Threads & Synchronization

CSC209H5: Software Tools & Systems Programming

Robert (Rupert) Wu
rupert.wu@utoronto.ca

Department of Computer Science
University of Toronto

April 3, 2023

Lesson Plan

1. Logistics: Lecture, Assignment 4, Final Exam
2. Parallel Computing
3. Introduction to Threads & Using `pthread.h`
   - Creating, terminating, joining, detaching.
   - Start functions and arguments.
4. Safety & Synchronization
5. Review of Course Material
6. Q&A and Farewell

Acknowledgements

These slides are mostly original, with some influence by Karen Reid and Andi Bergen.

Section 1

Logistics: Lecture, A4, Exam

Logistics: Lecture & Assignment 4

This week will be a little different...

LEC0101: Multithreading & Synchronization

We will primarily discuss parallel computing, multithreading, and synchronization.

- This material won't be on the CSC209 exam but you're strongly encouraged to attend (or learn after) whether or not you plan on taking CSC367/369 later.
- Whatever remaining time will be course review (mostly on Weeks 7-11).
- Students in LEC0102/0103/0104 are welcome to attend!

LEC0102/0103/0104: A4 & Review

Arnold/Bahar will go over Weeks 7-11 to assist with Assignment 4 and the exam.

- Includes going over old test and other practice questions.
- There will likely be some Q&A about A4 itself.
- Students in LEC0101 are encouraged to also attend if they need help with A4.

Logistics: Final Exam

The exam will cover all lectures, labs, and assignments.

- Date: Wednesday, April 19th
- Time: 13:00 (1PM) Eastern/Toronto
- Room: Gym A/B (Davis/RAWC)

Autofail policy

In order to pass this course, you must earn at least 40% on the final exam.

Section 2

Parallel Computing

Parallelism: Multiprocessing

Why do we use multiple processes? There's a limit to what single-processing (and single-core technology) can do. We want:

- Isolation: each process and its data can be protected from the others.
- Parallelism: if our computer has multiple processors, then each processor can be running a process!

Ultimately, multiprocessing takes advantage of the fact that many things can be done at the same time.

Parallelism: Types of Parallel Computing

There are several granularities to which we might apply parallelism.

1. Bit-level: operate on many bits in parallel.
2. Instruction-level: operate with many instructions in parallel.
3. Data-level: read/write several chunks of data in parallel.
4. Task-level: perform many higher-level tasks/programs in parallel.
5. Machine-level: federate or decentralize tasks across several machines/nodes such as servers and clients.

Which types are the result of multiprocessing via `fork()`?

- `fork()` typically resembles task-level parallelism.

What about `fork()`+`select()`?

- When `select()` is used for child process that read from files/pipes/sockets, there is some data-level parallelism.

Can you think of examples for the other types?

Parallelism: Synchronicity

Parallel solutions might require some coordination or synchronization if there exist inter-process dependencies. Otherwise, asynchronous parallelism might be possible.

1. Does multiprocessing via `fork()` resemble in synchronous or asynchronous parallelism?
   - By default child process as result of `fork()` are asynchronously parallel.
2. What happens when we invoke a blocking call like `wait()` or `read()`?
   - Processes are synchronized at the blocking call.
3. Why asynchronous parallelism?
   - When processes don't depend on each other, asynchronous parallelism allows them to make more progress or "throughput".

Section 3

Threads & `pthread.h`

Processes: The Anatomy

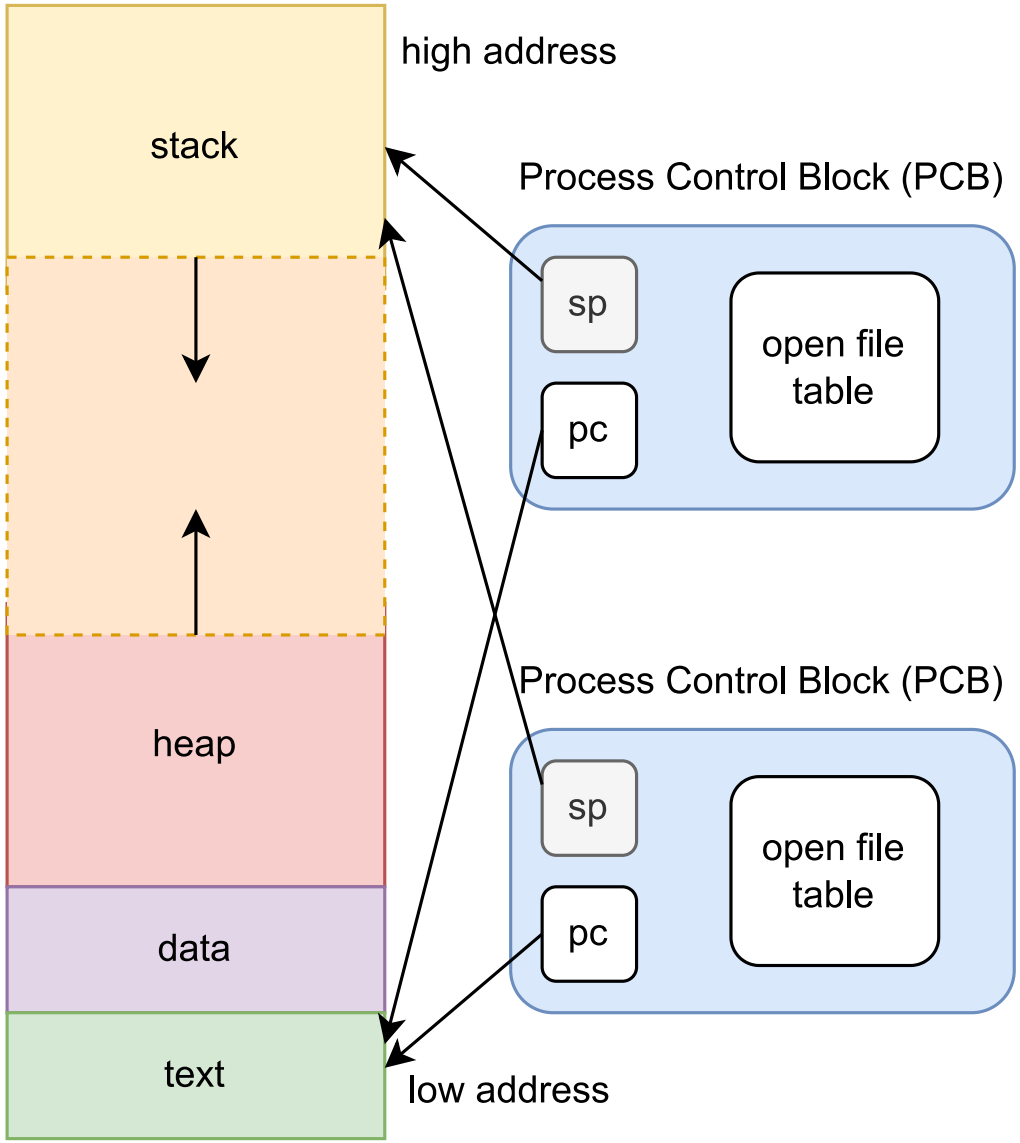

Figure 1: processes might share open files and/or pipes

Processes: The Downsides

Were processes not good enough?

- `fork()` is a heavyweight system call.
- Each process gets its own memory address space.
- Context-switching and synchronizing between processes are expensive.
- Communication between processes requires external kernel constructs (pipes, sockets, signals, etc.) which are also expensive.

So, what if we had multiple *threads* of execution, all within a single process?

Threads: An Introduction

Threads belong to a process and share the same PID and parent PID.

- Technically each process by default is running on a single thread.
- Threads of a process also share the heap and global variables.
- Each thread also has unique attributes:
  - Its own thread ID;
  - Its own `errno` variable;
  - Its own stack for local variables and function calls.

Advantages of threads

- Communication between threads is cheap; they can share variables!
- Threads are "lightweight"; creation and switching are faster.
- Synchronization avoids the kernel.

Specs of a CPU

The number of threads "on a CPU" refers to the number of high-level tasks that your CPU can do at a time. Use `htop` to monitor the compute load on each.

Threads: The Anatomy

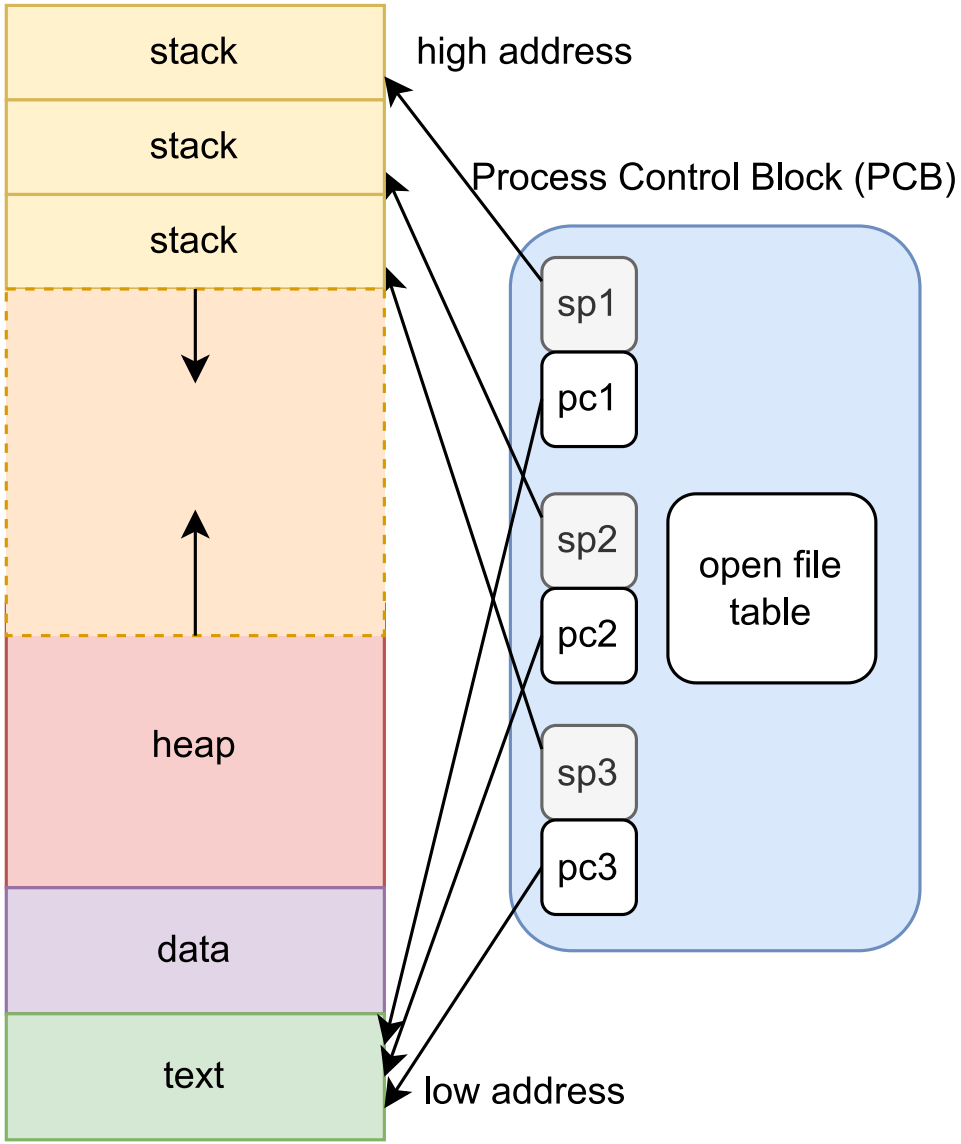

Figure 3: threads share an address space and open files

Multiprocessing vs. Multithreading

Both processes and threads perform multiple tasks concurrently. Here's how...

Attribute	Processes	Threads
Program Counter	Individual	Individual
Stack (pointer)	Individual	Individual
Memory	Individual	Shared
Address Space (code)	Individual	Shared
Open Files (FDs)	Shared	Shared
Pipes & Sockets	Individual or Shared	Shared

The stack and heap addresses are shared by threads of the same process, but each thread gets its own "mini-stack" within the stack space of that process.

Which should you use?

Depends what tasks you're performing and their complexity and dependencies.

- Processes can do some things that threads (which are simpler) can't.
- Sometimes threads can perform the same operations more efficiently.

Threads: Package pthread.h

POSIX threads (pthread's) is the most commonly used thread package on UNIX systems (unavailable on Microsoft Windows). Just compile with the -pthread flag.

```
#include "pthread.h"
#include "sys/types.h" // provides the *_t types
/* this is just a subset of functions */
int       pthread_cancel(pthread_t tid);
int       pthread_create(pthread_t *restrict tid,
                         const pthread_attr_t *restrict attr,
                         void *(*start_routine)(void *),
                         void *restrict arg);
int       pthread_detach(pthread_t tid);
void      pthread_exit(void *retval);
int       pthread_join(pthread_t tid, void **retval);
pthread_t pthread_self(void);
```

Alternatives in C++

pthread is a sensible standard and is rather feature-rich. But others exist:

- std::thread: pretty standard but requires C++11. Only basic features are portable.
- boost::thread: cross-platform but has external dependency.

Threads: Creation

Within a process, you can create a new thread with `pthread_create()`.

```
#include "pthread.h"
int pthread_create(pthread_t *restrict tid,
                   const pthread_attr_t *restrict attr,
                   void *(*start)(void *),
                   void *restrict arg);
```

- `tid` uniquely identifies a thread within a process and is returned by the function.
- `attr` specifies attributes, as priority, initial stack size
  - use `NULL` for defaults.
- `start` is a pointer to a function that starts the thread.
  - `arg` (usually heap/global) is the argument to `start`.

Error handling

`pthread_create()` doesn't set `errno` but returns compatible error codes. You can use 'stderr() to print error messages.

Threads: Termination

A thread typically terminates when its `start()` function returns or any thread in the process calls `exit()`?

If that's the case, calling `exit()` from any thread will terminate all threads in the process. What if we just want the thread to terminate? You'd call `pthread_exit()`.

```
#include "pthread.h"
void pthread_exit(void *retval);
int  pthread_cancel(pthread_t tid);
```

Another way for a thread to terminate is being cancelled by another thread using `pthread_cancel()`.

A refresher on `(void *)`

What does `(void *)` mean? It's a flexible pointer to any arbitrary data structure.

- To use it as any data type, you'd cast it to some compatible type. Otherwise, you might undetermined/unexpected behaviour, or a segmentation fault.

Threads: Joining

By default, a thread is *joinable*: it can be enjoined by another thread. Similarly to using `waitpid()` for child processes, `pthread_join()` waits for a thread to terminate, or returns immediately if the thread has already terminated.

```
#include "pthread.h"
int pthread_join(pthread_t tid, void **retval);
```

- `tid`: ID of the thread to wait for.
- `retval`: if not `NULL`, stores the `(void *)` pointer returned by thread `tid` upon termination.

Comparison to `waitpid()`

Both `wait()`/`waitpid()` and `pthread_join()` are blocking. But...

- There is no thread hierarchy. Any thread can use `pthread_join()` to wait for any other thread in the same process.
- There is no way to "join with any thread", i.e. no analogy to `wait()`.
- There is no equivalent to `WNOHANG`.

Threads: Detaching

You can also detach a *joinable* thread `tid` such that it'll be a non-joinable daemon thread. A "detached" thread will run as normal, but it'll release its resources to the system automatically without any thread calling `pthread_join()`.

- Trying to detach a *non-joinable* thread results in `EINVAL`.

```
#include "pthread.h"
int pthread_detach(pthread_t tid);
pthread_t pthread_self(void)
```

You can use `pthread_self(void)` to find the "parent" (there is no explicit hierarchy) thread, i.e. the one that created the current one.

```
pthread_detach(pthread_self(void)); // frequent use-case
```

Threads: Inner Function Arguments

We can pass any variable (including a structure or array) to our thread function.

```
pthread_t thread_ID; int fd, result;
fd = open("afile", O_RDONLY);
result = pthread_create(&thread_ID, NULL, myThreadFcn, (void *)&fd);
if (result != 0)
    printf("Error: %s\n", strerror(result));
```

- It assumes the thread function knows what type it is and will cast it therein.
- This example is *bad* if the main thread alters `fd` later.

A better solution might include `malloc()` to create memory for the variable.

1. Initialize variable's value.
2. Pass pointer to new memory via `pthread_create()`.
3. Thread function `myThreadFcn()` releases memory when done.

Threads: Inner Function Arguments

A better solution might include `malloc()` to create memory for the variable.

```
// 0. define struct for the variable
typedef struct myArg { int fd; char name[25]; } MyArg;

// 1. initialize variable's value
MyArg *p = (MyArg *)malloc(sizeof(MyArg));
p->fd = fd; // assumes fd is defined
strncpy(p->name, "CSC209", 7);

// 2. passes pointer to new memory
pthread_t thread_ID;
int result = pthread_create(&thread_ID, NULL, myThreadFunc, (void *)p);

void *myThreadFunc(void *p) {
    MyArg *theArg = (MyArg *) p;
    write(theArg->fd, theArg->name, 7);
    close(theArg->fd); free(theArg);
    return NULL;  // 3. releases memory when done
}
```

Threads: Program Models

Several common models for threaded programs exist:

- Manager/worker: a single manager thread assigns work to other threads, the workers. The manager typically handles all input and parcels out work to the workers.
- Pipeline: a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. Is like an automobile assembly line.
- Peer: similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

What kinds of parallelism are exhibited in these models?

Threads: Disadvantages & Safety

We must be careful when multithreading.

- A bug in one thread can damage other threads.
- It's difficult to use signals with threads.
- All threads in a process must run the same program.
- Only thread-safe functions can be used.

Thread-Safety

Due to global/static variables and heap memory, not all functions are thread-safe.

Unsafe	Safe
`ctime()`	`ctime_r()`
`gmtime()`	`gmtime_r()`
`gethost*()`	
`localtime()`	`localtime_r()`
`rand()`	`rand_r()`
`strtok()`	`strtok_r()`

Notice a pattern in these function names? The `_r` stands for "re-entrant".

Section 4

Safety & Synchronization

Exploiting Parallelism with Multithreading

Threads can easily share information using global variables. Consider this example.

```
int S = 0, N = 1073741824;
void *increment_by_one(void *_) {
    for (int j = 0; j < N; j++) { long local = S; local++; S = local; }
}
int main() { increment_by_one(NULL); }
```

If we're impatient, we can dispatch T threads to compute the sum in parallel.

```
int S = 0, N = 1073741824;
int main(int argc, char **argv) {
    int T = atoi(argv[1]); N /= T;
    pthread_t TIDs[T];
    for (int t = 0; t < T; t++)
        pthread_create(&TIDs[t], NULL, increment_by_one, NULL);
    for (int t = 0; t < T; t++) pthread_join(TIDs[t], NULL);
    return 0;
}
```

We are calling `pthread_join()`, so this is reliable and consistent, right?

Safety: Exploiting Parallelism is Risky!

No! Absolutely not. Consider many threads are probably running `increment_by_one()` at once, accessing `S` concurrently. In other words, `S` can have multiple "writers" and "readers" at the same time.

```
void *increment_by_one(void *_) {
    for (int j = 0; j < N; j++) {
        long local = S;  // read from S
        local++;
        S = local;       // write to S
    }
}
```

Here's a possible (problematic) execution path if there are `T=2` threads:

1. Thread 1 increments `S` 2000 times.
2. On iteration `j=2001`, it obtains the value of `S`, but then...
3. Thread 2 increments `S` 536870912 times, and terminates.
4. Now thread 1 takes over, but writes 2001 into `S`!

`S` won't always be the expected result 1073741824; you can imagine it gets worse with more threads! If the behaviour is inconsistent/unpredictable, we call this a *race condition*.

Safety: Critical Sections

We refer to the reading and writing of a shared resource as a *critical section*. In a perfect world, threads would not interfere with each other in these. One can characterize these phases of code as those with variables that can have at most one writer, or only readers.

Can we solve this by reducing the number of times a critical section is run?

```
void *increment_by_part(void *_) {
    long partial = 0;
    for (int j = 0; j < N; j++)
        partial += 1;
    S += partial;
}
```

It certainly reduces the number of times the critical section runs, but increases the odds that each one is compromised owing to the longer run-time of the for loop. Also, the consequences of a single interference can be worse!

Synchronization: Locks & Mutexes

```c
#include "pthread.h"
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

A *mutual exclusion (mutex)* ensures only one thread can access a variable at a time.

- A mutex is always in one of two states: locked or unlocked.
- When unlocked, any thread can lock (often phrase "acquire lock on") the mutex.
- Any thread that tries to acquire a locked mutex is blocked until it's unlocked.
- Only the thread that locked the mutex is allowed to unlock it.

Since threads share memory, their process's mutex `mtx` is common between them.

```c
pthread_mutex_t mtx; // can be global or local
int status = pthread_mutex_init(&mtx, NULL);
...
pthread_mutex_lock(&mtx);    // if acquired by thread x, thread y is blocked
...                          // access shared resource
pthread_mutex_unlock(&mtx); // x releases mtx, y can acquire as above
```

Synchronization: Locks & Mutexes

The terms "mutex" and "lock" are often used interchangeably. Whether a mutex/lock `mtx` applies across processes (mutex) or just between threads of the same process (lock) depends if `mtx` is global or local.

```c
// mtx can be a global mutex
void *increment_by_one(void *_) {
    for (int j = 0; j < N; j++) {
        pthread_mutex_lock(&mtx);
        long local = S;
        local++;
        S = local;
        pthread_mutex_unlock(&mtx);
    }
} // which function does it better?
```

```c
// or a local lock
void *increment_by_part(void *_) {
    long local = 0;
    for (int j = 0; j < N; j++)
     local += 1;
    pthread_mutex_lock(&mtx);
 S += local;
    pthread_mutex_unlock(&mtx);
}
// look at critical sections
```

Here, we can use either because we don't `fork()` any new processes (so all threads belong to the same process) or have other functions taking up stack space.

Full code

`github.com/rhubarbwu/csc209/blob/master/lectures/lec12/counting.c`

Synchronization: Locks & Mutexes (Example with Macros)

To write versions of your code with/out mutexes, you might wrap (macro-)conditional directives `#ifdef`/`#else`/`#endif` around your mutex-related code.

```
void *increment_by_part(void *_) {
    long local = 0;
    for (int j = 0; j < N; j++) local += 1;
#ifdef MUTEX
    pthread_mutex_lock(&mtx);
#endif
    S += local;
#ifdef MUTEX
    pthread_mutex_unlock(&mtx);
#endif
}
```

Then compile with the `MUTEX` macro with the `-D` compiler flag to use mutexes.

```
$ wget raw.githubusercontent.com/rhubarbwu/csc209/master/lectures/lec12/counting.c
$ gcc -pthread -o counting counting.c          # without mutexes
$ gcc -DMUTEX -pthread -o counting counting.c  # with mutexes
```

See if there are any changes in accuracy or efficiency...

Synchronization: But Wait! There's more!

Figure 4: finding out there's more than mutexes

Synchronization: Conditional Variables (CVs)

A *condition variable* (CV) allows threads to notify others of changes in resources. A CV also allows threads to block waiting for such notification. Without a CV, threaded programs can be inefficient (e.g. looping to poll a variable).

CVs are essentially mutexes with wait-queues and have three core operations.

- `signal`: wakes up at least one thread waiting for the CV.
- `broadcast`: wakes up all threads waiting for the CV.
- `wait`: waits (blocks) until signaled by a CV.

CVs and mutexes

A CV has an associated mutex, which must be locked by a thread before it calls `pthread_cond_wait()`.

- `pthread_cond_wait()` unlocks the mutex, blocks the thread, and (when the thread is later signaled) relocks the mutex.
- Unlocking the mutex and blocking the thread are *atomic*: no other thread can `signal` the CV between these two operations.

Synchronization: And more...

Figure 5: finding out there's more than mutexes and CVs

Synchronization: Semaphores

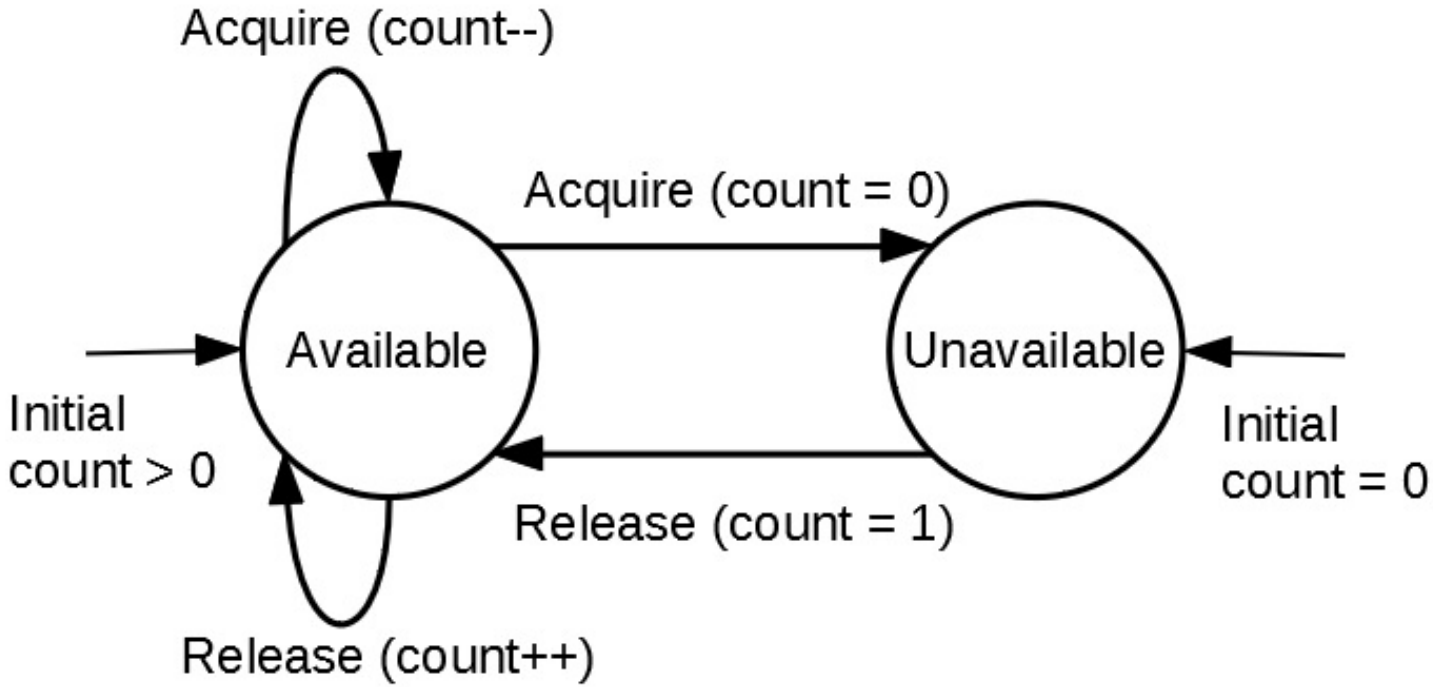

Figure 6: `open4tech.com/rtos-mutex-and-semaphore-basics/`

Semaphores are like CVs with a counter. In other words, instead of a binary locking status, a semaphore tracks some counter variable, essentially a multiplexing lock.

- Often tracks how many threads/processes are currently in a critical section.
- But it could be used to represent anything else the programmer wants.

Synchronization: Okay Stop!

Synchronization: Not a Silver Bullet do (Synchronized) Threads Make

Ideally, we synchronize our process/threads across their shared resources properly. Then, multithreading is useful but you can have too much of a good thing.

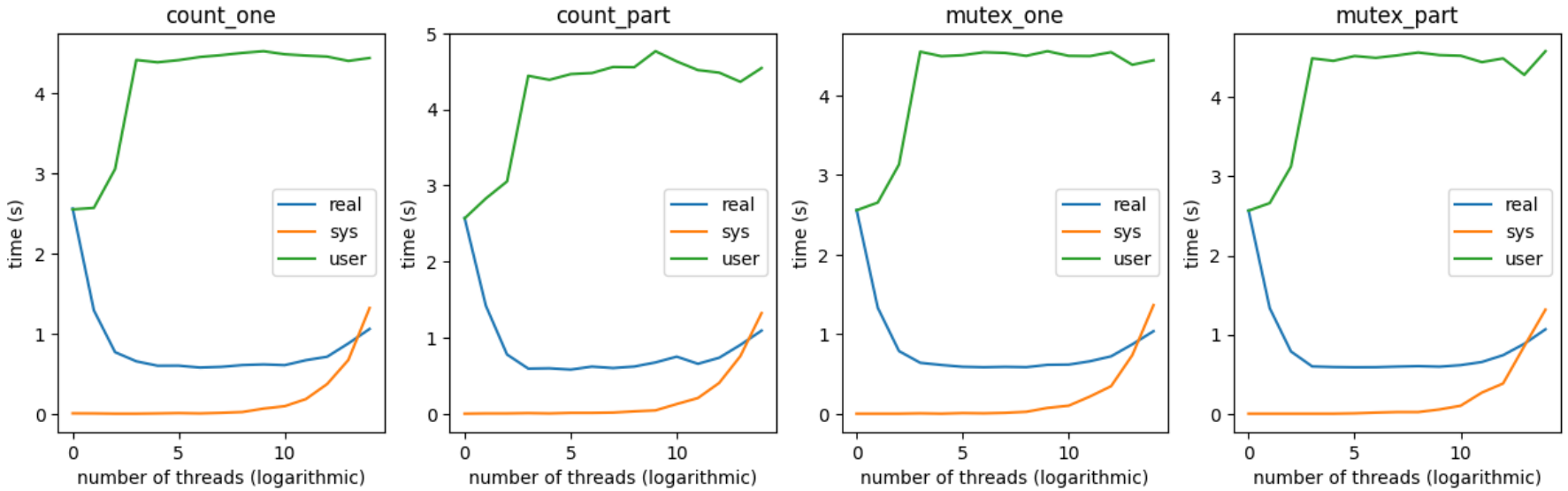

Figure 7: output of `time` from `bash`, with respect to (logarithmic) number of threads; notice that real/sys times eventually start to increase when there are too many threads.

Takeaway: Don't spin too many threads because they consume memory to maintain and can introduce synchronization latency, especially around mutexes/CVs/semaphores.

Code, scripts, and logs

`github.com/rhubarbwu/csc209/tree/master/lectures/lec12`

Parallel Computing: Beyond CSC209

Much parallel programming – CUDA, the cloud, ASICs, etc. – awaits! And development of new architectures, platforms, and algorithms is ongoing in both application and research!

CSC367: Parallel Programming

Topics include computer instruction execution, **instruction-level parallelism**, memory system performance, task and data **parallelism**, **parallel models** (**shared memory**, message passing), **synchronization**, scalability and Amdahl's law, Flynn taxonomy, vector processing and **parallel computing architectures**.

CSC369: Operating Systems

The operating system as a control program and as a resource allocator. **Processes and threads, concurrency (synchronization/mutual exclusion/deadlock)**, processor, scheduling, memory management, file systems, and protection.

CSC409: Scalable Computing

We investigate computation in the large [...] to solve complex problems involving big data, serving large collections of users, in high availability, global settings. [...] Topics include caching, load balancing, **parallel computing and models**, redundancy, failover strategies, use of **GPUs**, and noSQL databases.

ECE419: Distributed Computing

Heterogeneity, security, transparency, **concurrency**, fault-tolerance; networking principles; request-reply protocol; remote procedure calls; distributed objects; middleware architectures; CORBA; security and authentication protocols; distributed file systems; name services; **global states in distributed systems**; **coordination and agreement**; **transactions and concurrency control**; distributed transactions; replication.

Section 5

Review and Q&A

Review and Q&A: Course Material

Ask about any of the past lecture material!

1. UNIX & Shell Programming
2. More Shell, Commands & C
3. Arrays, Pointers, File I/O
4. Memory & Compilation
5. N-D Arrays, Dynamic Memory & Structs
6. Linked-Lists, Strings Manipulation, & Debugging
7. Low-Level I/O & Signals (Week 8 in LEC0104) *
8. Processses (Week 7 in LEC0104) *
9. Pipes in C *
10. Networks & Sockets *
11. Multiplexing I/O *
12. Threads & Synchronization

* important for A4

Webpage & Slides

`mcs.utm.utoronto.ca/~209/23s/lectures.shtml`

Q&A: Academics & Careers

Where do you go from here? Anywhere you like...

- Compilers & Interpreters: CSC488/2107/ECE467, CSCD70
- Computer Design & Architecture: ECE532, 552
- Computer Graphics: CSC317, 417, 419/2520
- Computer Networks: CSC358, 457, 458/2209
- Distributed Computing: ECE419, CSC2221
- Information Security: CSC333, 347, 423, 427/ECE568
- Microprocessors: CSC/ECE385
- Operating Systems: CSC369, 469/2208, 2227
- Parallel Programming: CSC367/ECE1747, CSC2224/ECE1755
- Robotics: CSC376, 476/2606, 477/2630
- Software Engineering: CSC301, 302/D01, 309, 409

Many potential career paths await you!

Last Reminders

A couple of last things...

Assignment 4 - Due April 7th

- Go to LEC0102/0103/0104 and office hours.
- Review the slides and readings.
- Use Google and StackOverflow (without cheating).
- Make Piazza questions public if possible.
  - You'll help each other more quickly.

Course Evaluations - Due April 11th

Please remember to do course evaluations! They help the instructors and the department improve our teaching and administration. Check your inbox/Quercus.

Final Exam - April 19th

And of course, study hard for your exam.

- Review readings, slides, exercises/assignments.
- Use Google, StackOverflow, Piazza if they help.

Farewell

Thank you for a great semester! :')