FAQ

• Amdahl’s law example:
  – Person A cooks, person B eats and then Person C eats.
  – Person A cooks, then both person B and person C eat at the same time.

• Thread: sequential execution of a program segment

• A user thread can be managed by a user-level library (not managed by OS) or be managed by OS.

• How many processes (or threads) can a single core run? Many, using “concurrency” (time sharing)

• A process can have one or more threads
• Is main in C the process and functions are threads?
  – The whole program runs as a process, some functions *may* run as separate threads concurrently/in parallel.

• Why use piles: inter-process communications

• What are pipes? Functions, arrays, strings?
  – Special kind of files

• Who ensures that pipes work as expected?

• Why ordinary pipes require a parent-child relationship?
Single and Multithreaded Processes

Single-threaded process

Multithreaded process
Thread Libraries

• **Thread library** provides programmer with API for creating and managing threads

• Two primary ways of implementing
  – Library entirely in user space
  – Kernel-level library supported by the OS
Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Specification, not implementation
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
### Some Pthread management functions

<table>
<thead>
<tr>
<th>POSIX function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_cancel</td>
<td>Terminate a thread</td>
</tr>
<tr>
<td>pthread_create</td>
<td>Create a thread</td>
</tr>
<tr>
<td>pthread_detach</td>
<td>Set thread to release resources</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Exit a thread without exiting process</td>
</tr>
<tr>
<td>pthread_kill</td>
<td>Send a signal to a thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread</td>
</tr>
<tr>
<td>pthread_self</td>
<td>Find out own thread ID</td>
</tr>
</tbody>
</table>

- Return 0 if successful
• Automatically makes the thread runnable without a start operation
• Takes 3 parameters:
  – Points to ID of newly created thread
  – Attributes for the thread
    – Stack size, scheduling information, etc.
  – Name of function that the thread calls when it begins execution with argument

/* create the thread */
pthread_create(&tid, &attr, runner, argv[1]);
• `pthread_detach()`
  – Sets internal options to specify that storage for thread can be reclaimed when it exits
  – 1 parameter: Thread ID of the thread to detach

• Undetached threads don’t release resources until
  – Another thread calls `pthread_join` for them
  – Process exits

• `pthread_join`
  – Takes ID of the thread to wait for
  – Suspends calling thread till target terminates
  – Similar to `waitpid` at the process level

`pthread_join(tid, NULL);`
• If a process calls exit, **all** threads terminate
• Call to pthread_exit causes only the calling thread to terminate

```c
pthread_exit(0)
```
• Threads can force other threads to return through a **cancellation** mechanism
  – pthread_cancel: takes thread ID of target
  – Depends on *type* and *state* of thread
Pthreads Example (next 2 slides)

- This process will have two threads
  - Initial/main thread to execute the main() function. It creates a new thread and waits for it to finish.
  - A new thread that runs function runner()
    - It will get a parameter, an integer, and will compute the sum of all integers from 1 to that number.
    - New thread leaves the result in a global variable sum.
  - The main thread prints the result.
#include <pthread.h>
#include <stdio.h>

int sum; /* this global data is shared by the thread(s) */

void *runner(void *param); /* the thread */

int main(int argc, char *argv[ ])
{

pthread_t tid; /* the thread identifier */
pthread_attr_t attr; /* set of attributes for the thread */

if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    /*exit(1);*/
    return -1;
}

if (atoi(argv[1]) < 0) {
    fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
    /*exit(1);*/
    return -1;
}
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid, &attr, runner, argv[1]);
/* now wait for the thread to exit */
pthread_join(tid, NULL);

printf("sum = %d\n", sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
int i, upper = atoi(param);
sum = 0;
if (upper > 0) {
    for (i = 1; i <= upper; i++)
        sum += i;
}
pthread_exit(0);
Pthreads Code for Joining 10 Threads

```c
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);

#define MAX_THREAD 4
```
Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
    - Override its run() method
  - More commonly, implementing the Runnable interface

Runnable interface is defined by

```java
public interface Runnable {
    public abstract void run();
}
```

- More commonly, implementing the Runnable interface

1. Has 1 method `run()`
2. Create `new Thread` class by passing a Runnable object to its constructor
3. `start()` method creates a new thread by calling the `run()` method.

- new features available in `java.util.concurrent` package
Java Thread States

New Thread() → New → Runnable

New Thread() → New

Runnable → Start() → Running

Runnable → run() → Waiting

Runnable → End of execution → Dead

Runnable → Sleep(), wait() → Waiting

Dead → End of execution
Java version of a multithreaded program that computes summation of a non-negative integer.
This program creates a separate thread by implementing the Runnable interface.

class Sum
{
    private int sum;

    public int get() {
        return sum;
    }

    public void set(int sum) {
        this.sum = sum;
    }
}

Structure
class Sum
    public void setSum( int sum) { ..
Class summation implements runnable
    public void run() { ..
Public class Driver
{
    public static void main(String[] args) {
        Thread worker = new Thread(new summation( ...
        worker.start();
        try {
            worker.join(); ....
        }
    }
class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        if (upper < 0)
            throw new IllegalArgumentException();

        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;

        for (int i = 0; i <= upper; i++)
            sum += i;

        sumValue.set(sum);
    }
}
public class Driver {
    public static void main(String[ ] args) {
        if (args.length != 1) {
            System.err.println("Usage Driver <integer>");
            System.exit(0);
        }
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread worker = new Thread(new Summation(upper, sumObject));
        worker.start();
        try {
            worker.join();
        } catch (InterruptedException ie) {
        }
        System.out.println("The sum of " + upper + " is " + sumObject.get());
    }
}

A call to run( )
Help Session

- Help Session Thursday at 5 PM (redo)
- File I/O, Fork(), Exec(), System() etc.
- Needed for HW2 now available.
- CSB 130
Implicit Threading

• Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads

• Creation and management of threads done by compilers and run-time libraries rather than programmers

• Three methods explored
  – Thread Pools
  – OpenMP
  – Grand Central Dispatch

• Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package
Implicit Threading 1: Thread Pools

- Create a number of threads in a pool where they await work.
- **Advantages:**
  - Usually slightly faster to service a request with an existing thread than create a new thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool.
  - Separating task to be performed from mechanics of creating task allows different strategies for running tasks.
    - i.e. Tasks could be scheduled to run periodically.
- Windows API supports thread pools.
Implicit Threading2: OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```c
#pragma omp parallel
Create as many threads as there are cores
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
Run for loop in parallel
```

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */
    return 0;
}
```

Compile using
`gcc -fopenmp openmp.c`
Implicit Threading#: Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }” - 
  
```
  ^{ printf("I am a block")}; 
```
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads
Signal Handling

• **Signals** are used in UNIX systems to notify a process that a particular event has occurred.

• A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined

• Every signal has **default handler** that kernel runs when handling signal
  – **User-defined signal handler** can override default
  – For single-threaded, signal delivered to process
Where should a signal be delivered for multi-threaded?

- Deliver the signal to the thread to which the signal applies?
- Deliver the signal to every thread in the process?
- Deliver the signal to certain threads in the process?
- Assign a specific thread to receive all signals for the process?
Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

- Pthread code to create and cancel a thread:

  ```c
  pthread_t tid;

  /* create the thread */
  pthread_create(&tid, 0, worker, NULL);

  ...

  /* cancel the thread */
  pthread_cancel(tid);
  ```
**Thread Cancellation (Cont.)**

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

<table>
<thead>
<tr>
<th>Mode</th>
<th>State</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Disabled</td>
<td>-</td>
</tr>
<tr>
<td>Deferred</td>
<td>Enabled</td>
<td>Deferred</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Enabled</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches *cancellation point*
    - i.e. `pthread_testcancel()`
    - Then *cleanup handler* is invoked
- On Linux systems, thread cancellation is handled through signals
Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
  - Ex: Each transaction has a thread and a transaction identifier is needed.
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **static data**
  - TLS is unique to each thread
Is complexity always good?

• Is something that is
  – More advanced
  – More complex

Generally better?
• “Hyper-threading”: simultaneous multithreading:
  – Hardware support for multiple threads in the same core (CPU)

• Performance:
  – performance improvements are very application-dependent
  – Higher energy consumption
  – Not better than out-of-order execution
  – Intel has dropped it
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2018
Scheduling

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
• Compare:
  – Calling a function
  – Starting a child
  – Creating a thread
Diagram of Process State

Ready to Running: scheduled by scheduler
Running to Ready: scheduler picks another process, back in ready queue
Running to Waiting (Blocked): process blocks for input/output
Waiting to Ready: Input available
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern
Histogram of CPU-burst Times
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- **CPU scheduling decisions may take place when a process:**
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive. These need to be considered**
  - access to shared data by multiple processes
  - preemption while in kernel mode
  - interrupts occurring during crucial OS activities

Not forced
Dispatcher

• Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  – switching context
  – switching to user mode
  – jumping to the proper location in the user program to restart that program

• **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible: Maximize
- **Throughput** – # of processes that complete their execution per time unit: Maximize
- **Turnaround time** – time to execute a process from submission to completion: Minimize
- **Waiting time** – amount of time a process has been waiting in the ready queue: Minimize
- **Response time** – time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment): Minimize
Terms for a single process

- Command arrives
- Command begins running
- The first output of command appears
- Command finishes executing

- Wait time
- Response time
- Execution time
- Turnaround time
We will now examine several major scheduling approaches. Decides which process in the ready queue is allocated the CPU. Could be preemptive or nonpreemptive — preemptive: remove in middle of execution. Optimize measure of interest — We will use Gantt charts to illustrate schedules — Bar chart with start and finish times for processes.