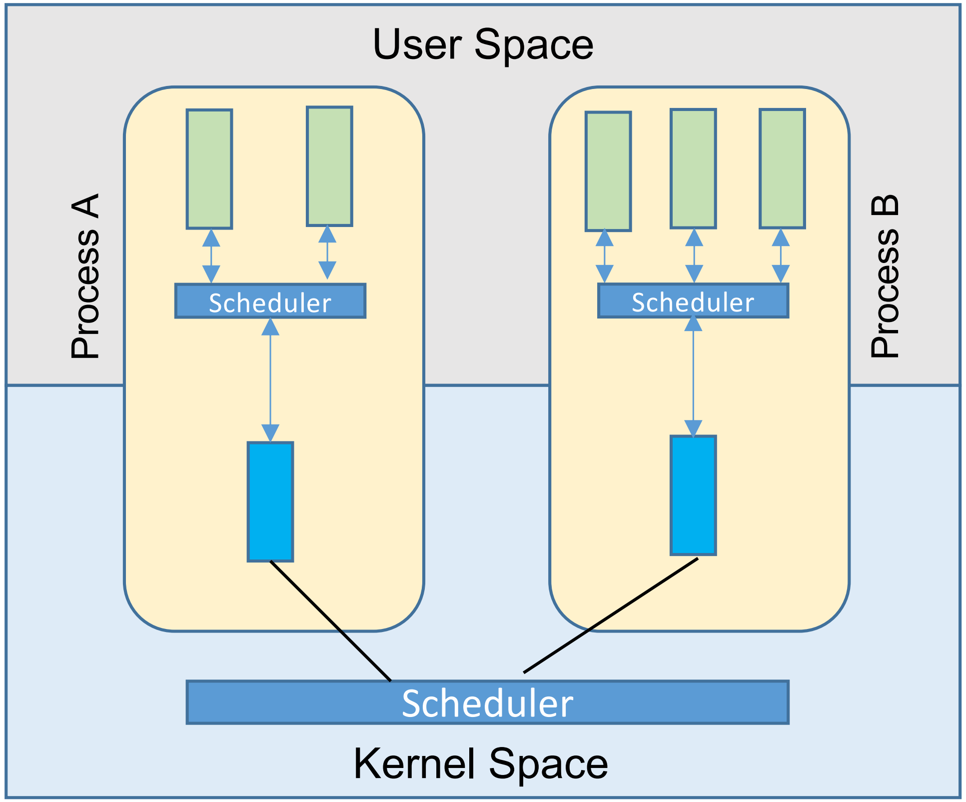
- **1x1** threading Model (Kernel Level Threads)
- **MxN** threading model (User Level Threads)

1x1 Threading Model



- Every thread created by the user has 1x1 mapping with the kernel thread
  - E.g., pthread library on Linux
- OS manages scheduling policy
- Switching between threads of same process much cheaper than switching between two processes
  - No need to switch address space (page table)
  - However, creating large number of threads would add to overheads
    - Cycles lost in thread creation
    - Frequent context switches

MxN Threading Model



- User gets to create several threads, but each of these threads can be mapped to a single kernel level thread
  - E.g., fibers in boost C++ library
- Runtime library (in user space) manages all thread operations (including scheduling policy)
  - Lightweight operations (OS is totally unaware of user level thread operations)
    - Cost of thread creation is low
    - Infrequent context switches due to lesser number of kernel threads than user threads
  - Covered in depth in CSE513 (Parallel Runtimes for Modern Processors)

Measures of parallel performance

- Speedup = $T_{serial}/T_{parallel}$
- Parallel efficiency = $T_{serial}/(pT_{parallel})$

Fig. source: http://www.drdobbs.com/cpp/going-superlinear/206100542

Next Lecture

- Race condition in multithreading
- Assignment-5 will be released today with a deadline of one week (**No extensions**!)