Concurrency

Kerrisk, Ch 29, 47, 48, 53, 54
Concurrency

• The two key concepts driving computer systems and applications are
  – communication: the conveying of information from one entity to another
  – concurrency: the sharing of resources in the same time frame

• Concurrency can exist in a single processor as well as in a multiprocessor system

• Managing concurrency is difficult, as execution behaviour is not always reproducible.
Concurrency Example

• Program a:
  ```bash
  #!/usr/bin/sh
  count=1
  while [ $count -le 20 ]
  do
    echo -n "a"
    count=`expr $count + 1`
  done
  ```

• Program b
  ```bash
  #!/usr/bin/sh
  count=1
  while [ $count -le 20 ]
  do
    echo -n "b"
    count=`expr $count + 1`
  done
  ```

• When run sequentially (a; b) output is sequential.
• When run concurrently (a&; b&) output is interspersed and different from run to run.
Race conditions

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- E.g., If any code after a fork depends on whether the parent or child runs first.
- A parent process can call wait() to wait for termination (may block)
- A child process can wait for parent to terminate by polling (wasteful) (How would you do this?)
- One standard solution is to use signals.
Example 1

Process A

\[ x = \text{get(count)} \]
\[ \text{write}(x + 1) \]

\[ x = 1 \]
\[ \text{write}(2) \]

Process B

\[ y = \text{get(count)} \]
\[ \text{write}(y + 1) \]

\[ y = 2 \]
\[ \text{write}(3) \]

The value of count is what we expect.
Example 2

Process A

\[ x = \text{get(count)} \]
\[ \text{write}(x + 1) \]

\[ x = 1 \]

\[ \text{write}(2) \]

Process B

\[ y = \text{get(count)} \]
\[ \text{write}(y + 1) \]

\[ y = 1 \]
\[ \text{write}(2) \]
\[ y = 2 \]
\[ \text{write}(3) \]

Count

1
2
3
2

Not what we wanted!
Example: Race Conditions

```
#!/bin/sh

c=1

while [ $c -le 10 ]
do
  sd=`cat sharedData`
  sd=`expr $sd + 1`
  echo $sd > sharedData
  c=`expr $c + 1`
  echo d = $sd

done

#file sharedData must exist and hold
#one integer
```
Producer/Consumer Problem

- Simple example: `who | wc -l`
- Both the writing process (`who`) and the reading process (`wc`) of a pipeline execute concurrently.
- A pipe is usually implemented as an internal OS buffer.
- It is a resource that is concurrently accessed by the reader and the writer, so it must be managed carefully.
Producer/Consumer

- **consumer** should be blocked when buffer is empty
- **producer** should be blocked when buffer is full
- producer and consumer should run independently as far as buffer capacity and contents permit
- producer and consumer should never be updating the buffer at the same instant (otherwise data integrity cannot be guaranteed)
- producer/consumer is a harder problem if there are more than one consumer and/or more than one producer.
Threads
Motivation

• Processes are expensive to create.
• It takes quite a bit of time to switch between processes.
• Communication between processes must be done through an external structure – files, pipes, shared memory.
• Synchronizing between processes is cumbersome.
• *Is there another model that will solve these problems?*
Processes

- Each process has its own
  - program counter
  - stack
  - stack pointer
  - address space
- Processes may share
  - open files
  - pipes
Threads

- Each thread has its own
  - program counter
  - stack
  - stack pointer

- Threads share
  - address space
    - variables
    - code
  - open files
What is a process?

- OS abstraction for execution
- Running instance of a program

Components of a process:
- Address space
- Code and data
- Stack
- Program Counter (PC)
- Set of registers
- Set of OS resources: open files, network connections...
Rethinking Processes

• What is similar in cooperating processes?
  – They all share the same code and data (address space)
  – They all share the same privileges
  – They all share the same resources (files, sockets, etc.)

• What don’t they share?
  – Each has its own execution state: PC, SP, and registers

• Key idea: Why don’t we separate the concept of a process from its execution state?
  – Process: address space, privileges, resources, etc.
  – Execution state: PC, SP, registers

• Exec state also called thread of control, or thread
What is a Thread?

• A thread is a single control flow through a program
  – What is a “control flow”?
  – How is control flow represented?

• A program with multiple control flows is multithreaded
Control Flow

• **Control** includes all of the values that select which instructions in a program are executed.
• **Control flow**, then, is the sequence of instructions being executed.
• The hardware uses the program counter (PC) and stack to make control flow decisions.
Advantages

• Communication between threads is cheap
  – they can share variables!

• Threads are “lightweight”
  – faster to create
  – faster to switch between
Pthreads

• POSIX threads (pthreads) is the most commonly used thread package on Unix/Linux
**pthread_create**

```c
int pthread_create(pthread_t *tid,
                   pthread_attr_t *attr,
                   void *(*func)(void*), void *arg);
```

- `tid` uniquely identifies a thread within a process and is returned by the function.
- `attr` sets attributes such as priority, initial stack size – can be specified as NULL to get defaults.
- `func` - the function to call to start the thread – accepts one void *argument, returns void *.
- `arg` is the argument to `func`.
- returns 0 if successful, a positive error code if not.
- does not set `errno` but returns compatible error codes.
- can use `strerror()` to print error messages.
int pthread_join(pthread_t tid,  
     void **status)

• tid - the tid of the thread to wait for
  – cannot wait for any thread (as in wait())
• status, if not NULL returns the void * returned by the thread when it terminates.
• a thread can terminate by
  – returning from func
  – the main() function exiting
  – pthread_exit()
More functions

- **void pthread_exit(void *status)**
  - a second way to exit, returns status explicitly
  - status must not point to an object local to the thread, as these disappear when the thread terminates.

- **int pthread_detach(pthread_t);**
  - if a thread is detached its termination cannot be tracked with pthread_join()
  - it becomes a daemon thread

- **pthread_t pthread_self(void)**
  - returns the thread ID of the thread which called it
  - often see pthread_detach(pthread_self())
Passing Arguments to Threads

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

- We can pass any variable (including a structure or array) to our thread function.
- It assumes the thread function knows what type it is.
- This example is **bad** if the main thread alters fd later.
Solution

• Use malloc() to create memory for the variable
  – initialize variable’s value
  – pass pointer to new memory via pthread_create()
  – thread function releases memory when done.

• Example:

```c
typedef struct myArg {
    int fd;
    char name[25];
} MyArg;

int result;
pthread_t thread_ID;
```
Example (cont’d)

MyArg *p = (MyArg *)malloc(sizeof(MyArg));
p->fd = fd; /* assumes fd is defined */
strncpy(p->name, "CSC209", 7);
result = pthread_create(&threadID, NULL,
                        myThreadFcn, (void *)p);
void *myThreadFcn(void *p) {
    MyArg *theArg = (MyArg *) p;
    write(theArg->fd, theArg->name, 7);
    close(theArg->fd);
    free(theArg);
    return NULL;
Thread-safe functions

• Not all functions can be called from threads
  – many use global/static variables
  – new versions of UNIX have thread-safe replacements like strtok_r()

• Safe:
  – ctime_r(), gmtime_r(), localtime_r(), rand_r(), strtok_r()

• Not Safe:
  – ctime(), gmtime(), localtime(), rand(), strtok(), gethostxxx()

• Could use semaphores to protect access but will generally result in poor performance.
Pthread Mutexes (Semaphores)

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);

• easier to use than semget() and semop()
• only the thread that locks a mutex can unlock it
• mutexes often declared as globals
Example

```c
pthread_mutex_t myMutex;
int status;

status = pthread_mutex_init(&myMutex, NULL);
if(status != 0)
    printf("Error: %s \n", strerror(status));
pthread_mutex_lock(&myMutex);
/* critical section here */
pthread_mutex_unlock(&myMutex);
status = pthread_mutex_destroy(&myMutex);
if(status != 0)
    printf("Error: %s\n", strerror(status));
```