Lecture 39:
Processes and Threads

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
June 4th, 2010

The Fork System Call

- The **fork()** system call creates a "clone" of the calling process.
- Identical in every respect except
  - the parent process is returned a non-zero value (namely, the process id of the child)
  - the child process is returned zero.
- The process id returned to the parent can be used by parent in a **wait** or **kill** system call.

Example using fork

```
 1. #include <unistd.h>
 2. main(){
 3.    pid_t pid;
 4.    printf("Just one process so far\n");
 5.    pid = fork();
 6.    if (pid == 0) /* code for child */
 7.       printf("I'm the child\n");
 8.    else if (pid > 0) /* code for parent */
 9.       printf("The parent, child pid =%d\n",
10.              pid);
11.    else /* error handling */
12.       printf("Fork returned error code\n");
13. }
```

Spawning Applications

fork() is typically used in conjunction with exec (or variants)

```
pid_t pid;
if ( ( pid = fork() ) == 0 ) {
   /* child code: replace executable image */
   execv( "/usr/games/tetris", "-easy" )
} else {
   /* parent code: wait for child to terminate */
   wait( &status )
}
```

exec System Call

A family of routines, **execl**, **execv**, ..., all eventually make a call to **execve**.

execve(program_name, arg1, arg2, ..., environment)

- text and data segments of current process replaced with those of **program_name**
- stack reinitialized with parameters
- open file table of current process remains intact
- the last argument can pass environment settings
- as in example, **program_name** is actually path name of executable file containing program

Note: unlike subroutine call, there is no return after this call. That is, the program calling exec is gone forever!

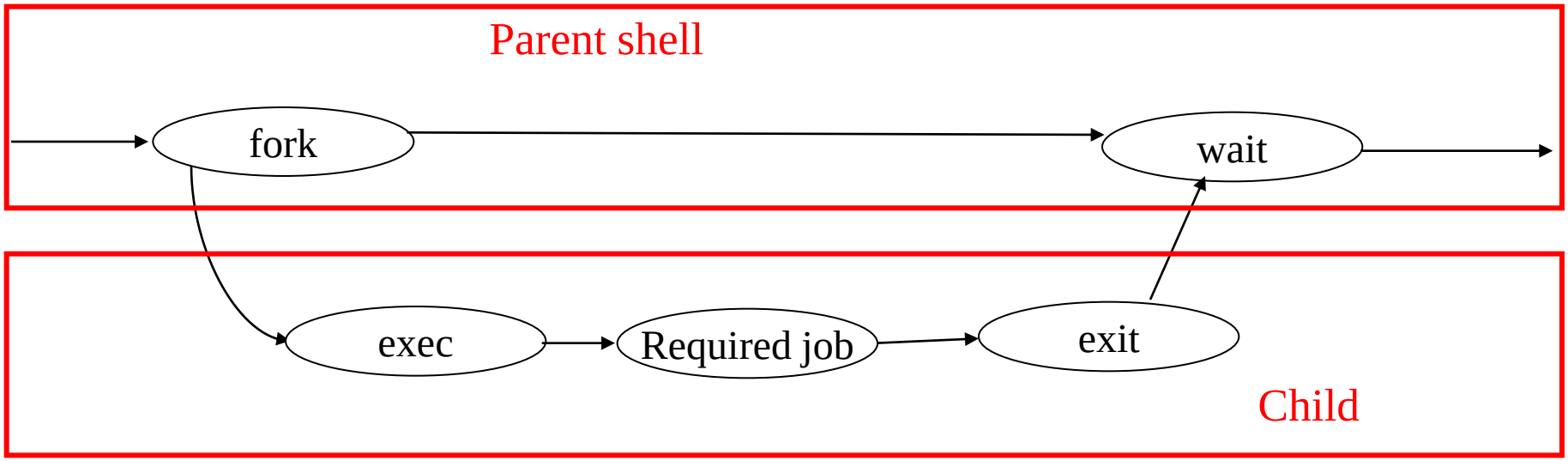
Parent-Child Synchronization

- **exit(status)** - executed by a child process when it wants to terminate. Makes **status** (an integer) available to parent.
- **wait(&status)** - suspends execution of process until *some* child process terminates
  - **status** indicates reason for termination
  - return value is process-id of terminated child
- **waitpid (pid, &status, options)**
  - pid can specify a specific child
  - Options can be to wait or to check and proceed

Process Termination

- Besides being able to terminate itself with **exit**, a process can be killed by another process using **kill**:
  - **kill(pid, sig)** - sends signal **sig** to process with process-id **pid**. One signal is **SIGKILL** (terminate the target process immediately).
- When a process terminates, all the resources it owns are reclaimed by the system:
  - "process control block" reclaimed
  - its memory is deallocated
  - all open files closed and Open File Table reclaimed.
- Note: a process can kill another process only if:
  - it belongs to the same user
  - super user

How shell executes a command



- when you type a command, the shell forks a clone of itself
- the child process makes an exec call, which causes it to stop executing the shell and start executing your command
- the parent process, still running the shell, waits for the child to terminate

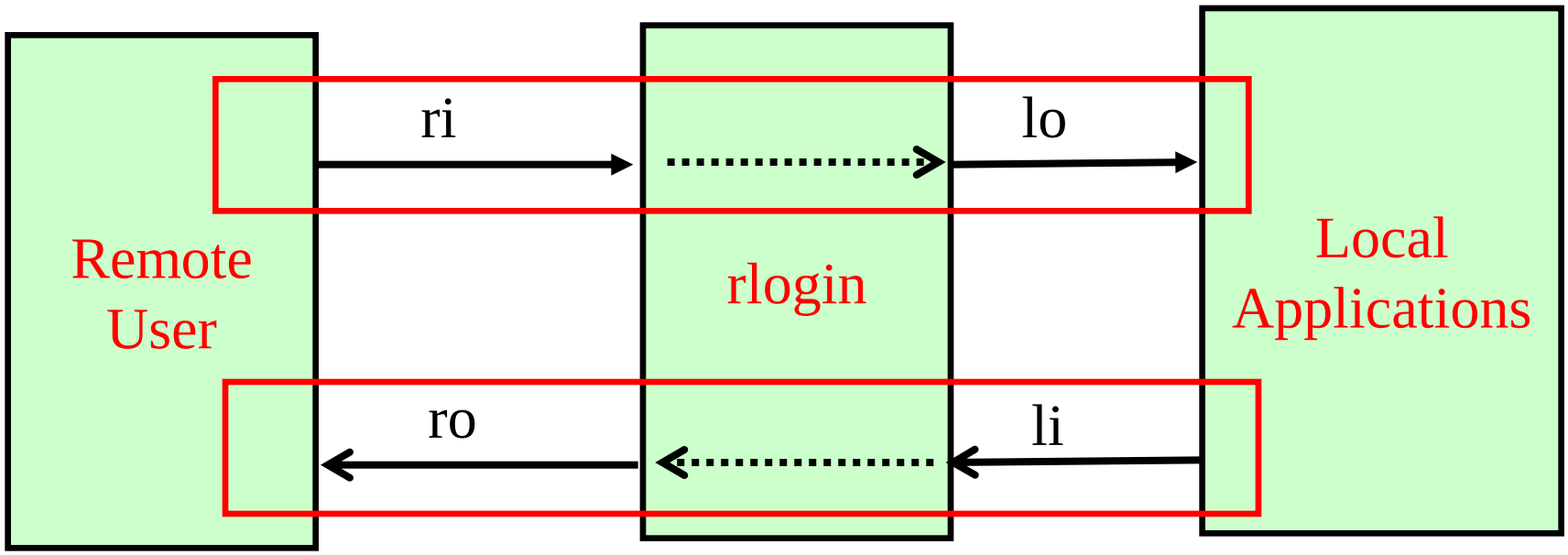
Outline for Today

- Motivation and definitions
- Processes
- Threads
- Synchronization constructs
- Speedup issues
  - Overhead
  - Caches
  - Amdahl's Law

Introduction to Threads

- Multitasking OS can do more than one thing concurrently by running more than a single process
- A process can do several things concurrently by running more than a single thread
- Each thread is a different stream of control that can execute its instructions independently.
- Ex: A program (e.g. Browser) may consist of the following threads:
  - GUI thread
  - I/O thread
  - Computation thread

When are threads useful?



Challenges in single-threaded soln

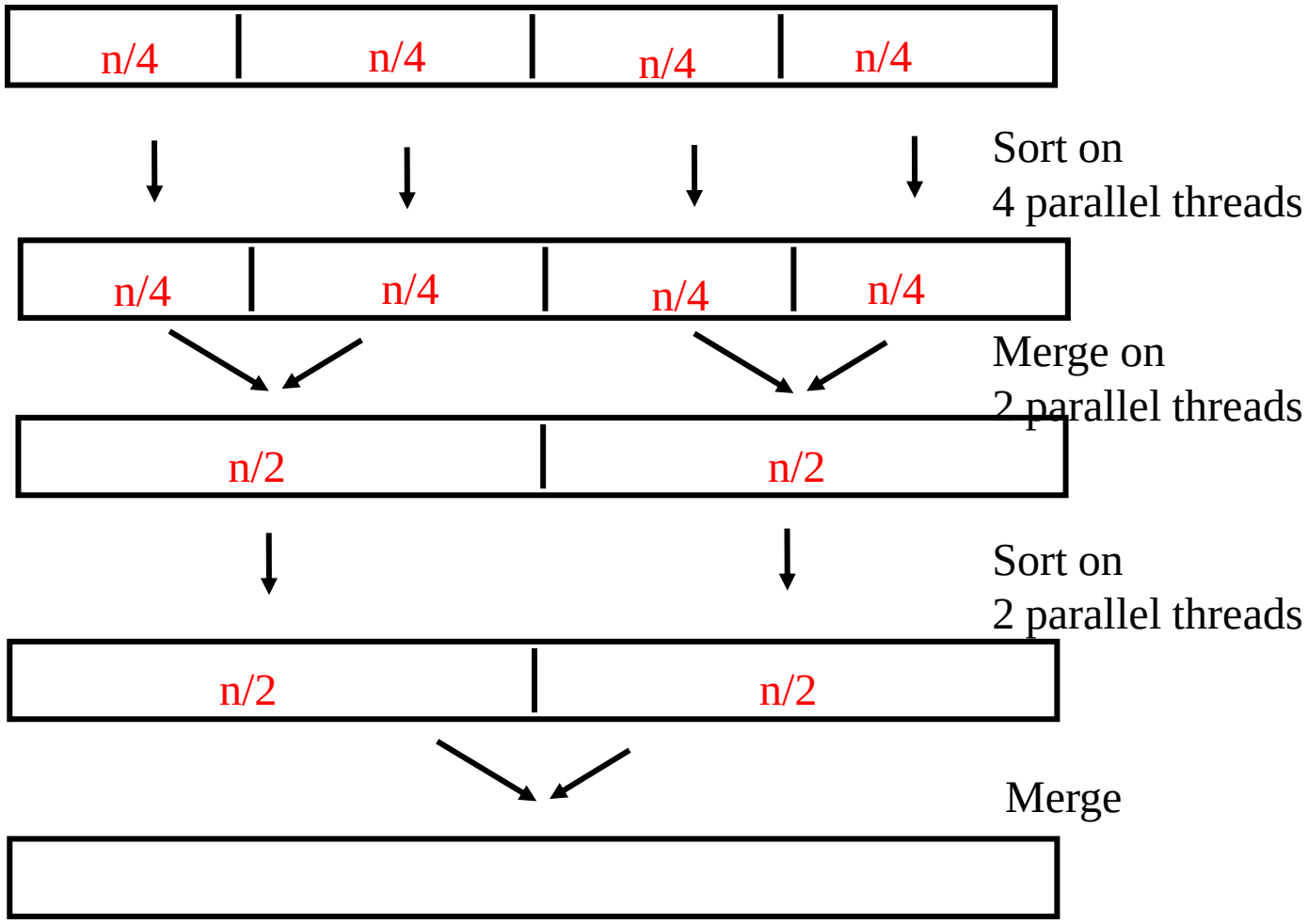
- There are basically 4 activities to be scheduled
  - read(li), read(ri), write(lo), write(ro)
- **read** and **write** are blocking calls
- So before issuing any of these calls, the program needs to check readyness of devices, and interleave these four operations
  - System calls such as FD_SET and select
- Bottomline: single-threaded code can be quite tricky and complex

Solution with Threads

```
incoming(int ri, lo){
 int d=0;
 char b[MAX];
 int s;
 while (!d) {
  s=read(ri,b,MAX);
  if (s<=0) d=1;
  if (write(lo,b,s)<=0)
  d=1;
  }
}
```

```
outgoing(int li, ro){
 int d=0;
 char b[MAX];
 int s;
 while (!d) {
  s=read(li,b,MAX);
  if (s<=0) d=1;
  if (write(ro,b,s)<=0)
  d=1;
  }
}
```

Parallel Algorithms: Eg. mergesort



Is there a speed-up ?

Benefits of Threads: Summary

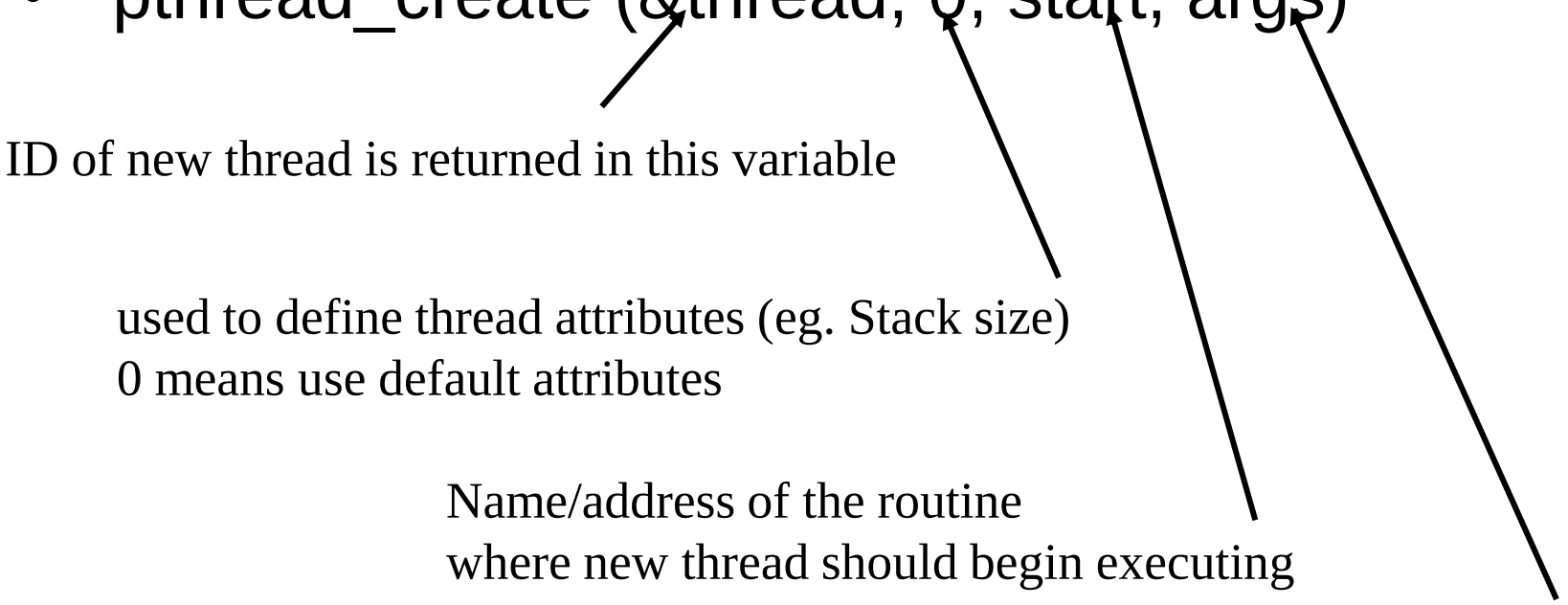
1. Superior programming model of parallel sequential activities with a shared store
2. Easier to create and destroy threads than processes.
3. Better CPU utilization (e.g. dispatcher thread continues to process requests while worker threads wait for I/O to finish)
4. Guidelines for allocation in multi-processor systems

Processes and Threads

- A UNIX Process is
  - a running program with
  - a bundle of resources (file descriptor table, address space)
- A thread has its own
  - stack
  - program counter (PC)
  - All the other resources are shared by **all** threads of that process. These include:
    - open files
    - virtual address space
    - child processes

Thread Creation

- POSIX standard API for multi-threaded programming
- A thread can be created by **pthread_create** call
- pthread_create (&thread, 0, start, args)



Sample Code

```
typedef struct { int i, o } pair;
rlogind ( int ri, ro, li, lo) {
  pthread_t in_th, out_th;
  pair in={ri,lo}, out={li,ro};
  pthread_create(&in_th,0, incoming, &in);
  pthread_create(&out_th,0, outgoing, &out);
}
```

Note: 2 arguments are packed in a structure

Problem: If main thread terminates, memory for in and out structures may disappear, and spawned threads may access incorrect memory locations

If the process containing the main thread terminates, then all threads are automatically terminated, leaving their jobs unfinished.

Ensuring main thread waits…

```
typedef struct { int i, o } pair;
rlogind ( int ri, ro, li, lo) {
  pthread_t in_th, out_th;
  pair in={ri,lo}, out={li,ro};
  pthread_create(&in_th,0, incoming, &in);
  pthread_create(&out_th,0, outgoing, &out);
  pthread_join(in_th,0);
  pthread_join(out_th,0);
}
```

Thread Termination

- A thread can terminate
  1. by executing **pthread_exit**, or
  2. By returning from the initial routine (the one specified at the time of creation)
- Termination of a thread unblocks any other thread that's waiting using **pthread_join**
- Termination of a process terminates all its threads

Creating and destroying PThreads

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

#define NUM_THREADS 5
pthread_t threads[NUM_THREADS];

int main(void) {
  for(int ii = 0; ii < NUM_THREADS; ii+=1) {
    (void) pthread_create(&threads[ii], NULL, threadFunc, (void *) ii);
  }

  for(int ii = 0; ii < NUM_THREADS; ii+=1) {
    pthread_join(threads[ii],NULL); // blocks until thread ii has exited
  }

  return 0;
}

void *threadFunc(void *id) {
  printf("Hi from thread %d!\n",(int) id);
  pthread_exit(NULL);
}
```

To compile against the PThread library, use gcc's -lpthread flag!

Side: OpenMP is a common alternative!

- PThreads aren't the only game in town
- OpenMP can automatically parallelize loops and do other cool, less-manual stuff!

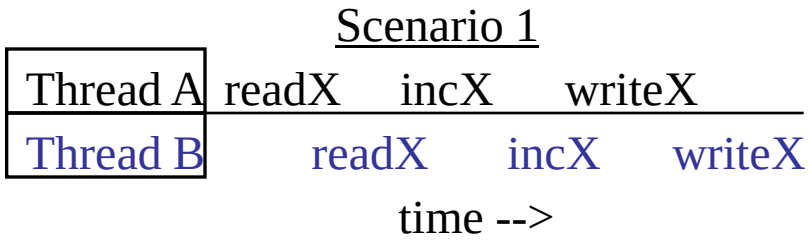
```
#define N 100000
int main(int argc, char *argv[]){
   int i, a[N];
   #pragma omp parallel for
   for (i=0;i<N;i++)
      a[i]= 2*i;
   return 0;
}
```

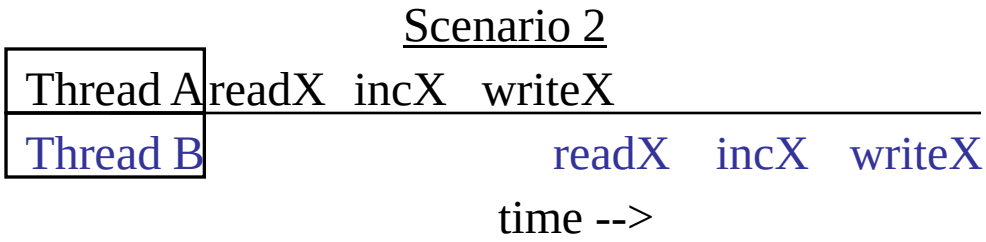
Outline for Today

- Motivation and definitions
- Processes
- Threads
- Synchronization constructs
- Speedup issues
  - Overhead
  - Caches
  - Amdahl's Law

How can we make threads cooperate?

- If task can be completely decoupled into independent sub-tasks, cooperation required is minimal
  - Starting and stopping communication
- Trouble when they need to share data!
- Race conditions:





- We need to force some serialization
  - Synchronization constructs do that!

Lock / mutex semantics

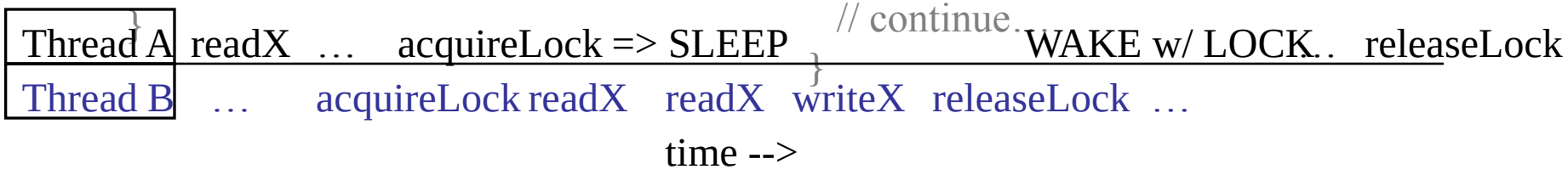
- A *lock* (mutual exclusion, mutex) guards a *critical section* in code so that only one thread at a time runs its corresponding section
  - *acquire* a lock before entering crit. section
  - *releases* the lock when exiting crit. section
  - Threads share locks, one per section to synchronize
- If a thread tries to acquire an in-use lock, that thread is put to sleep
  - When the lock is released, the thread wakes up *with the lock*! (blocking call)

Lock / mutex syntax example in PThreads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int x;

threadA() {
  int temp = foo(x);
  pthread_mutex_lock(&lock);
  x = bar(x) + temp;
  pthread_mutex_unlock(&lock);
  // continue…
}
```

```
threadB() {
  int temp = foo(9000);
  pthread_mutex_lock(&lock);
  baz(x) + bar(x);
  x *= temp;
  pthread_mutex_unlock(&lock);
  // continue…
}
```



- But locks don't solve everything…
  - Problem: potential deadlock!

```
threadA() {
  pthread_mutex_lock(&lock1);
  pthread_mutex_lock(&lock2);
}
```

```
threadB() {
  pthread_mutex_lock(&lock2);
  pthread_mutex_lock(&lock1);
}
```

Condition variable semantics

- *A condition variable* (CV) is an object that threads can sleep on and be woken from
  - *Wait* or *sleep* on a CV
  - *Signal* a thread sleeping on a CV to wake
  - *Broadcast* all threads sleeping on a CV to wake
  - I like to think of them as thread pillows…
- *Always* associated with a lock!
  - Acquire a lock before touching a CV
  - Sleeping on a CV releases the lock in the thread's sleep
  - If a thread wakes from a CV it will have the lock
- Multiple CVs often share the same lock

Outline for Today

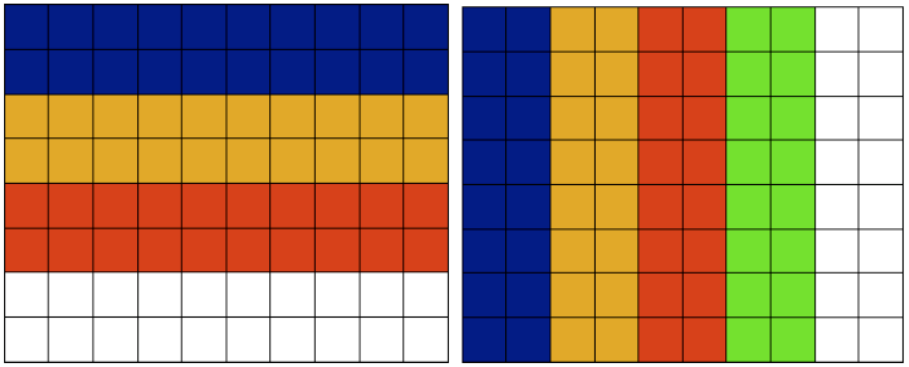
- Motivation and definitions
- Processes
- Threads
- Synchronization constructs
- Speedup issues
  - Overhead
  - Caches
  - Amdahl's Law

Speedup issues: overhead

- More threads does not always mean better!
  - I only have two cores…
  - Threads can spend too much time *synchronizing* (e.g. waiting on locks and condition variables)
- Synchronization is a form of overhead
  - Also communication and creation/deletion overhead

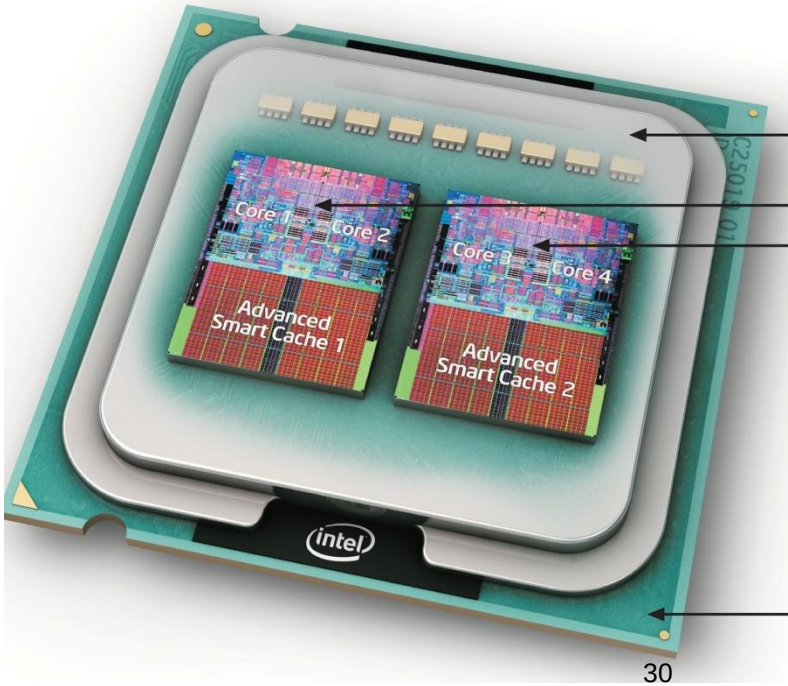
Speedup issues: caches

- Caches are often one of the largest considerations in performance
- For multicore, common to have independent L1 caches and shared L2 caches
- Can drive domain decomposition design



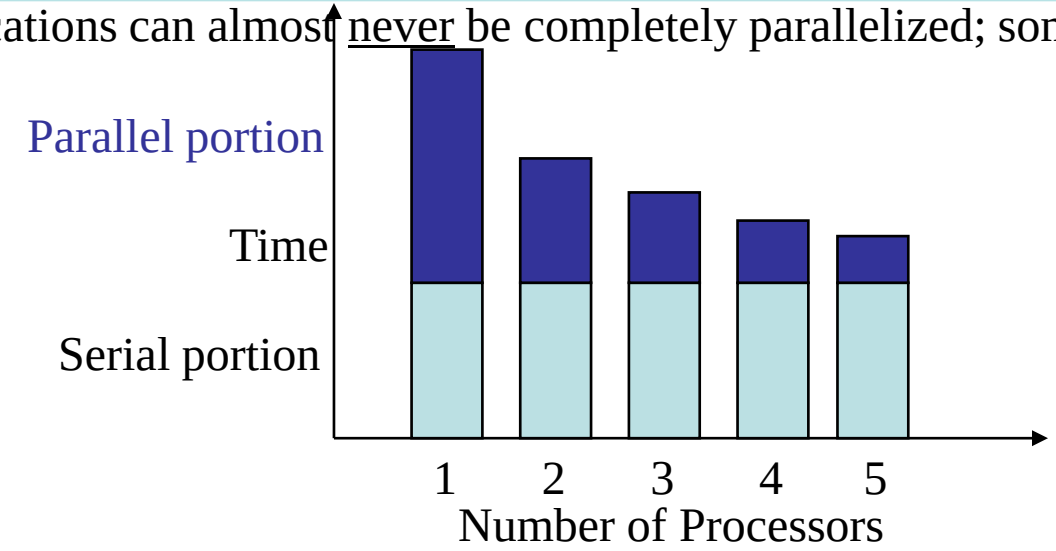
(a) "Horizontal" Decomposition

(b) "Vertical" Decomposition



Speedup Issues: Amdahl's Law

- Applications can almost never be completely parallelized; some serial code remains



- s is serial fraction of program, P is # of processors
- Amdahl's law:

Speedup(P) = Time(1) / Time(P)

$\leq 1 / (s + ((1-s) / P))$, and as $P \rightarrow \infty$

$\leq 1/s$

- Even if the parallel portion of your application speeds up perfectly, your performance may be limited by the sequential portion

Pseudo Quiz

- Super-linear speedup is possible
- Multicore is hard for architecture people, but pretty easy for software
- Multicore made it possible for Google to search the web

Quiz Answers!

- Super-linear speedup is possible
  True: more cores means simply more cache accessible (e.g. L1), so some problems may see super-linear speedup
- Multicore is hard for architecture people, but pretty easy for software
  False: parallel processors put the burden of concurrency largely on the SW side
- Multicore made it possible for Google to search the web
  False: web search and other Google problems have huge amounts of data. The performance bottleneck becomes RAM amounts and speeds! (CPU-RAM gap)

Summary

- Threads can be *awake* and ready/running on a core or *asleep* for sync. (or blocking I/O)
- Use PThreads to thread C code and use your multicore processors to their full extent!
  - `pthread_create()`, `pthread_join()`, `pthread_exit()`
  - `pthread_mutex_t`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`
  - `pthread_cond_t`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`
- Domain decomposition is a common technique for multithreading programs
- Watch out for
  - Synchronization overhead
  - Cache issues (for sharing data, decomposing)
  - Amdahl's Law and algorithm parallelizability

Questions