CS 550:
Advanced Operating Systems

Processes and Threads

Ioan Raicu
Computer Science Department
Illinois Institute of Technology

CS 550
Advanced Operating Systems
February 15th, 2011

Outline for Today

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

What's in a Process?

- Dynamic execution context of an executing program
- Several processes may run the same program, but each is a distinct process with its own state
- Process state includes:
  - The code for the running program;
  - The static data;
  - Space for dynamic data (heap)& the heap pointer (HP);
  - The Program Counter (PC) indicating the next instruction;
  - An execution stack and the stack pointer (SP);
  - Values of CPU registers;
  - A set of OS resources;
  - Process execution state (ready, running, etc.)

Creating processes in UNIX

- To see how processes can be used in application and how they are implemented, we study how processes are created and manipulated in UNIX.
- Important source of information on UNIX is “man.”
- UNIX supports multiprogramming, so there will be many processes in existence at any given time.
  - Processes are created in UNIX with the fork() system call.
  - When a process P creates a process Q, Q is called the child of P and P is called the parent of Q.

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a *process group*
- Signals can be sent all processes of a group
- Windows has no concept of process hierarchy
  - all processes are created equal

Initialization

At the root of the family tree of processes in a UNIX system is the special process init:

- created as part of the bootstrapping procedure
- process-id = 1
- among other things, init spawns a child to listen to each terminal, so that a user may log on.
- do "man init” to learn more about it

UNIX Process Control

UNIX provides a number of system calls for process control including:

- fork - used to create a new process
- exec - to change the program a process is executing
- exit - used by a process to terminate itself normally
- abort - used by a process to terminate itself abnormally
- kill - used by one process to kill or signal another
- wait - to wait for termination of a child process
- sleep - suspend execution for a specified time interval
- getpid - get process id
- getppid - get parent process id

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

```
#include <unistd.h>
main(){
  pid_t pid;
  printf(“Just one process so far\n”);
  pid = fork();
  if (pid == 0) /* code for child */
     printf(“I’m the child\n”);
  else if (pid > 0) /* code for parent */
     printf(“The parent, child pid =%d\n”,
             pid);
  else /* error handling */
     printf(“Fork returned error code\n”);
}
```

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!

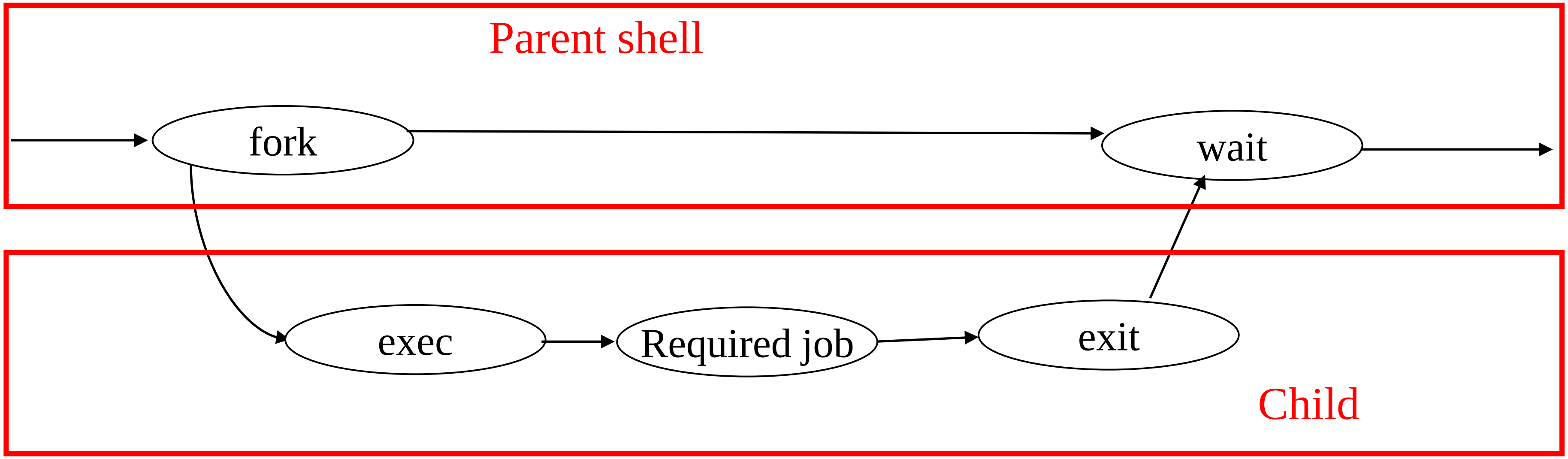
Parent-Child Synchronization

- **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

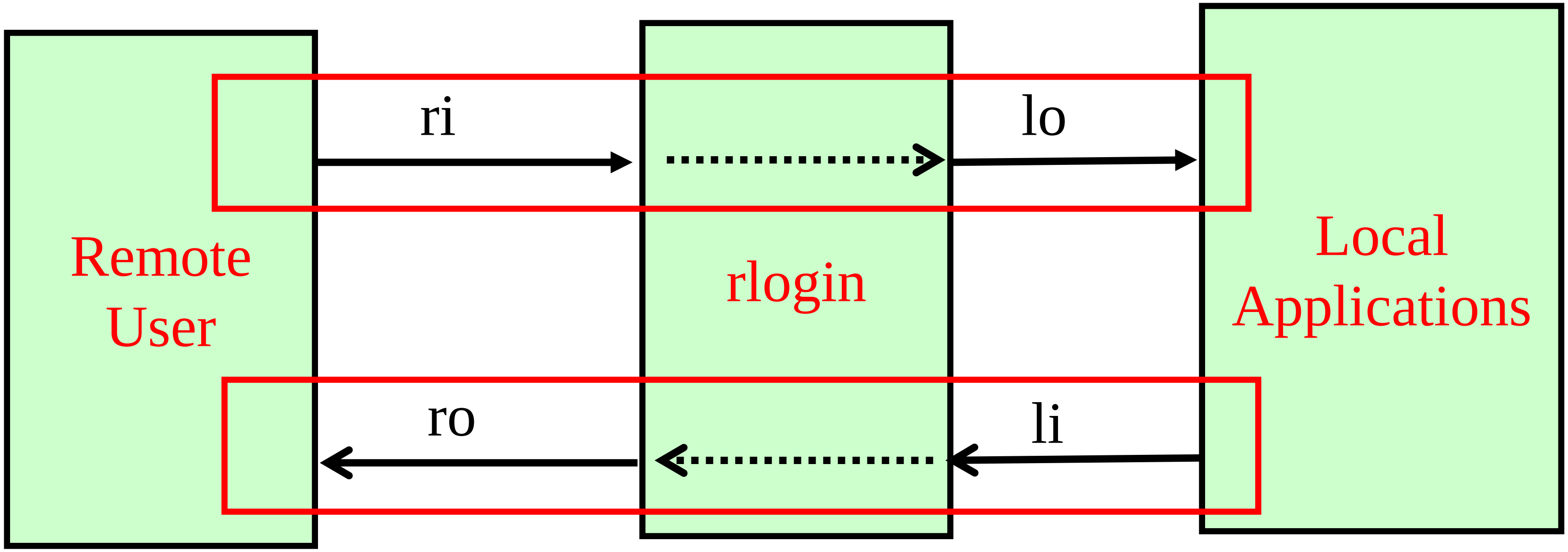
Introduction to Threads

- 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

Defining Threads

- A thread defines a single sequential execution stream within a process
- Threads are bound to a single process
- Does each thread have its own stack, PC and registers?
- Each process may have multiple threads of control within it:
  - The address space of a process is shared or not?
  - No system calls are required to cooperate among threads
  - Simpler than message passing and shared-memory

When are threads useful?



Challenges in single-threaded soln

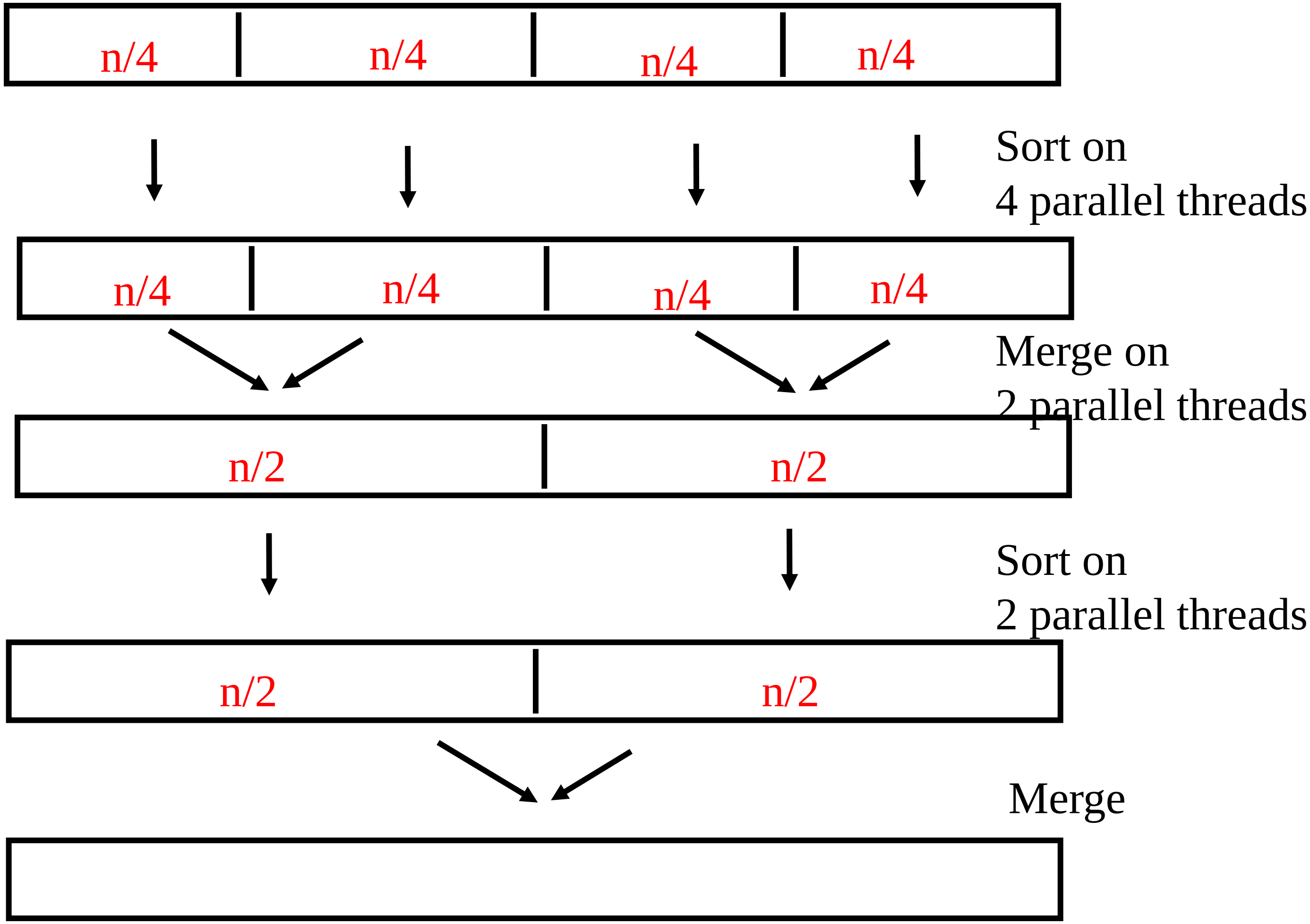
- There are basically 4 activities to be scheduled
  - read(li), read(ri), write(lo), write(ro)
- **read** and **write** are blocking calls
- So before issuing any of these calls, the program needs to check readyness of devices, and interleave these four operations
  - System calls such as FD_SET and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

ID of new thread is returned in this variable

used to define thread attributes (eg. Stack size)
0 means use default attributes

Name/address of the routine
where new thread should begin executing

Arguments passed to start

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!

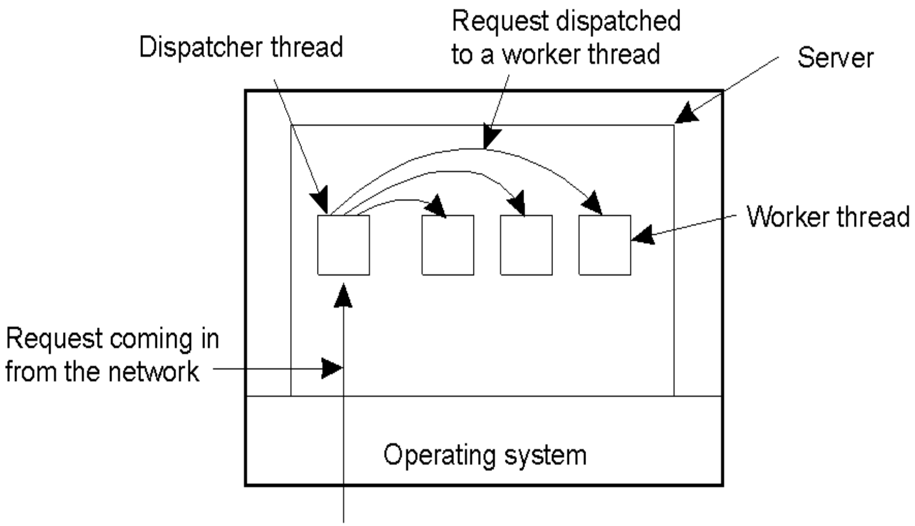
```
#define N 100000
int main(int argc, char *argv[]){
   int i, a[N];
   #pragma omp parallel for
   for (i=0;i<N;i++)
      a[i]= 2*i;
   return 0;
}
```

Multi-threaded Clients Example

- Web Browsers such as IE are multi-threaded
- Such browsers can display data before entire document is downloaded: performs multiple simultaneous tasks
  - Fetch main HTML page, activate separate threads for other parts
  - Each thread sets up a separate connection with the server
    - Uses blocking calls
  - Each part (gif image) fetched separately and in parallel
  - Advantage: connections can be setup to different sources
    - Ad server, image server, web server…

Multi-threaded Server Example

- Apache web server: pool of pre-spawned worker threads
  - Dispatcher thread waits for requests
  - For each request, choose an idle worker thread
  - Worker thread uses blocking system calls to service web request



Questions

?