Recap

Replacement Policies

- Clock algorithm (approximate LRU)
- Comparisons/Limitations of the policies

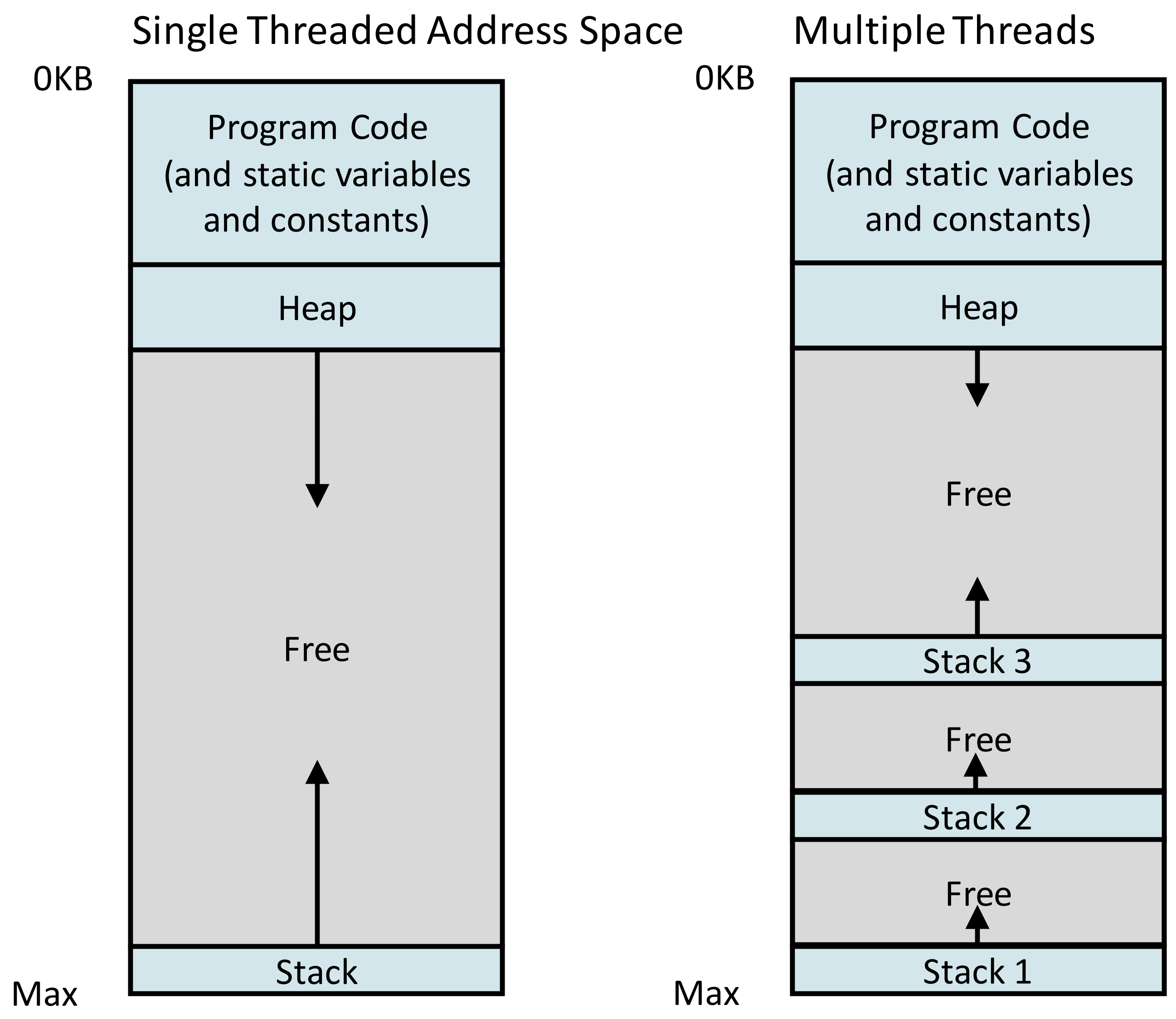
Concurrency

- Muti-thread process: address space and TCB/PCB

Multi-Threaded Address Space

Each thread has its own stack segment

Program, data and heap are shared with the process


Single Threaded Address Space
0KB
Program Code (and static variables and constants)
Heap
Free
Stack
Max
Multiple Threads
0KB
Program Code (and static variables and constants)
Heap
Free
Stack 3
Free
Stack 2
Free
Stack 1
Max

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

Process ID (pid)
State (e.g., running, runnable, blocked) [no need for multi-thread process]
Program Counter [no need for m-t process]
CPU Register values (context) [no need for m-t process]
Stack pointer [no need for m-t process]

Pointers to code, data and heap segments
Pointer to PCB of the parent process
Open file descriptors

Thread Control Block

Thread ID (tid)
State (e.g., running, runnable, blocked)
Program Counter
CPU Register values (context)
Stack pointer

Pointer to the PCB of the process that thread belongs to

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

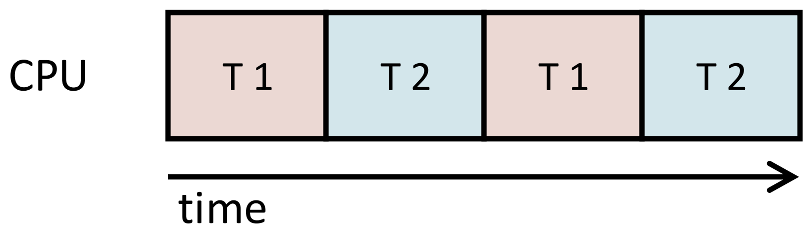
Concurrency of threads enables

- **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

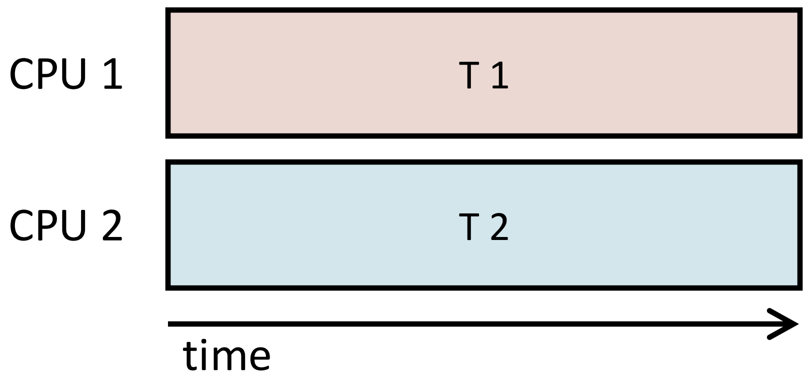
Parallelism of threads enables

- **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;
```

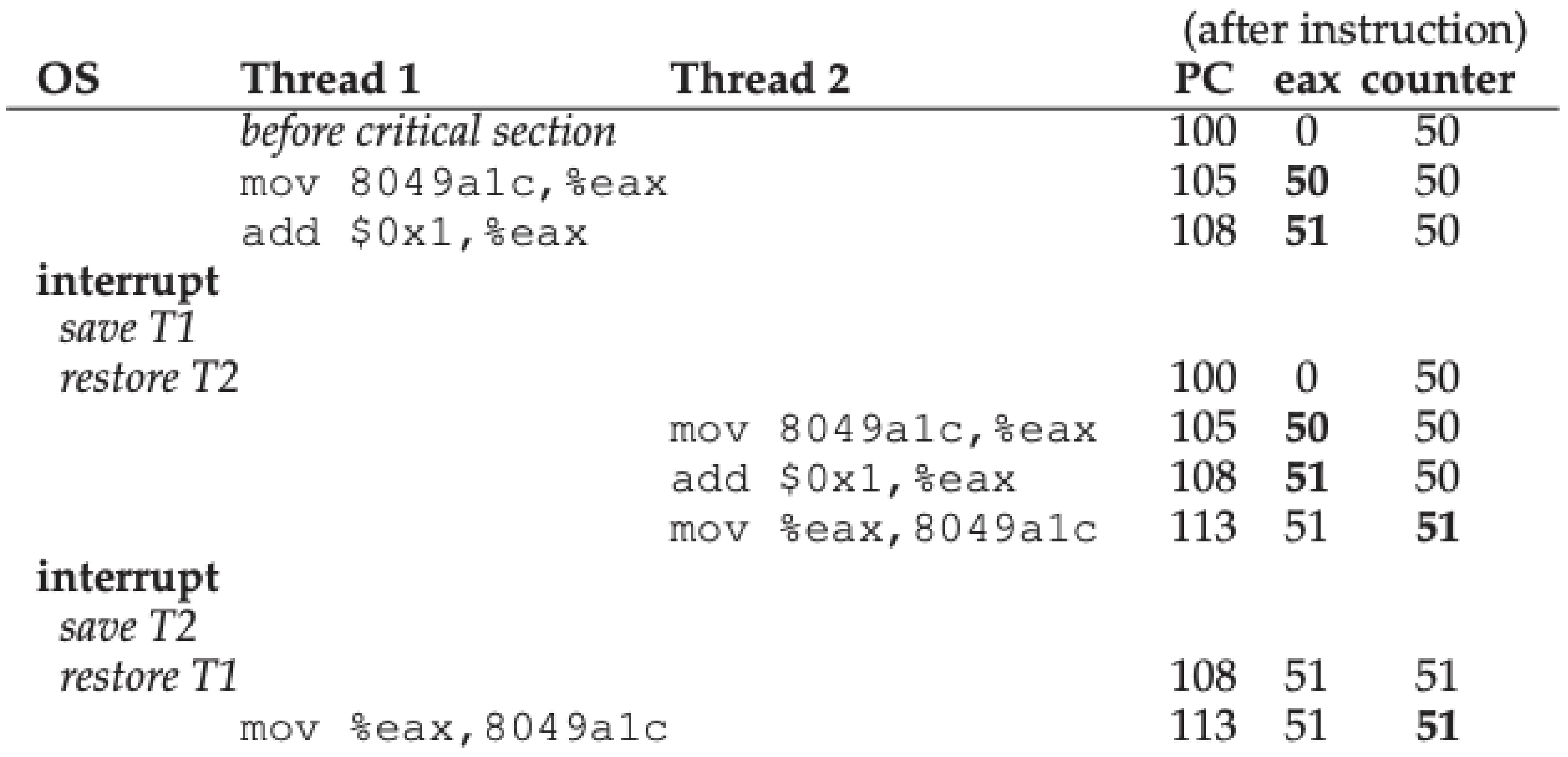
→

Assembly instructions

```
mov 0x8049a1c, %eax    ←  Can switch here
add $0x1, %eax         ←  Or here
mov %eax, 0x8049a1c    ←  Or here
```

The Problem (Race Condition)

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



			(after instruction)		
OS	**Thread 1**	**Thread 2**	**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**

Thread API

(based on Ch. 27)

Slides revised from
Matthew Tancreti
Iowa State University

Concurrency APIs

Concurrency is difficult, developers need support to create safe multi-threaded programs

We first look at a common API for threads called POSIX pthreads

After this we will examine how systems implement libraries that provide multi-threading support

Thread Library

Thread library provides API for creating and managing threads

Can be implemented in two places

- Entirely in user space, the library requires no special OS support
- OS supported, uses system calls to operate at kernel level

Examples of common thread libraries

- POSIX pthreads: can be implemented at user level or kernel level
- Windows threads: implemented at kernel level
- Java threads: JVM (Java Virtual Machine) implements threads using thread libraries available on host system

Pthreads

Portable Operating System Interface (**POSIX**) are set of standards for OS APIs

pthreads provide API for thread creation and synchronization

- API specifies behavior of the thread library, implementation is up the development of the library
- Available on many Unix-like OSes (Linux, Unix, BSD, MacOS)

Using pthreads in Linux

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

Compile and link with the pthread library: `gcc myprogram.c -lpthread`

pthread_create

```
#include <pthread.h>
int
pthread_create( pthread_t *          thread,
          const pthread_attr_t *   attr,
                void *             (*start_routine)(void*),
                void *             arg);
```

Create a new thread

- thread points to a buffer that stores the ID of the new thread
- attr points to a structure containing the attributes of the new thread
  - If attr is NULL, the thread is created with default attributes
- The new thread starts execution by invoking start_routine()
- arg is a pointer to the argument of start_routine()
  - If multiple arguments are needed, arg points to a data structure that contains all arguments
- Returns 0 on success, returns an error number on error

pthread_create

The new thread executes concurrently with the parent thread

The new thread runs until one of the following happens

- It returns from <u>start_routine</u>
- It calls **pthread_exit()**
- Any of the threads in the process calls **exit()** or the main thread performs a return from **main()**. This causes the termination of all threads in the process.

pthread_create() Example

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

void *my_thread (void *arg)
{
        char *msg = (char *) arg;
        printf("Thread says \%s\n", msg);
}

int main (int argc, char *argv[])
{     pthread_t t;
      char msg[20] = "Hello World";
      pthread_create(&t, NULL, my_thread, msg);
      sleep(3); //what happens if this statement is removed?
      return 0;
}
```

pthread_exit()

```
#include <pthread.h>
void pthread_exit(void *retval)
```

Terminate calling thread

- The function returns a value via retval that is available to another thread in the same process that calls pthread_join()
- The function does not return to the caller

pthread_join()

```
#include <pthread.h>
int pthread_join(pthread_t th, void **retval)
```

Wait for a thread to terminate

- th: the thread to wait for
- If retval is not NULL, then **pthread_join**() copies the exit status of the target thread into the location pointed to by retval
- Returns 0 on success, returns an error number on error
- When a thread terminates, its TCB is not deallocated until another thread performs **pthread_join()** on it

Pthread_join() Example

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

int n;

void * thread_start(void *arg)
{
  int *id = (int *)arg;
  while (n != *id);
  printf("Thread %d \n", *id);
  n--;
  pthread_exit(NULL);
}
```

```
int main()
{
  n = 0;
  int id1=1;
  int id2=2;
  int id3=3;

  pthread_t t1, t2, t3;
  printf("Parent creating threads\n");
  pthread_create(&t1, NULL, thread_start, &id1);
  pthread_create(&t2, NULL, thread_start, &id2);
  pthread_create(&t3, NULL, thread_start, &id3);
  printf("Threads created\n");
  n = 3;
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  printf("Threads are done\n");
  return 0;
}
```

Locks

Locks provide mutual exclusion to a critical section of code

- Mutual exclusion – only one thread at a time
- Critical section – a section of code that can only be executed by one thread at a time and the thread must execute the code to completion before another thread can enter

Shared variables can be accessed in a critical section

Initializing a Lock

```
int pthread_mutex_init(
          pthread_mutex_t     * mutex,
    const pthread_mutexattr_t * mutexattr)
```

This function initializes a mutex lock

- First parameter is a pointer to the mutex
- Second parameter specifies the attributes of the mutex
- If mutexattr is NULL, default attributes are used
- Return 0 on success; otherwise, an error number is returned

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

This function destroys a mutex

- The mutex must be unlocked when called
- Attempting to destroy a locked mutex results in undefined behavior
- Return 0 on success; otherwise, an error number is returned

Using Lock to Create Critical Section

```
int pthread_mutex_lock(pthread_mutex_t*mutex)
```

Acquire the mutex lock

- If the mutex is unlocked, it becomes locked and owned by the calling thread
- If the mutex is already locked, the calling thread blocks until the mutex is unlocked
- Return 0 on success; otherwise, an error number is returned

```
int pthread_mutex_unlock(pthread_mutex_t*mutex)
```

- Release the mutex lock
- This function unlocks a mutex if called by the owning thread
    - An error will be returned if the mutex is owned by another thread
- Return 0 on success; otherwise, an error number is returned

Example

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

static volatile int counter = 0;
pthread_mutex_t mutex;


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

        pthread_mutex_lock(&mutex);
        counter++;

        pthread_mutex_unlock(&mutex);
    }
    printf("thread %s: end\n", (char *) arg);
    return NULL;
}


int main() {
    pthread_t p1, p2;

    printf("main: begin\n");

    pthread_mutex_init(&mutex,NULL);

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

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

    Pthread_mutex_destroy(&mutex);

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

Condition Variables

Condition variables are used to put a thread to sleep until another thread signals it

```
int pthread_cond_wait(
        pthread_cond_t   * cond,
        pthread_mutext_t * mutex)
```

Wait for signal

```
int pthread_cond_signal(
        pthread_cond_t   * cond)
```

Send signal