Multi Level Feedback Queue (MLFQ) Scheduling

- optimize turnaround time
- minimize response time - system feel responsive to interactive users

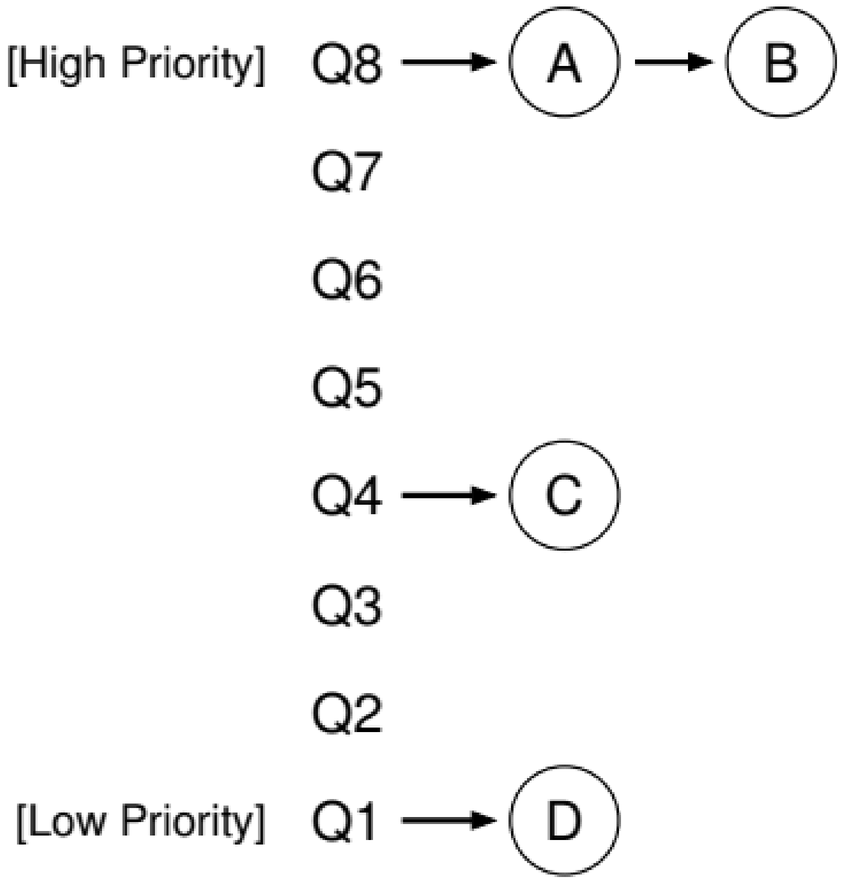
- number of distinct queues each assigned a different priority level
- a job that is ready to run is on a single queue

first two basic rules for MLFQ:

Rule 1: If Priority(A) > Priority(B), A runs (B doesn’t)

Rule 2: If Priority(A) = Priority(B), A & B run in RR.

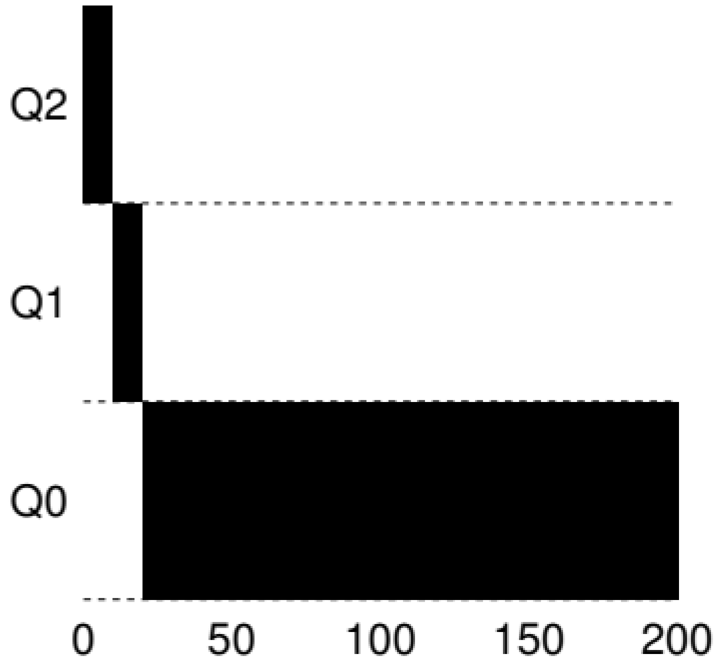
- MLFQ varies the priority of a job based on its observed behavior
  - If a job repeatedly relinquishes the CPU while waiting for input from the keyboard, **MLFQ will keep its priority high**, as this is how an interactive process might behave
  - If, instead, a job uses the CPU intensively for long periods of time, **MLFQ will reduce its priority**



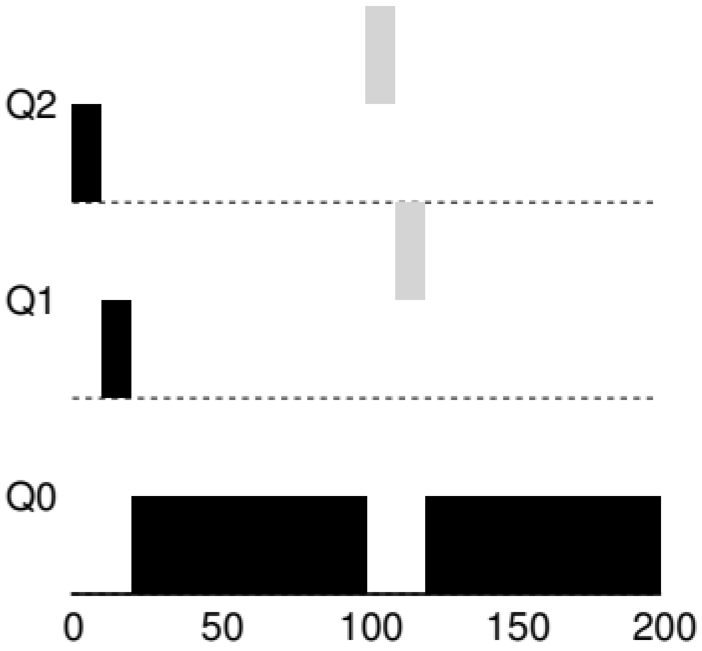
Changing Priority

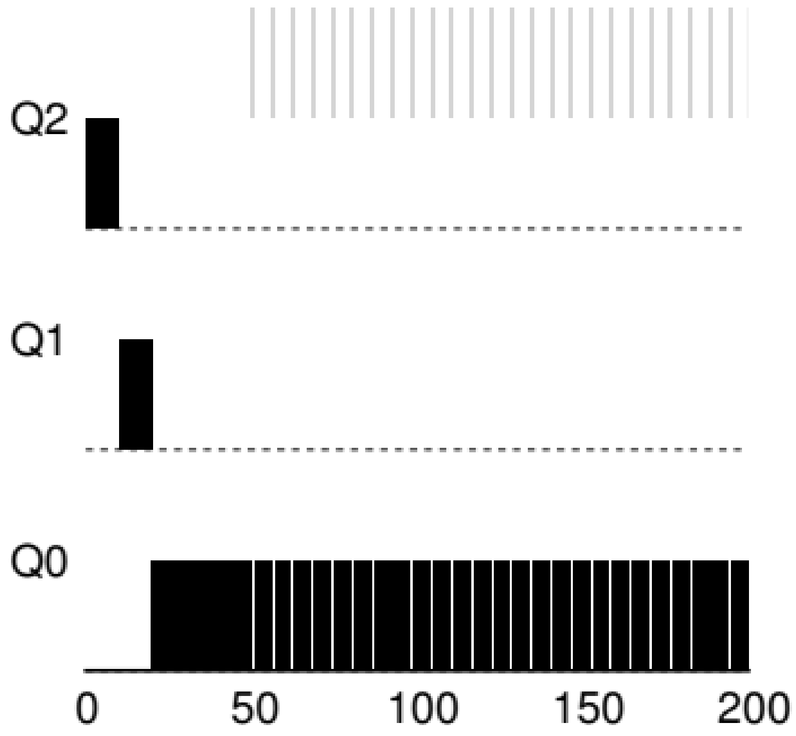
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue)
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

Long Process A (CPU intensive)



Short running interactive Process B

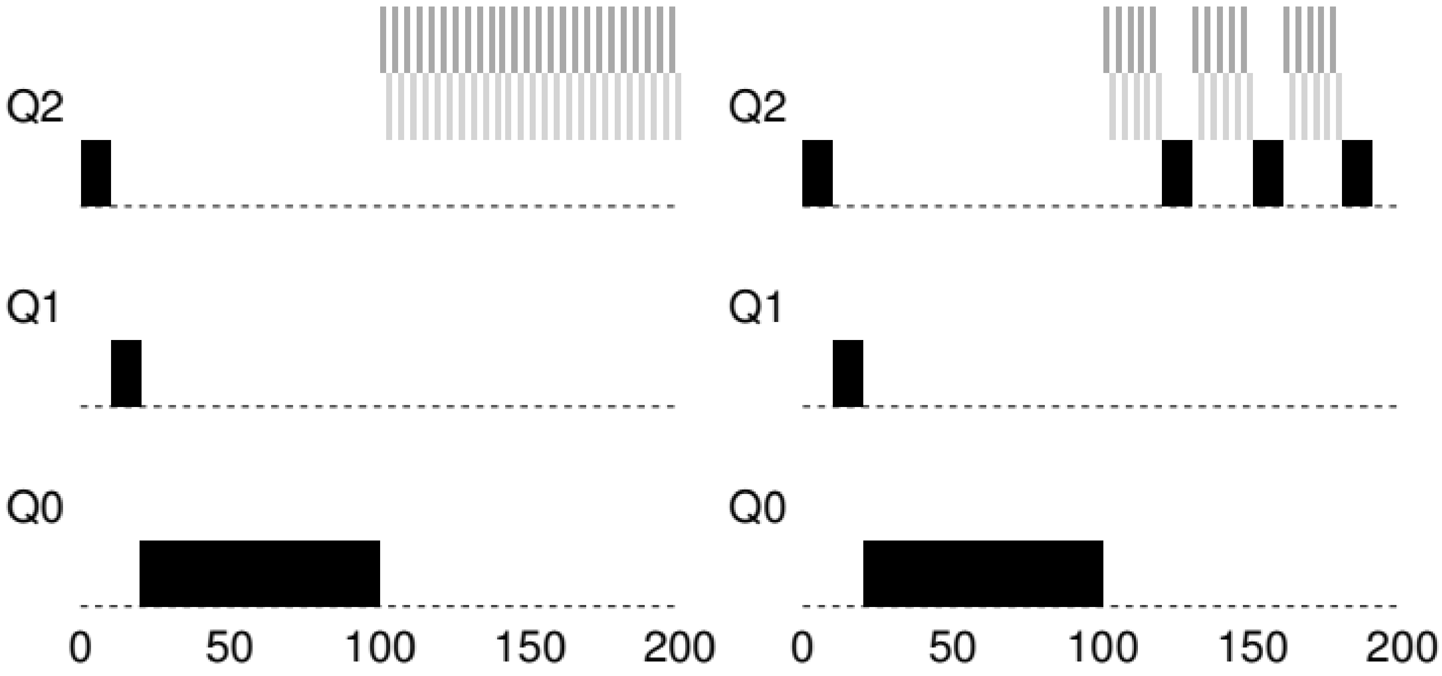




- Mixture of I/O-intensive and CPU-intensive Workload
- Interactive Process B (gray) that uses CPU for only 1 ms

Priority Boost

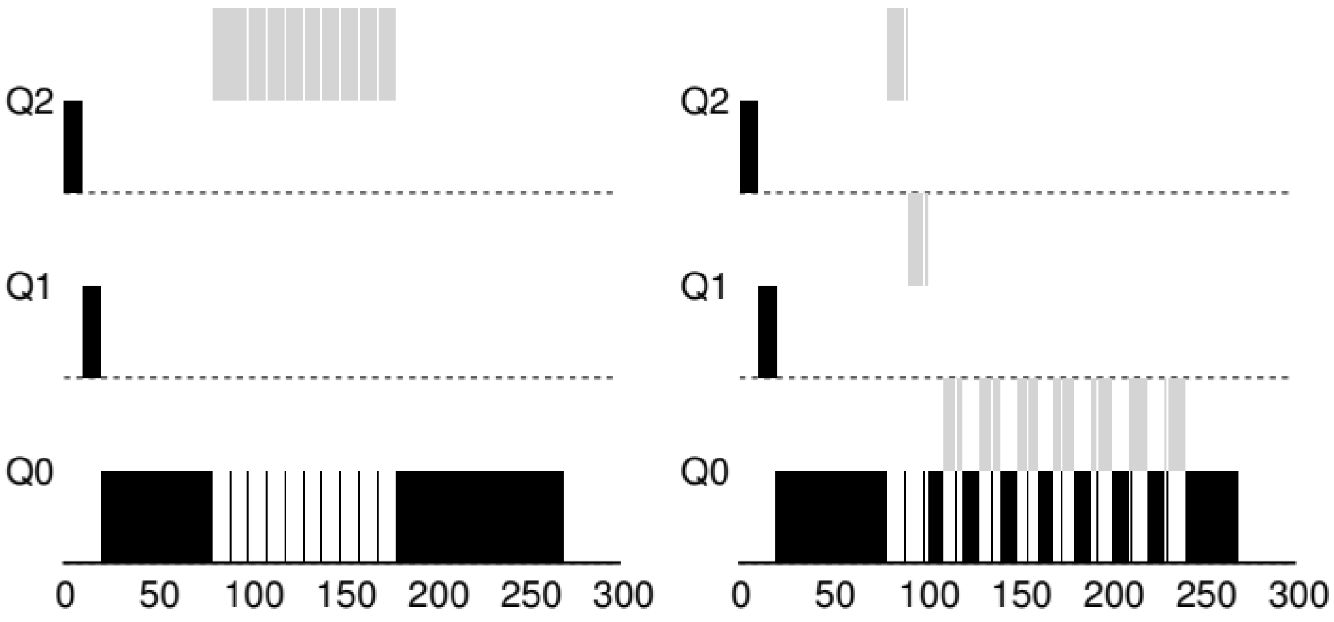
- Move all the jobs to the topmost queue after time period.
- Advantages:
  - processes are guaranteed not to starve



- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

What is the value for S?

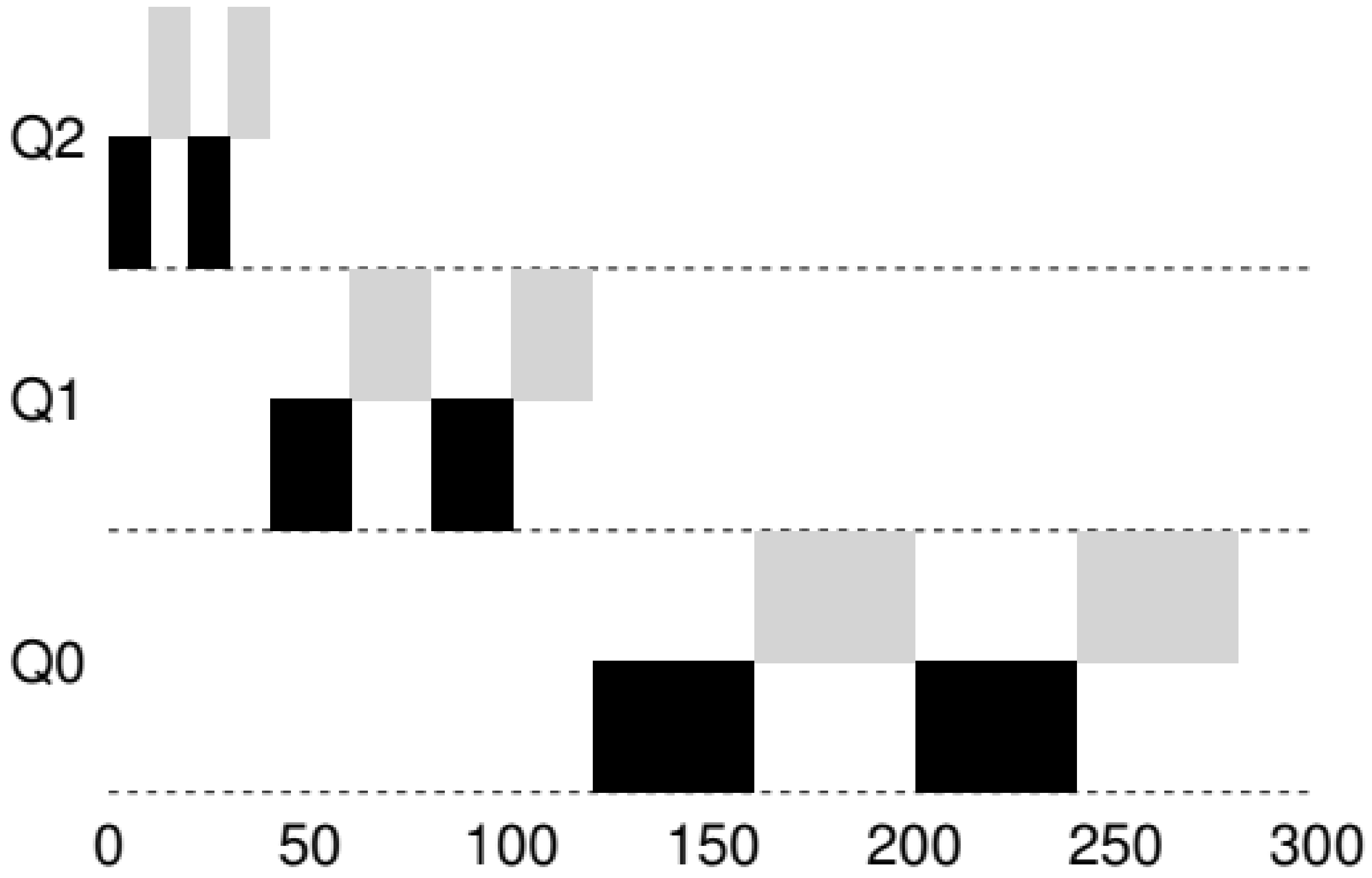
Gaming the scheduler



- generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource
- before the time slice is over, issue an I/O operation (to some file you don't care about) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time
- thereby, a job could nearly monopolize the CPU

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)

Lower Priority, Longer Quanta



Summary - MLFQ

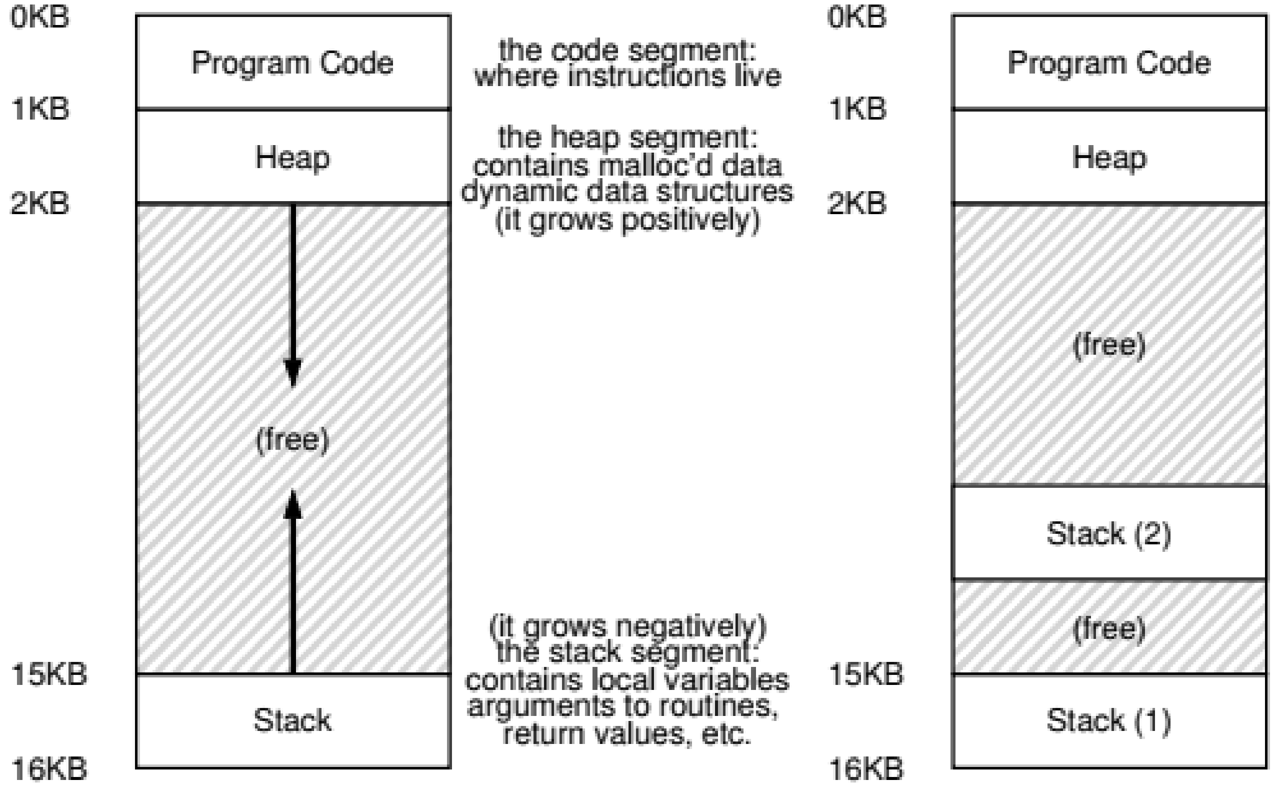
- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

Threads

- Concept of Process
  - **Resource ownership**: A process includes a virtual address space to hold the process image, i.e., the collection of program, data, stack, and attributes defined in the PCB. Resources are main memory, Disk I/O, I/O devices, and files
  - **Scheduling/execution**: The execution of a process follows an execution path (trace) through one or more programs. A process has an execution state (Running, Ready, etc.) and a dispatching priority
- Threading
  - Ability of an OS to support multiple, concurrent paths of execution within a single process.
  - Lightweight process
  - Achieves parallelism

Threads … Contd.

- Single point of execution within a program
- Share same address space and can access same data
- Context switching: between threads; Adv.: address space remains the same, page table too
- Supports parallelism



Single-Threaded and Multi-Threaded Address Spaces

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

- Main program creates two threads
- pthread_create() : Creates thread
- Pthread_join(): waits for a particular thread to complete

Thread trace (1)

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Thread trace (2)

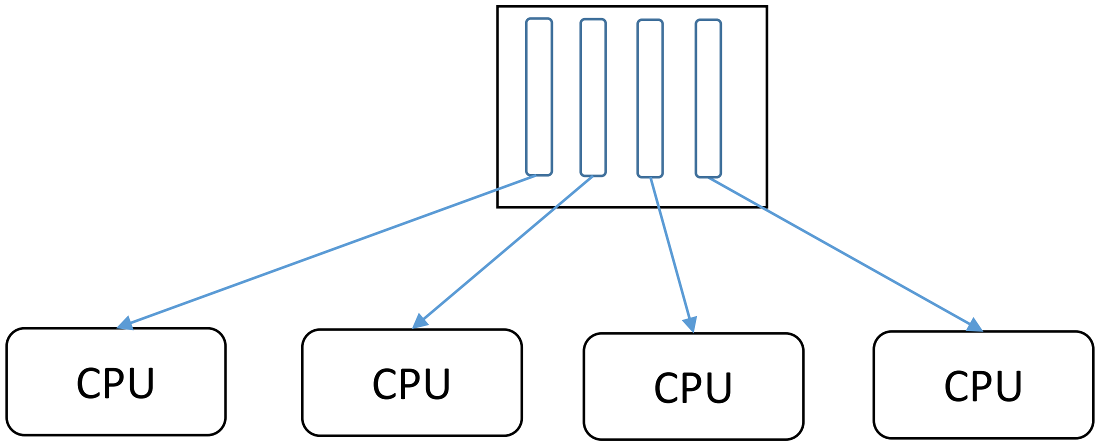
main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Thread trace (3)

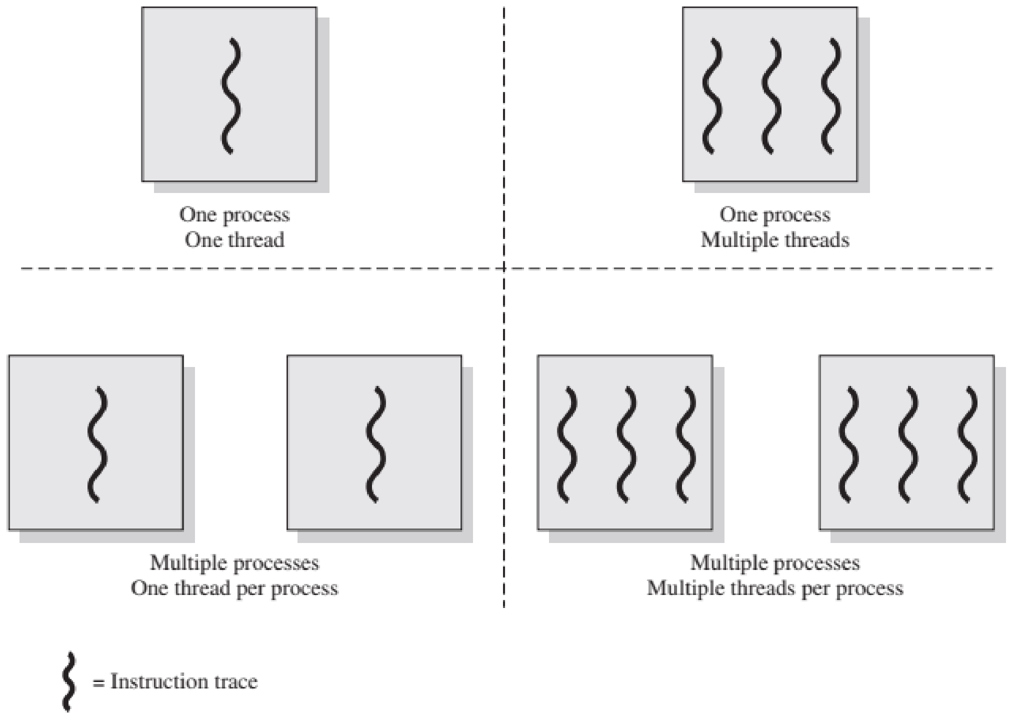
main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

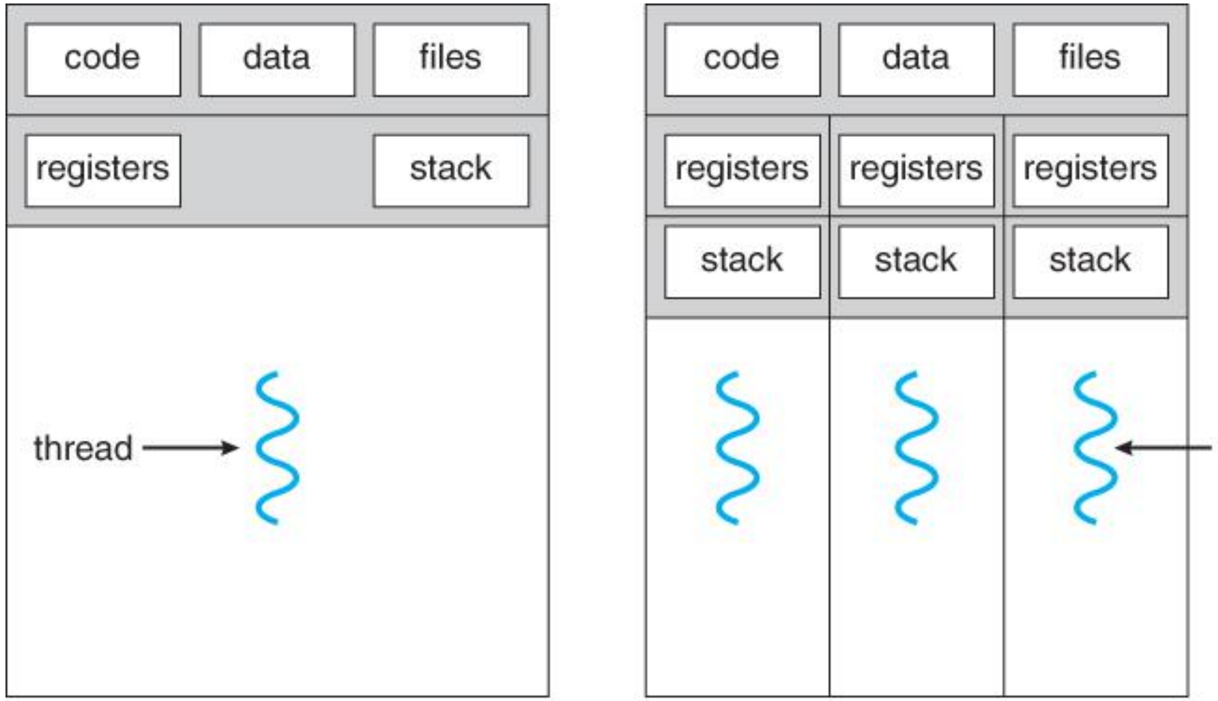
Threads

- Four threads created
- Each thread is independent
- Management of threads is simpler than processes
- Shared instructions, global, and heap regions
- Each thread has its own stack



- Process:
  - A virtual address space that holds the process image
  - Protected access to processors, other processes, files, and I/O resources
- Threads
  - A thread execution state (Running, Ready, etc.)
  - A saved thread context when not running; one way to view a thread is as a independent program counter operating within a process
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process, shared with all other threads in that process





single-threaded process

multithreaded process

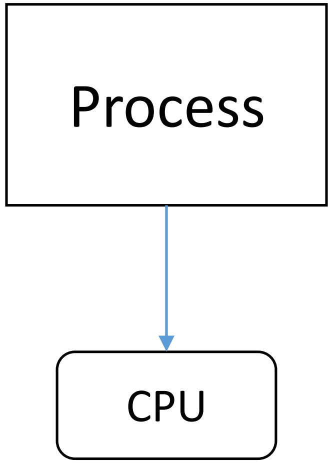
Per process	Per thread
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Signals and signal handlers	
Accounting info	

POSIX threads – IEEE 1003.1c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS 10
void *print_hello_world(void *tid)
{
        /* This function prints the thread’s identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[]) {
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NTHREADS];
    int status, i;
    for(i=0; i < NTHREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
    status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
    if (status != 0) {
        printf("pthread returned error code %d\n", status);
        exit(-1);
    }
    }
    exit(NULL);
}
```

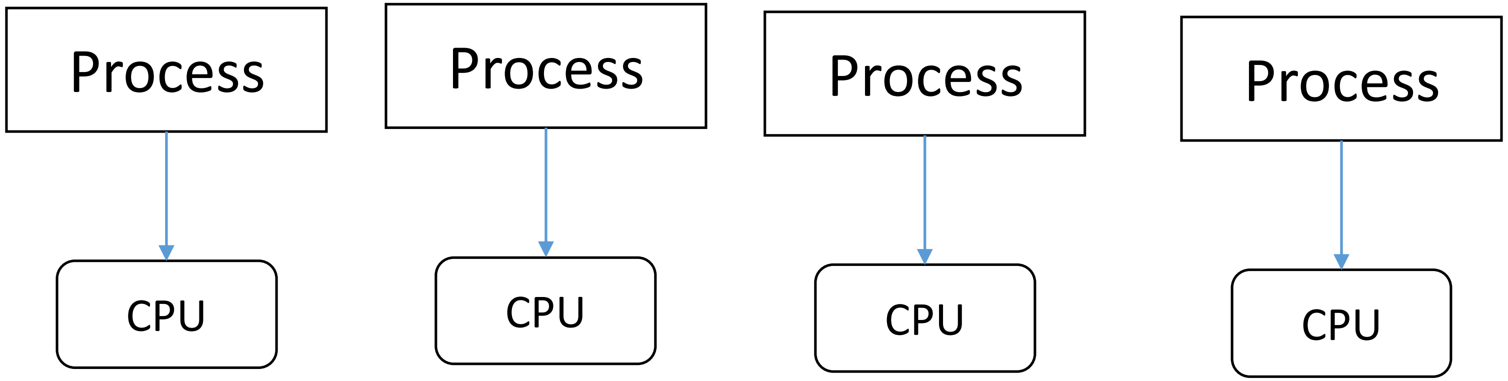
Sum of first 1,00,00,000 numbers



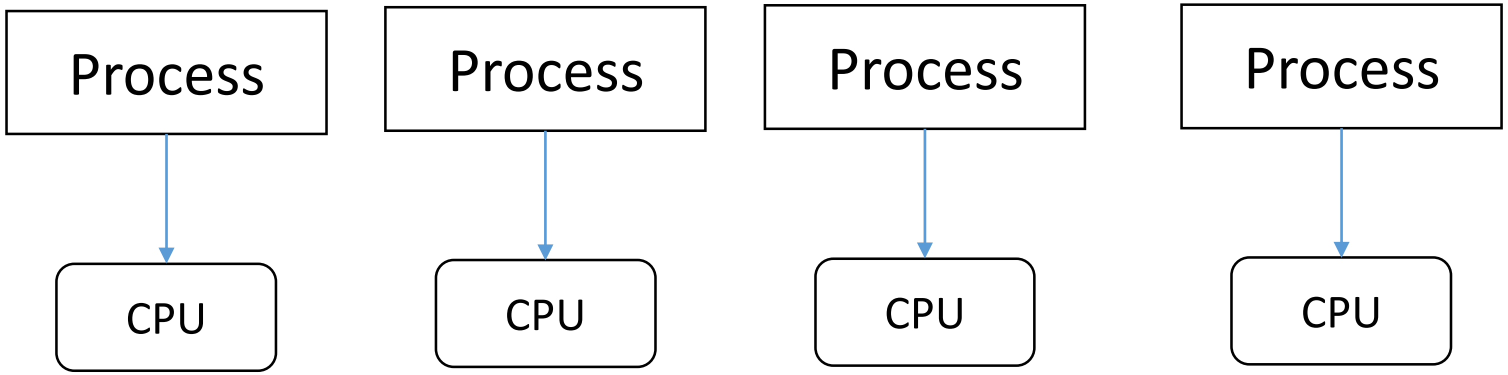

```
#include<stdio.h>
long add() {
        int i=0;
        long sum=0;

        while(i < 10000000) {
         sum = sum+=i;
         i++
        }
        return sum;
}
```

```
int main() {
        long sum;
        sum = add();
        printf("%l", sum);
}
```

Is it possible to speed up the operation? How?
Assume multiple processors/CPU

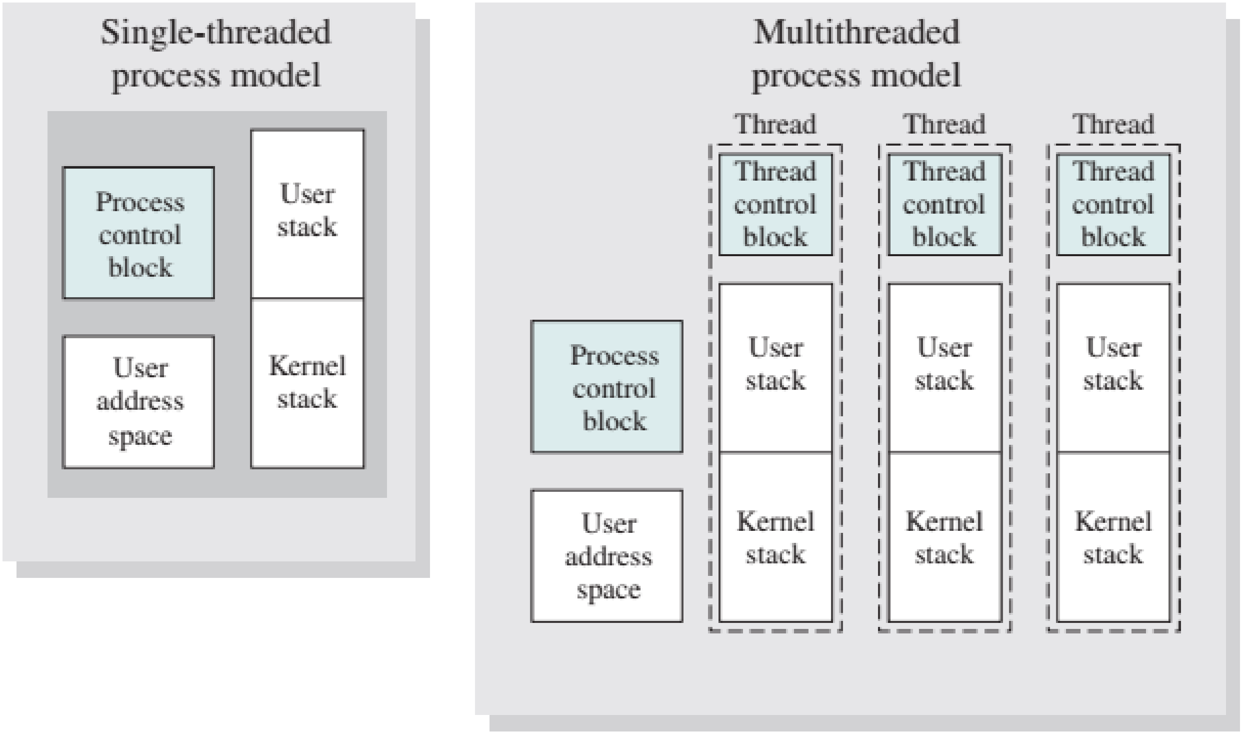


- Four fork() system calls; one for each process
- Each process executes independently
- IPC mechanism to communicate between processes
- Each process has its own instruction, data, heap, and stack



Benefits of Threads

- Far less time to create a new thread in an existing process, than to create a brand-new process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process than to switch between processes
- Threads enhance efficiency in communication between different executing programs



Threads vs Processes

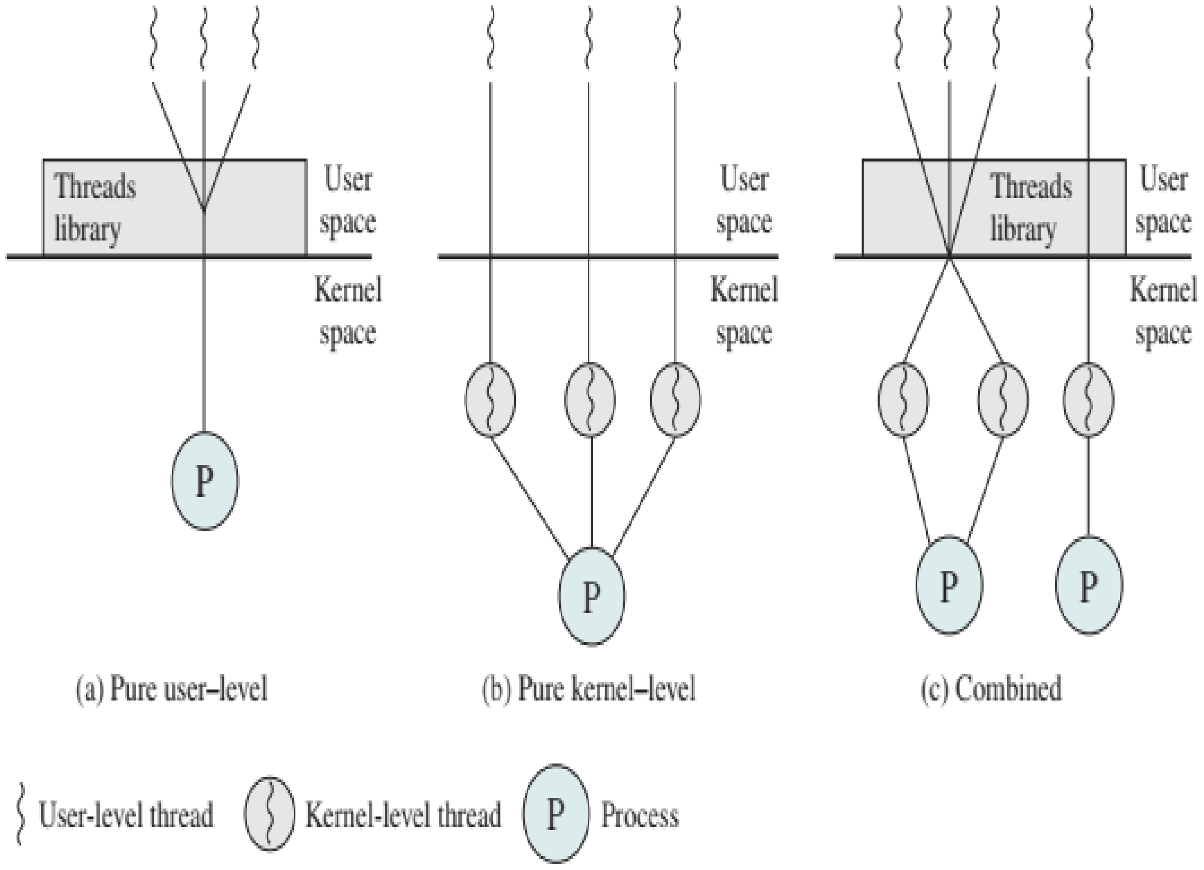
- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main() & has the process's stack
- Inexpensive creation
- Inexpensive context switching
- Efficient communication
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- A process has at least one thread
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- Interprocess communication can be expressive
- If a process dies, its resources are reclaimed & all threads die

Source: http://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l08-thread.pdf

Types of Threads

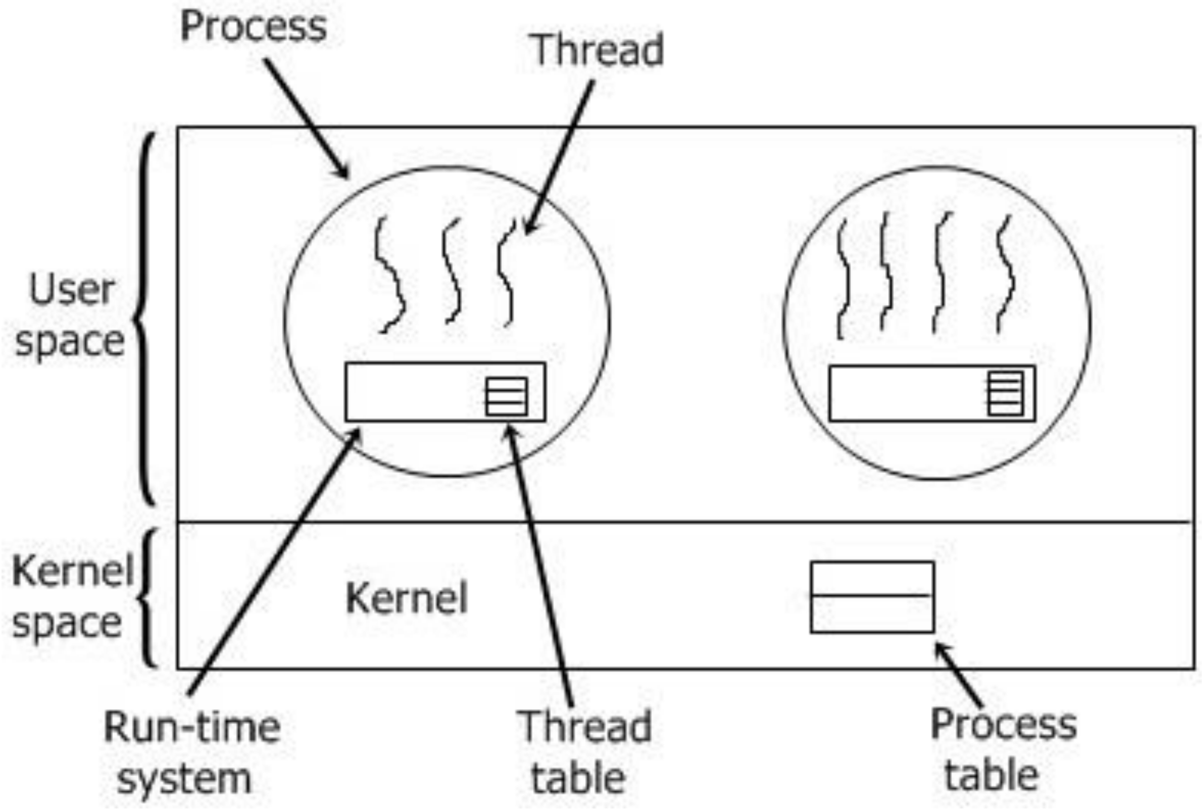
- User-Level Threads
  - thread management is done by a user level thread library
  - the kernel does not know anything about the threads running
- Kernel-Level Threads
  - threads are directly supported by the kernels
  - also known as light weight processes



User Level Threads

- Fast as no system call to manage. Thread library does everything
- Switching is fast. NO switch from user to protected mode

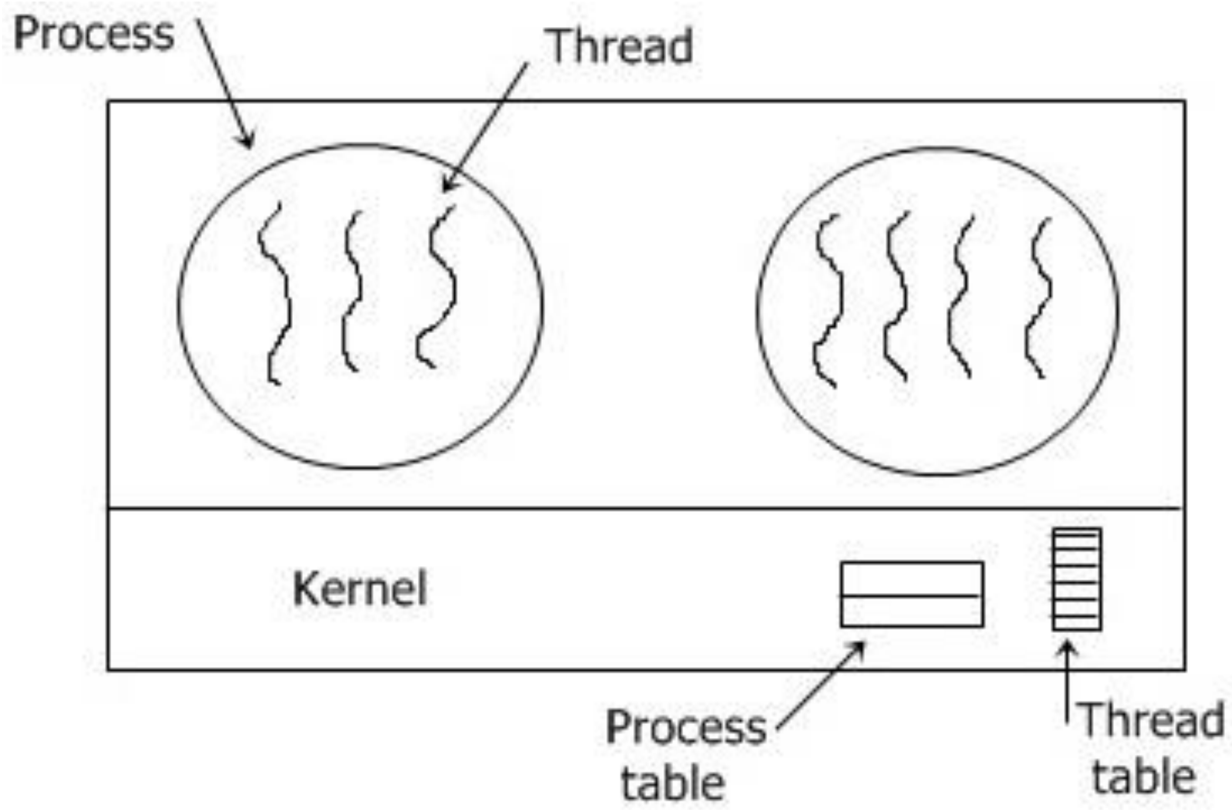
- Scheduling can be an issue
- Lack of coordination between kernel and threads
- If one thread invokes a system call, all threads need to wait



Kernel Level Threads

- Scheduler can decide to give more time to a process that large number of threads
- Since threads managed by kernel, no blocking on system calls

- Slow in comparison
- Overheads – scheduling threads apart from processes



References

- William Stallings, "Operating Systems: Internals and Design Principles", 9th edition, Pearson Edu. Ltd., 2018
- Charles Crowley, "Operating Systems: A design-oriented approach", TMH
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison), "Operating Systems: Three Easy Pieces".

  URL: http://pages.cs.wisc.edu/~remzi/OSTEP/

Process Synchronization
Deadlocks
Inter Process Communication (IPC)

CS3003D: Operating Systems

Concurrency

- Concurrency encompasses a host of design issues, including
  - communication among processes
  - sharing of and competing for resources (such as memory, files, and I/O access)
  - synchronization of the activities of multiple processes, and allocation of processor time to processes
- Concurrency arises in three different contexts
  - Multiple applications
  - Structured applications
  - Operating system structure

Synchronization

shared variable
int flag = 5

ProgramA

flag++

1) reg1 = flag
2) reg1 = reg1 + 1
3) flag = reg1

Output value of flag can be 5, 4, or 6 based on the way the processes are executing, when context switching happens

ProgramB

flag --

4) reg2 = flag
5) reg2 = reg2 - 1
6) flag = reg2

Scenario1

ProcessA
1)
2)
3) flag = 6
Context Switch
ProcessB
4)
5)
6) flag = 5

Scenario2

ProcessB
1)
2)
3) flag = 4
Context Switch
ProcessA
4)
5)
6) flag = 5

Scenario3

ProcessA
1) reg1=5
Context Switch
ProcessB
2) reg2 = 5
3) reg2 = 4
4) flag = 4
Context Switch
ProcessA
5) reg1 = 6
6) flag = 6

Scenario4

ProcessB
1) reg2=5
Context Switch
ProcessA
2) reg2 = 5
3) reg2 = 6
4) flag = 6
Context Switch
ProcessB
5) reg1 = 4
6) flag = 4

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
            counter);
    return 0;
}
```

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Source: Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison), "Operating Systems: Three Easy Pieces".

Race condition

- Many processes manipulate the same data portion
- During concurrent execution, outcome depends upon the order in which the access happens
- Incorrect data leads to misleading output
- Can be prevented by synchronization between processes

How to avoid race condition?

- Prohibit more than one process from reading and writing the shared data (critical section) at the same time

Definitions

- Critical Section
  - A section of code within a process that requires access to shared resources, and that must not be executed while another process is in a corresponding section of code
- Race condition
  - A situation in which multiple threads or processes read and write a shared data item, and the final result depends on the relative timing of their execution
- Mutual exclusion
  - The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources

Operating System concerns

- OS must able to keep tract of the various processes
  - PCB
- OS must allocate and deallocate resources for each active processes
  - Processor timer, Memory, Files, I/O devices
- Protection of data and resources of each process against unintended interference by other processes

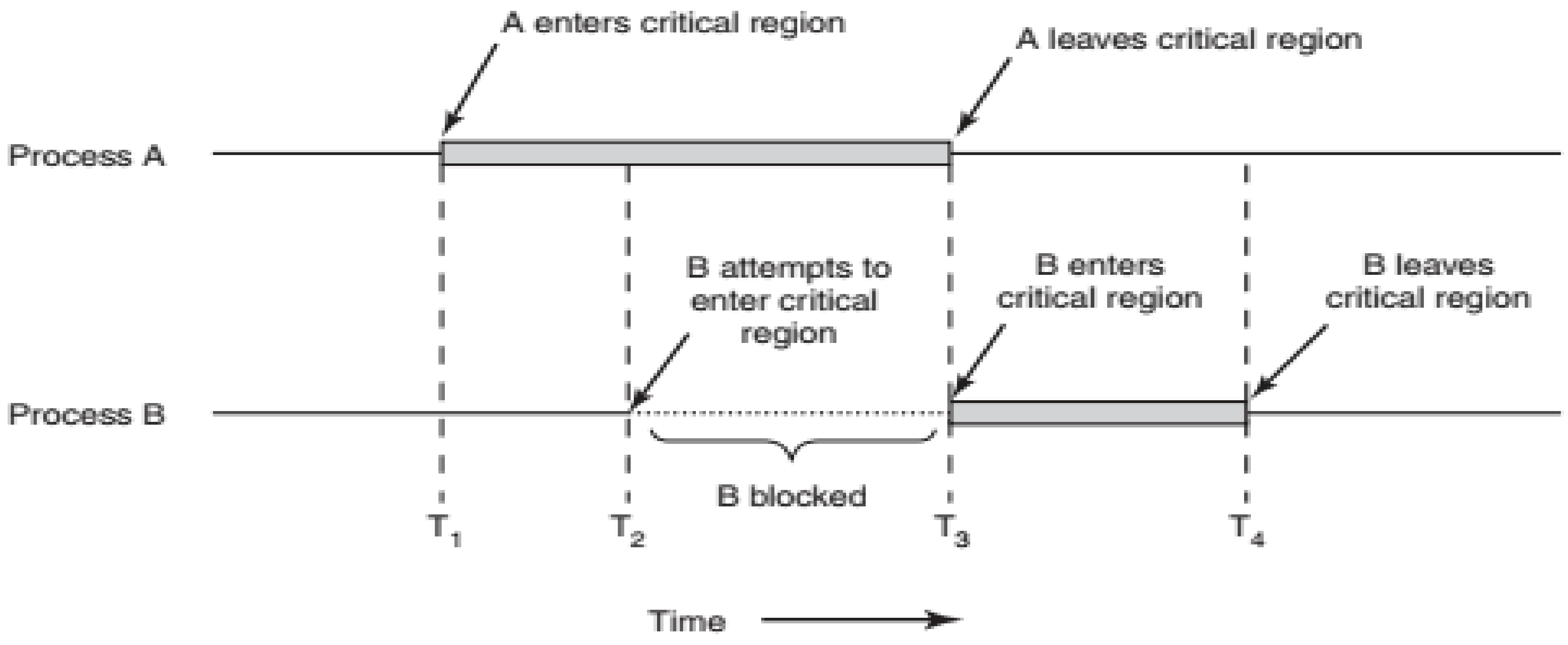
Process Interaction

(Competition for resources and Cooperation among processes)

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Three requirements for critical section problem

- Mutual Exclusion
  - No two processes may be simultaneously inside their critical regions
- Progress
  - No process running outside its critical region may block any process
- No starvation (bounded waiting)
  - No process should have to wait forever to enter its critical region



Solutions to critical section

- Disable interrupts
  - Context switches will not happen
  - Codes that execute in the kernel can only disable interrupts
  - User processes/application programs cannot disable interrupts

```
While(TRUE) {
   // code area
   disable_ interrupts()    < LOCK
   critical_section
   enable_interrupts()     < UNLOCK
   // other code area
}
```

Busy waiting

Shared
int turn = 1;

Process-1

```
while (TRUE) {
  while (turn == 2);      // LOCK
  critical_section
  turn = 2;               // UNLOCK
  reminder_code_here
}
```

Process-2

```
while (TRUE) {
  while (turn == 1);      // LOCK
  critical_section
  turn = 1;               // UNLOCK
   reminder_code _here
}
```

- Mutual exclusion achieved
- Busy waiting – resource wastage
  - When Process-2 executes first, always in loop; always in primary memory – either at READY state or at RUNNING state
- Progress condition is violated

No Mutual Exclusion

Shared
p1_inside = false, p2_inside = false

Process-1

```
while (TRUE) {
  while (p2_inside == TRUE);       // LOCK
  p1_inside = TRUE;
  critical_section
  p1_inside = FALSE;               // UNLOCK
}
```

Process-2

```
while (TRUE) {
  while (p1_inside == TRUE);       // LOCK
  p2_inside = TRUE;
  critical_section
  p2_inside = FALSE;               // UNLOCK
}
```

```
while(p2_inside == TRUE);
// Context Switch (Process-2)
while(p1_inside == TRUE);
p2_inside = TRUE;
// Context Switch (Process-1)
p1_inside = TRUE;
```

- Mutual exclusion is not guaranteed
- Both processes can enter into critical section

Endless Wait

Shared
p1_inside = false, p2_inside = false

Process-1

```
while (TRUE) {
  p1_inside = TRUE;
  while (p2_inside == TRUE);      // LOCK
  critical_section
  p1_inside = FALSE;              // UNLOCK
}
```

Process-2

```
while (TRUE) {
  p2_inside = TRUE;
  while (p1_inside == TRUE);      // LOCK
  critical_section
  p2_inside = FALSE;              // UNLOCK
}
```

```
p1_inside = TRUE
// Context Switch (Process-2)
p2_inside = TRUE;
```

- Achieves Mutual exclusion
- Can it progress?
  - DEADLOCK!

Peterson's Solution

Shared
p1_inside, p2_inside, favoured

Process-1

```
while (TRUE) {
  p1_inside = TRUE;
  favoured = 2;
  while (p2_inside == TRUE AND favoured =2);  // LOCK
  critical_section
  p1_inside = FALSE;                // UNLOCK
}
```

Process-2

```
while (TRUE) {
  p2_inside = TRUE;
  favoured = 1;
  while (p1_inside == TRUE AND favoured = 1);    //
LOCK
  critical_section
  p2_inside = FALSE;                // UNLOCK
}
```

- Breaking the deadlock as one of the processes is favoured
- Solves critical section problem for two processes.

Bakery Algorithm – synchronization between N processes (N > 2)

- Proposed by Leslie Lamport
- Similar to token system in the bakeries/banks
  - Customers upon entering the bank is issued with the token
  - Waits until his/her turn arrives
  - Dispense the token and the service is rendered

Ref.: http://www.cs.umd.edu/~shankar/412-S99/note-7.html

Simplified version of Bakery algorithm

- Each process is numbered 0 to N-1
- Each process i has an integer variable num[i], initially 0, that is readable by all processes but writeable by process i only

```
Entry(i) {
  num[i] = MAX( num[0], num[1], ... , num[N-1] ) + 1 ;
  for p = 0 to N-1 do  {
      while (num[p] != 0 AND num[p] < num[i])   do no-op ;
  }
}
```

Lock

Critical section

```
Exit(i) {
  num[i] = 0 ;
}
```

Unlock

Example

num[i]	P0	P1	P2	P3	P4
Initial	0	0	0	0	0
P2	0	0	1	0	0
P3	0	0	0	2	0
P4	0	0	0	0	3
P0	4	0	0	0	0
P1	4	5	0	0	0
Final	4	5	1	2	3

num[i] = MAX(num[0], num[1], ... , num[N-1]) + 1

num[i]	P0	P1	P2	P3	P4
Initial	4	5	1	2	3
P2	4	5	0	2	3
P3	4	5	0	0	3
P4	4	5	0	0	0
P0	0	5	0	0	0
P1	0	0	0	0	0
Final	0	0	0	0	0

```
for p = 0 to N-1 do  {
        while(num[p] != 0 AND num[p] < num[i]) do no-op ;
  }
```

Problem!

Assumption: No two processes get the same token

When two process gets the same num[i] value (same token)

Two processes enter into the critical section