Recap

Replacement Policies (for Physical Memory or TLB Cache)

- Optimal
- FIFO
- Random
- LRU

Clock Algorithm

LRU performs the closest in optimal for typical workloads, but it is really costly on every memory access

Clock Algorithm - is a way to approximate LRU cheaply

Add extra **use bit** to page table entry, on every memory access MMU set use bit of page to 1

When looking for page to evict

- visit pages in round-robin order
- if page use bit is 0, choose that page to evict
- else set use bit to 0 and continue search

Clock Algorithm: Example

Frames after accessing: 0 1 2 0 1

Frame #	0	1	2
Page	0	1	2
Use bit	1	1	1

No eviction; all pages are marked as used recently

To access page: 3

Frame #	0	1	2
Page	0 ->3	1	2
Use bit	1->0->1	1->0	1->0

Page 0 (not exactly the LRU) is evicted; then page 3 (new) is marked as used recently

To access pages: 1 3

Frame #	0	1	2
Page	3	1	2
Use bit	1	0->1	0

No eviction; pages 1 and 3 are marked as used recently

To access pages: 0

Frame #	0	1	2
Page	3	1	2->0
Use bit	1	1->0	0->1

Page 2 (the LRU) is evicted; page 0 (new) is marked as used recently

Dirty Bit Optimization

If a page has only been read from (never written to) there is no reason to write it back to the swap space (if the swap page is large enough to contain a copy of it already) when it is evicted

Add a **dirty bit** to the page table, initialize to 0, on every write to the page table the MMU (hardware) sets the bit to 1

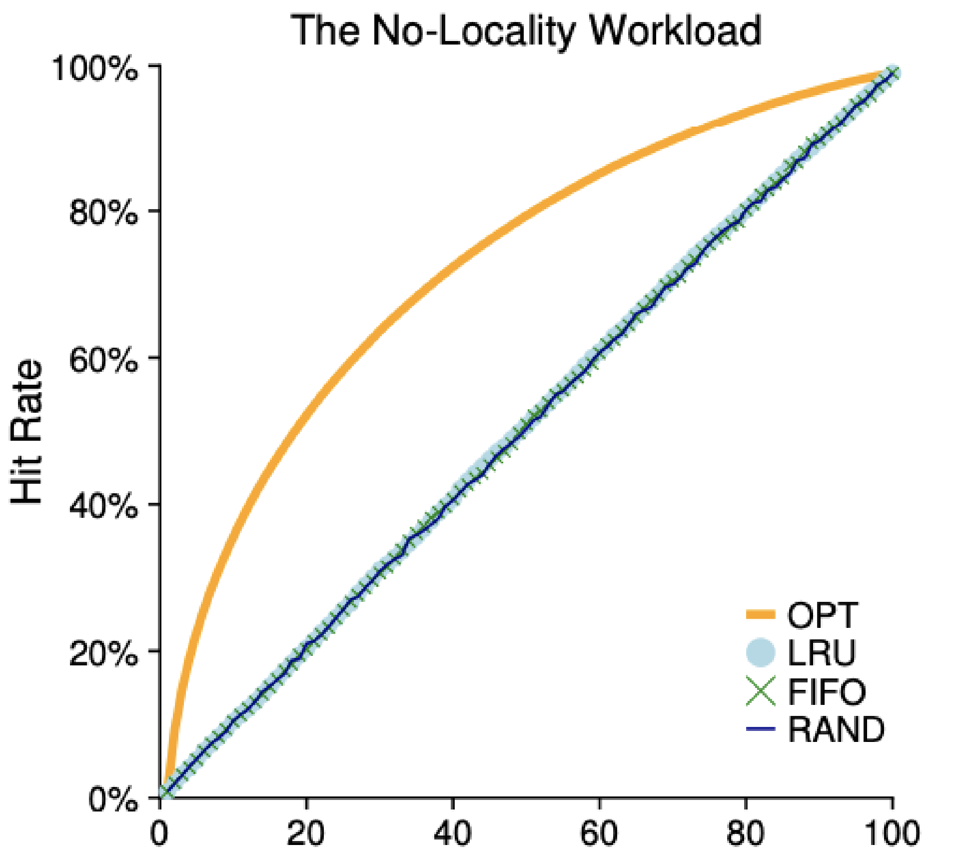
Can also make clock algorithm more efficient

- First try to find a page with dirty bit and use bit both 0, because it is lower cost to replace

What if All Accesses Were Random?

Without locality, none of the policies (except the cheating OPT) provide cost-effective benefit

If main memory is 50% of total pages, only 50% hit rate



Percent of total pages that fit in main memory

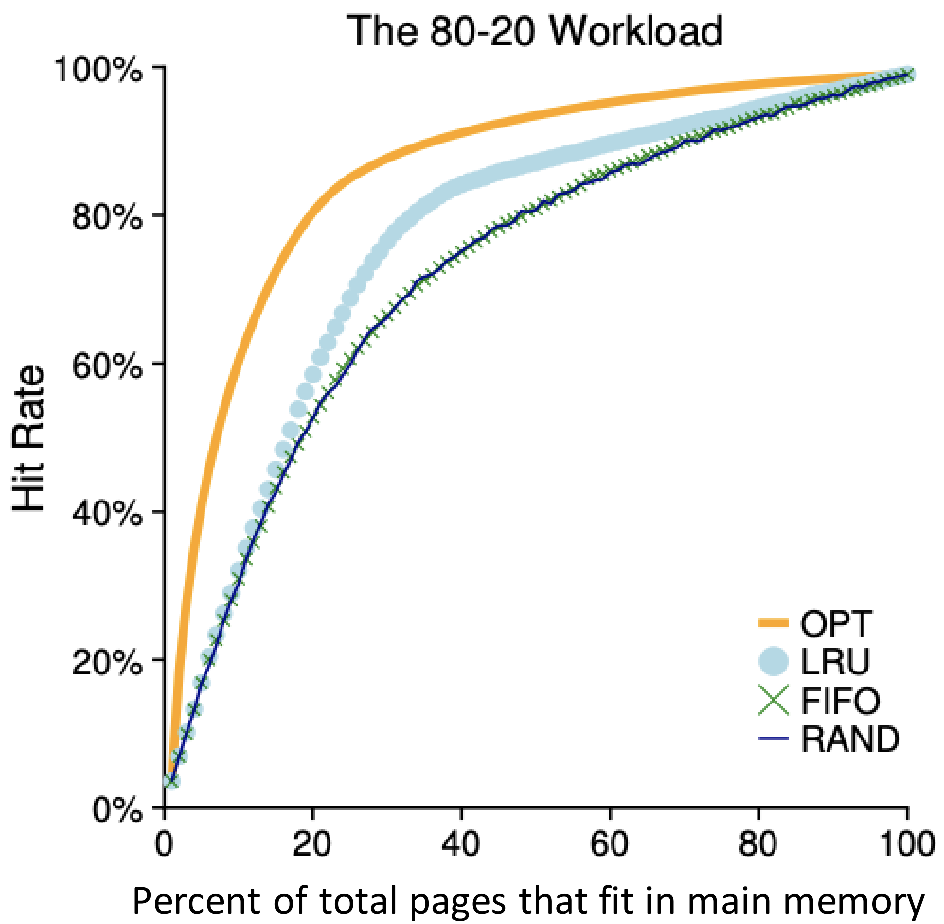
80-20 Workload

More typical, the **80-20 workload** (80% of accesses are to 20% of pages)

- A few pages get most accesses
- Most pages get few accesses

Results from locality (either spatial or temporal)

LRU performs closest to optimal, FIFO and RAND perform the same

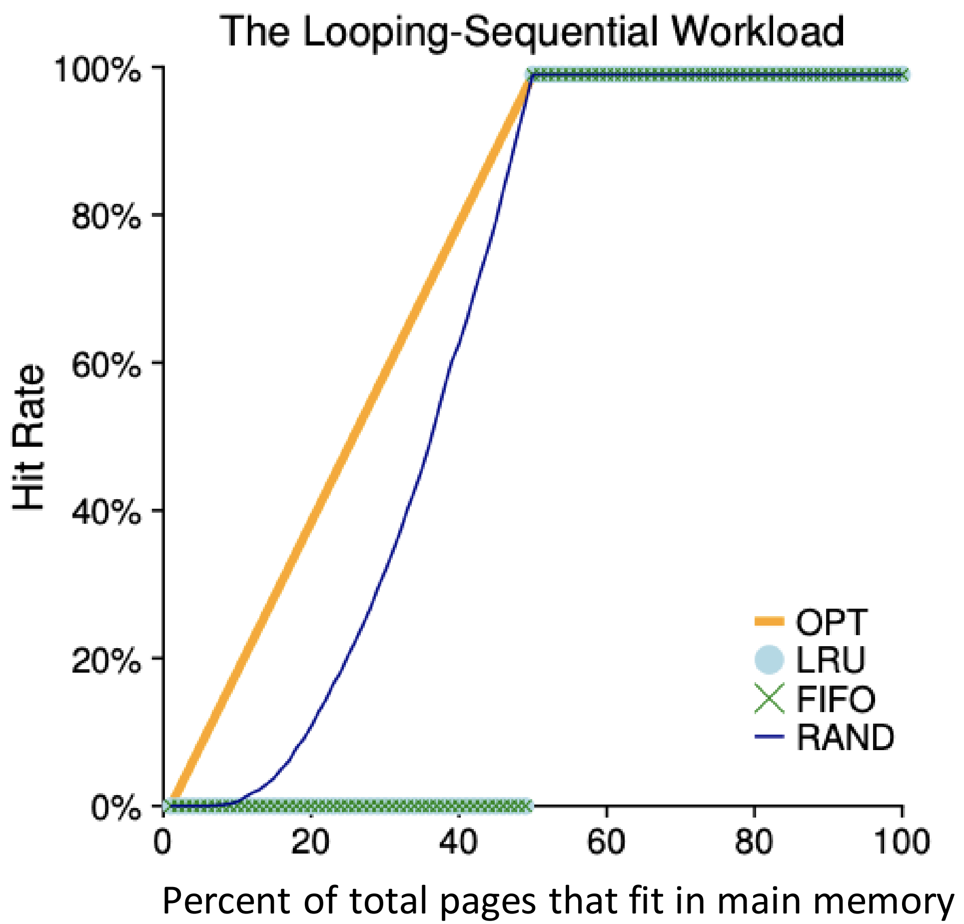


Looping-Sequential Workload

Assume loop repeatedly reads pages 0 to 49 in increasing order

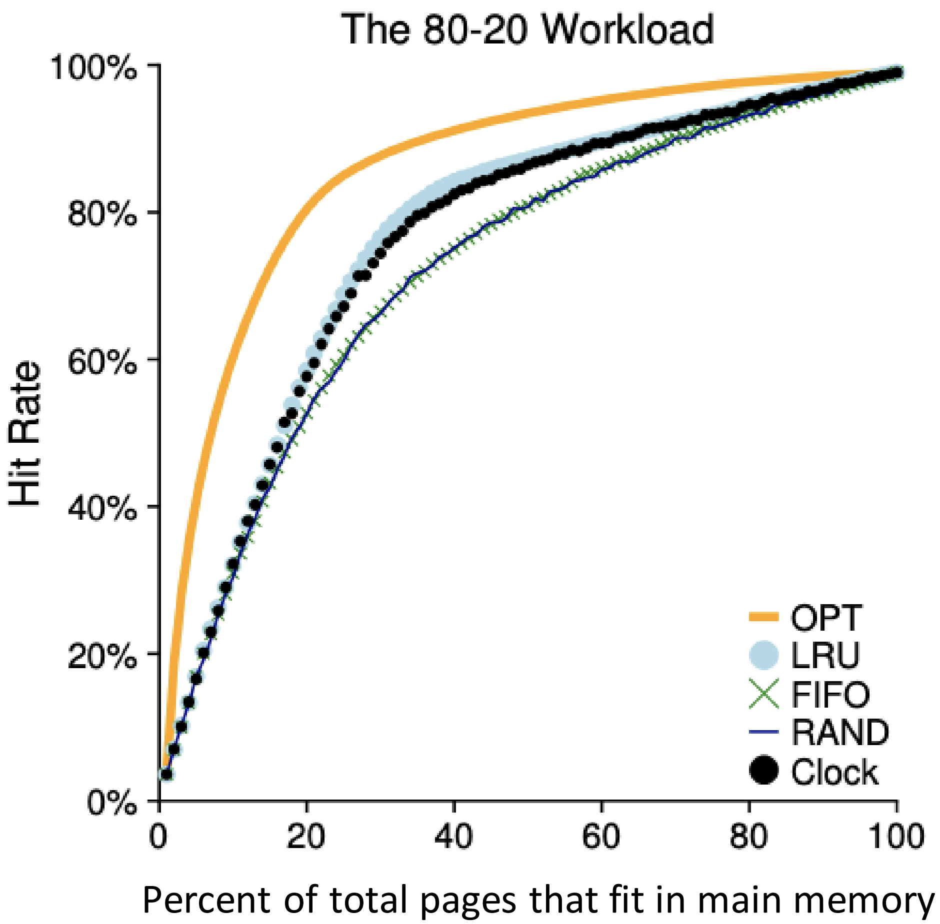
When cache size is below 50, LRU and FIFO have same corner case that causes every access to be a miss

Random avoids corner cases



Clock Policy with 80-20 Workload

Clock is a very close approximation of LRU



Thrashing

Example: System has two processes that both sequentially read from N pages in a continuous loop. The System only has enough main memory to store N pages.

Thrashing is when process is spending more time on handling page-fault than executing

- Starts at one processes and snowballs into several processes thrashing
- Multiprogramming and multitasking make it worse, not better
- Sudden and extreme drop in system performance

Need to reduce the amount of multiprogramming and multitasking to avoid thrashing

L13: Concurrency

(based on Ch. 26)

Slides revised from
Matthew Tancreti
Iowa State University

Concurrency

We have explored how OSes use concurrency

- multiprogramming – to utilize the CPU efficiently when programs block for I/O
- multitasking – to give all programs a fair slice of the CPU and make progress

What about providing concurrency within a process?

- Programs have tasks that need to block for I/O
- Programs want to be responsive to the user while performing tasks in the background
- Programs want to distribute their computations across multiple CPUs to complete faster

Processes can have multiple **threads** - concurrent points of execution

How does the OS provide programs concurrent threads of execution?

What are Threads?

Multiple points of execution of a program within a single process

Thread2

Thread1

Thread0

```
#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;
}
```

What are Threads?

Multiple points of execution means each thread must have its own program counter (PC - next instruction to execute) and call stack

Thread2
PC
Stack2

Thread1
PC
Stack1

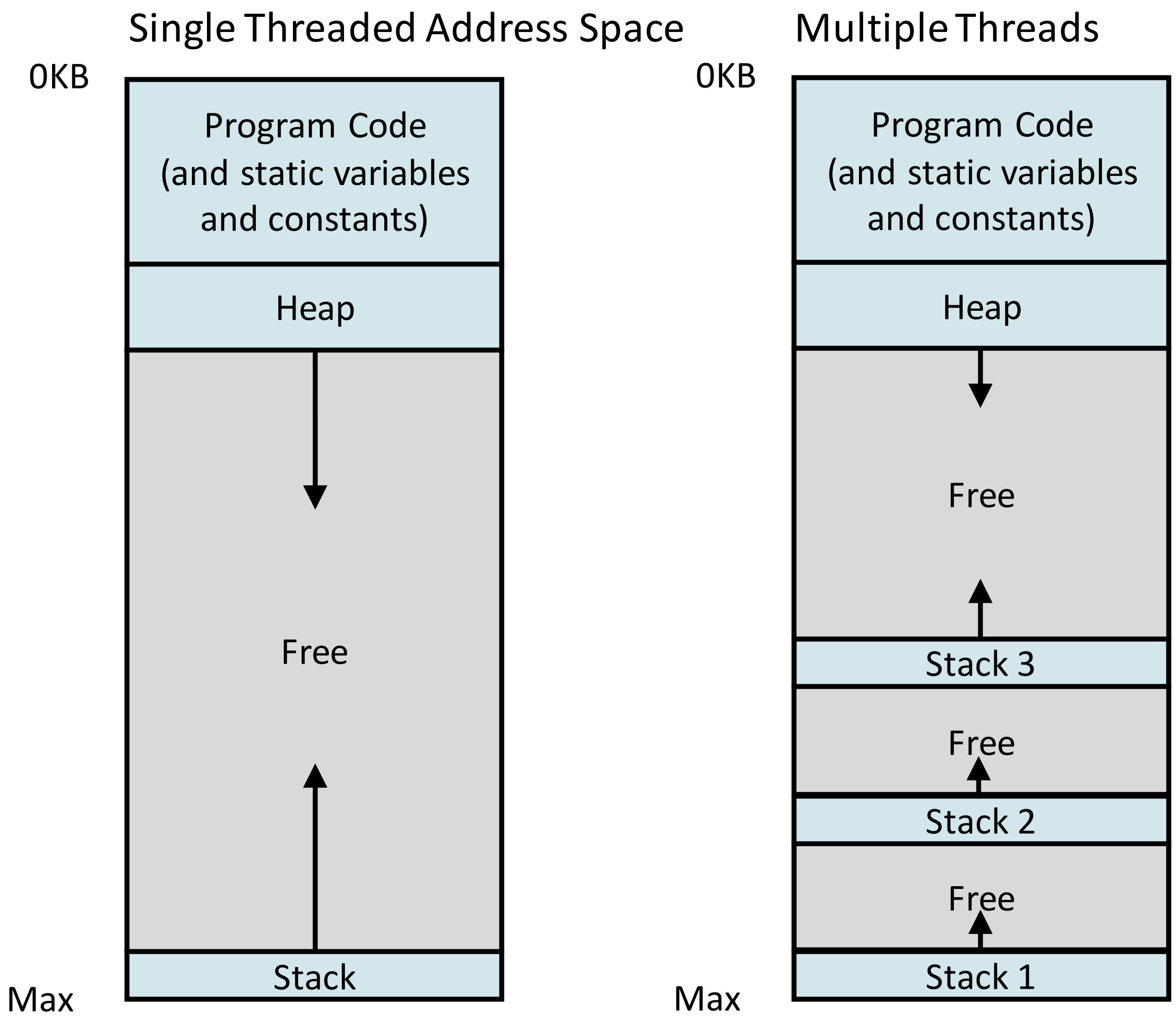
Thread0
PC
Stack0

```
#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;
}
```

Multi-Threaded Address Space

Each thread has its own stack segment

Program, data and heap are shared with the process



Thread Control Block (TCB)

Recall the Process Control Block (PCB)

- Keeps track of information for each process
- Stores execution context to enable context switching out of and back into the process

Thread Control Block (TCB) is similar, but

- threads share process code, data and heap segments

Scheduler uses both PCBs and TCBs to decide which thread to run next

Process Control Block	Thread Control Block
Process ID (pid)	Thread ID (tid)
State (e.g., running, runnable, blocked)	State (e.g., running, runnable, blocked)
Program Counter	Program Counter
CPU Register values (context)	CPU Register values (context)
Stack pointer	Stack pointer
Pointers to code, data and heap segments	
Pointer to PCB of the parent process	Pointer to the PCB of the process that thread belongs to
Open file descriptors	

Concurrency vs Parallelism

Concurrent means multiple threads making progress in time but may be implemented by time-sharing, can be on one or more CPU cores

Parallel means multiple threads instructions executing independently on multiple CPU cores

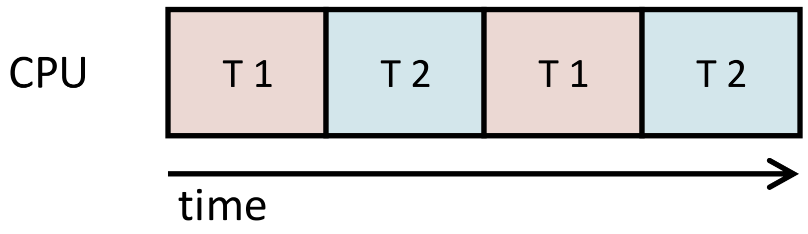
Concurrency of threads enables

- **I/O overlap** - can overlap blocking I/O with other program tasks (same concept as multiprogramming)
- **Responsiveness** - user can continue to interact with system even when program is performing heavy processing in the background

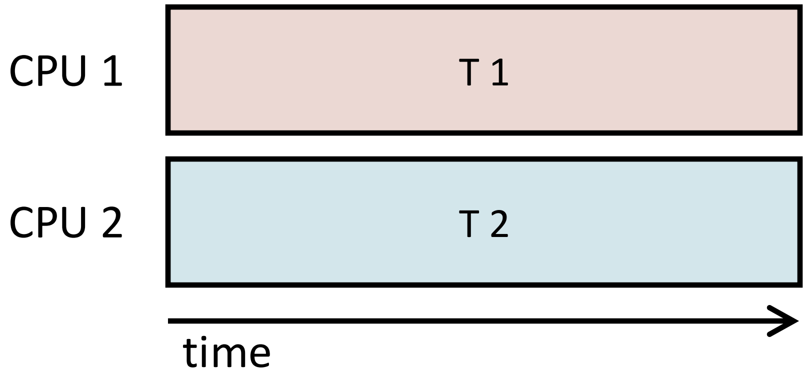
Parallelism of threads enables

- **Performance** – finish more tasks in less time by distributing load to multiple CPU cores

Concurrent execution of threads



Parallel execution of threads



Are Threads Needed?

What about multiple processes?

- Xv6 does not have user threads
- Can use fork(), pipe() and wait() to manage concurrent processes

Threads provide more convenience and better performance

- Simple memory sharing (all threads share same data and heap)
- Lower cost of thread creation (don’t need to allocate new address space, just stack)
- Lower cost of context switch (only stack and registers change)

Example

```
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;

void *mythread(void *arg) {
    printf("thread %s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++) {
        counter++;
    }
    printf("thread %s: end\n", (char *) arg);
    return NULL;
}


int main() {
    pthread_t p1, p2;
    printf("main: begin\n");

    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

When can Context Switch Occur?

Scheduler can decide to context switch to another thread at any instruction

High level code

```
counter = counter + 1;
```

Assembly instructions

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Can switch here

Or here

Or here

The Problem (Race Condition)

Concurrent update of shared memory can result in **race condition** bug

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**