CSCB09 Software Tools and Systems Programming

IO Multiplexing // Threads

Marcelo Ponce

Winter 2023

Department of Computer and Mathematical Sciences - UTSC

Today's class

Today we will discuss the following topics:

- IO Multiplexing
- Threads

IO Multiplexing

- One basic concept of Linux systems is the rule that everything in Unix/Linux is a file.
- Each process has a table of file descriptors that point to files, sockets, devices and other operating system objects.
- Not so surprising then is the fact, that processes will have to deal with a large number of files, i.e. FDs.
- A simple solution would be to create a process (or thread) to deal with them...

- One basic concept of Linux systems is the rule that everything in Unix/Linux is a file.
- Each process has a table of file descriptors that point to files, sockets, devices and other operating system objects.
- Not so surprising then is the fact, that processes will have to deal with a large number of files, i.e. FDs.
- A simple solution would be to create a process (or thread) to deal with them... For small numbers of FDs this may work, but for large ones is a bad idea!

- One basic concept of Linux systems is the rule that everything in Unix/Linux is a file.
- Each process has a table of file descriptors that point to files, sockets, devices and other operating system objects.
- Not so surprising then is the fact, that processes will have to deal with a large number of files, i.e. FDs.
- A simple solution would be to create a process (or thread) to deal with them... For small numbers of FDs this may work, but for large ones is a bad idea!

The solution to this, is to use a kernel mechanism for *polling* –check the status– over a set of FDs:

Multiplexing
a system or signal involving simultaneous transmission of several messages along a single channel of communication.

- One basic concept of Linux systems is the rule that everything in Unix/Linux is a file.
- Each process has a table of file descriptors that point to files, sockets, devices and other operating system objects.
- Not so surprising then is the fact, that processes will have to deal with a large number of files, i.e. FDs.
- A simple solution would be to create a process (or thread) to deal with them... For small numbers of FDs this may work, but for large ones is a bad idea!

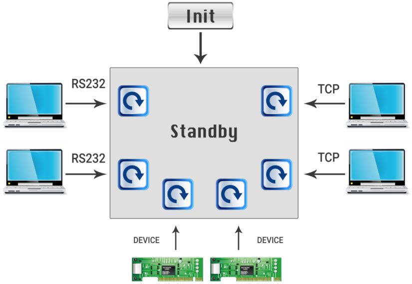
The solution to this, is to use a kernel mechanism for *polling* –check the status– over a set of FDs:

Multiplexing
a system or signal involving simultaneous transmission of several messages along a single channel of communication.

There are three main options in Linux: `select`, `poll`, `epoll`

- Typical system that works with many IO sources has an initialization phase and then enter some kind of standby mode – wait for any client to send request and response it
- A tentative "solution" is to create a thread (or process) for each client, block on read until a request is sent and write a response.
- This may work with a small amount of clients, but if we want to scale it to hundred of clients, creating a thread for each client is a bad idea

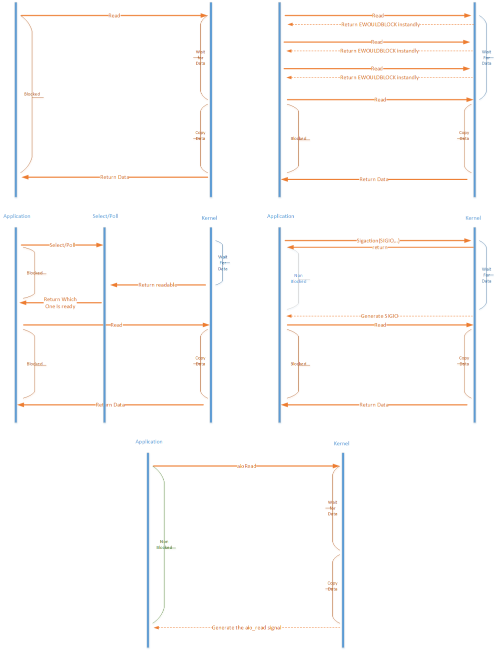
- Typical system that works with many IO sources has an initialization phase and then enter some kind of standby mode – wait for any client to send request and response it
- A tentative "solution" is to create a thread (or process) for each client, block on read until a request is sent and write a response.
- This may work with a small amount of clients, but if we want to scale it to hundred of clients, creating a thread for each client is a bad idea



The solution is to use a kernel mechanism for polling over a set of file descriptors.

I/O Models

- Blocking I/O
- Non-blocking I/O
- I/O Multiplexing (`select`/`poll`/...)
- Signal driven I/O (`SIGIO`)
- Asynchronous I/O



The `select()` system call provides a mechanism for implementing synchronous multiplexing I/O

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
    struct timeval *timeout);
```

A call to `select( )` will **block** until the given FDs are ready to perform I/O, or until an optionally specified timeout has elapsed.

`poll` system call

Unlike `select()`, with its inefficient three bitmask-based sets of file descriptors, `poll()` employs a single array of `nfds` `pollfd` structures

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

```
struct pollfd {
      int fd;
      short events;
      short revents;
};
```

`epoll` system calls

`epoll*` system calls help us to create and manage the context in the kernel

- create a context in the kernel using `epoll_create`
- add and remove file descriptors to/from the context using `epoll_ctl`
- wait for events in the context using `epoll_wait`

`select`

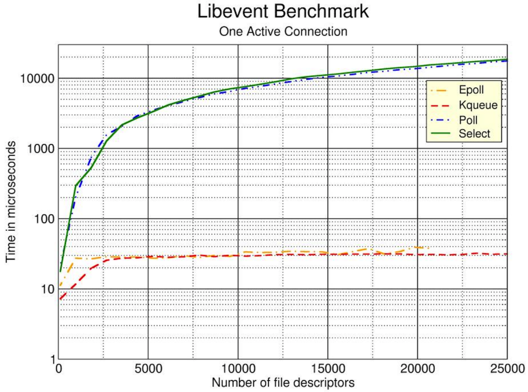
- FD sets statically sized
- FD sets reconstructed on return
- timeout parameter undefined on return
- more portable
- `pselect`

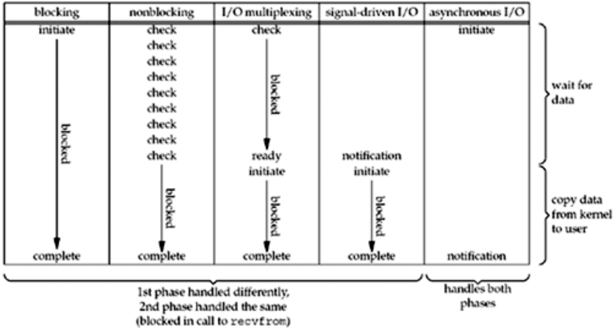
`poll`

- does not require user to calculate max nbr of FDs
- more efficient for large-valued FDs
- separates input (events) from the output (revents)
- more portable

`epoll`

- can add/remove FDs while waiting
- better performance: $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$
- can behave as level triggered or edge triggered
- linux specific, so non portable


Libevent Benchmark
One Active Connection
Time in microseconds
Number of file descriptors
Epoll
Kqueue
Poll
Select
1
10
100
1000
10000
0
5000
10000
15000
20000
25000


blocking
nonblocking
I/O multiplexing
signal-driven I/O
asynchronous I/O
initiate
check
check
check
check
check
check
check
check
check
blocked
ready
initiate
notification
initiate
complete
complete
complete
complete
notification
wait for data
copy data from kernel to user
1st phase handled differently, 2nd phase handled the same (blocked in call to recvfrom)
handles both phases

Concurrency

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

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

```
#!/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.

Race Condition – Example

Process A		*Process B*
`x = get(count)`		`y = get(count)`
`write(x+1)`	Count	`write(y+1)`
`x=1` $\leftarrow$	$\boxed{1}$	
`write(2)` $\rightarrow$	$\boxed{2}$	$\rightarrow$ `y=2`
	$\boxed{3}$	$\leftarrow$ `write(3)`
	$\checkmark$	

Race Condition – Example

Process A	Count	*Process B*
`x = get(count)`		`y = get(count)`
`write(x+1)`		`write(y+1)`
`x=1` ←	$\boxed{1}$	
`write(2)` →	$\boxed{2}$	→ `y=2`
	$\boxed{3}$	← `write(3)`
	✓	

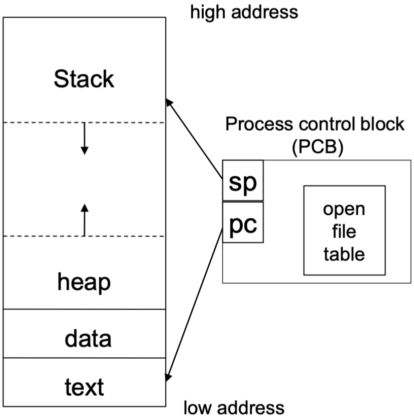
Process A	Count	*Process B*
`x = get(count)`		`y = get(count)`
`write(x+1)`		`write(y+1)`
`x=1` ←	$\boxed{1}$	→ `y=1`
	$\boxed{2}$	← `write(2)`
		→ `y=2`
	$\boxed{3}$	← `write(3)`
`write(2)` →	$\boxed{2}$	

Threads

- 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.

- 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

What is a process?

- OS abstraction for execution
- Running instance of a program

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.)

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?

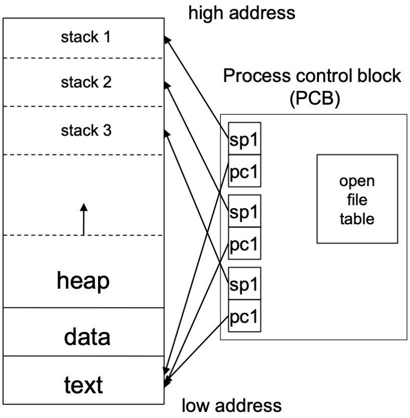
  - Each has its own execution state: PC, SP, and registers

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
- *Execution state* also called thread of control, or thread

Threads



- Each thread has its own
  - program counter
  - stack
  - stack pointer
- Threads share
  - address space
    - variables
    - code
  - open files

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**

What is a Thread?

Control Flow

- 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**

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.

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.

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.

Threads – Advantages

- Communication between threads is cheap
  - they can share variables!
- Threads are "lightweight"
  - faster to create
  - faster to switch between

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.

Pthreads

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

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

POSIX: Potable Operating System-(ix)
a set of formal descriptions that provide a standard for the design of operating systems, especially ones which are compatible with Unix.

```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

```c
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 `pthread` Functions

More `pthread` 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

More `pthread` 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
- `void pthread_detach(pthread_tid)`
  - if a thread is detached its termination cannot be tracked with `pthread_join()`
  - it becomes a daemon thread

More `pthread` 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
- `void pthread_detach(pthread_tid)`
  - 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

```
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.

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

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

int result;
pthread_t thread_ID;

MyArg *p = (MyArg *)malloc(sizeof(MyArg));
p->fd = fd; /* assumes fd is defined */
strncpy(p->name, "CSCB09", 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()`

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()`

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()`

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

```
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));


///////////////////////////////////////////////
```

Some final comment...

Concurrency

- Critical Region
- Atomic instructions
- Barrier

- Higher level abstraction
  OpenMP

References and Further Resources

- Networking and IO Models
  `https://seenaburns.com/network-io/`