Advanced Linux Programming

ECE650 – Methods and Tools for Software Eng.

Alireza Sharifi

From a presentation by Carlos Moreno

And advanced-linux-programming.pdf

UNIVERSITY OF WATERLOO

Outline

- **During today's lecture, we'll look at:**
  - Some of POSIX/Linux facilities
    - Main focus on processes, concurrency, communication, threads and synchronization.
    - Issues with concurrency: race conditions, deadlock, starvation.
    - Tools and techniques to deal with the above: critical sections, mutual exclusion / atomicity, semaphores, pipes, message queues, shared memory.

Systems Programming

- One of the most important notions is that of a *Process*.
- Possible definitions:
  - A program in execution / An instance of a program running on a computer
    - Not really: execution of a program can involve multiple processes!
  - A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system instructions

Process

- An entity representing activity consisting on three components:
  - An executable program
  - Associated data needed by the program
  - Execution context of the program (registers, PC, pending I/O operations, etc.)
- OS assigns a unique identifier (PID)
  - See command **ps**.
- Processes can create other processes (denoted *“child process”* in that context)

See **ps -e -o pid,ppid,command --forest**

UNIVERSITY OF WATERLOO

print-pid.cpp

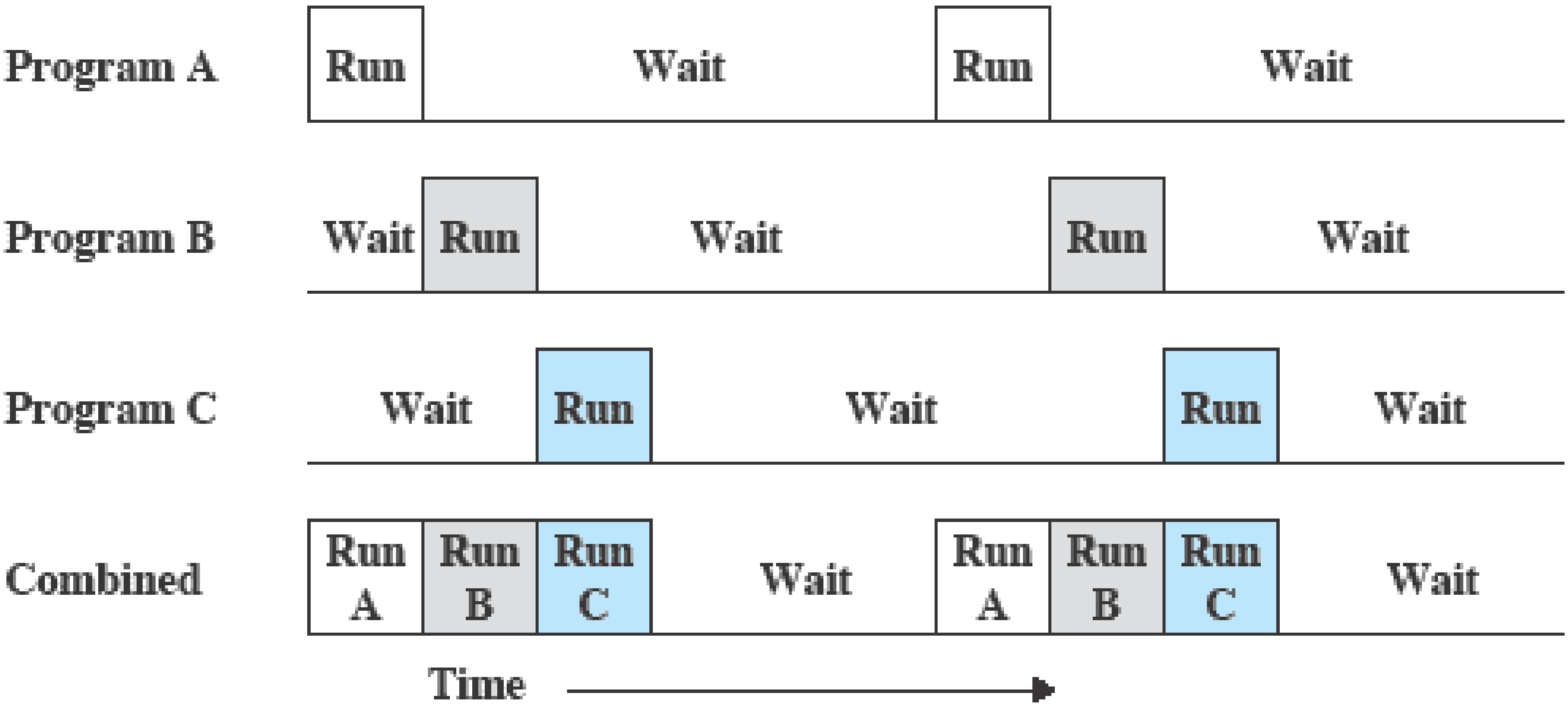
```
#include <iostream>
#include <unistd.h>

int main ()
{
  printf ("The process id is %d\n", (int) getpid ());
  printf ("The parent process id is %d\n", (int) getppid ());
  return 0;
}
```

Multiprogramming

- Concurrent execution of multiple tasks (e.g., processes)
  - Each task runs as if it was the only task running on the CPU.

- Benefits:
  - When one task needs to wait for I/O, the processor can switch to the another task.
  - (why is this potentially a *huge* benefit?)

Multiprogramming



(c) Multiprogramming with three programs

Creating Processes – `fork()`

Forks an execution of the process

- after a call to fork(), a new process is created (called child)
- the original process (called parent) continues to execute concurrently
- in the parent, fork() returns the process id of the child that was created
- in the child, fork() return 0 to indicate that this is a child process

Man(ual) Page

- man 2 fork

fork.cpp

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
  pid_t child_pid;

  std::cout << "the main program process id is " << (int) getpid
() << std::endl;
  child_pid = fork();
  if (child_pid != 0) {
        std::cout << "this is the parent process, with id " <<
(int) getpid () <<  std::endl;
        std::cout << "the child's process id is " << (int)
child_pid << std::endl;
  }
  else
        std::cout << "this is the child process, with id " <<
(int) getpid () << std::endl;

  return 0;
}
```

`exec()` – executing a program in a process

exec() series of functions are used to start another program in the current process

- after a call to exec() the current process is replaced with the image of the specified program
- different versions allow for different ways to pass command line arguments and environment settings
- `int execv(const char *file, char *const argv[ ])`
  - file is a path to an executable
  - argv is an array of arguments. By convention, argv[0] is the name of the program being executed

Man page

- man 3 exec

Spawn a process

```
int spawn (char* program, char** arg_list)
{
  pid_t child_pid;
  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.
*/
    execvp (program, arg_list);
    /* The execvp function returns only if an error
occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}
```

`kill()` – sending a signal

A process can send a signal to any other process

- usually the parent process sends signals to its children
- `int kill(pid_t pid, int sig)`
  - send a signal `sig` to a process `pid`
- useful signal: SIGTERM
  - asks a process to terminate

When a parent process exits, the children processes are terminated

It's a good practice to kill and wait for children to terminate before exiting

Man page

- man 2 kill

`wait()` – Waiting for a child

```
spawn ("ls", arg_list);

printf ("main waiting\n");
wait(&child_status);

if(WIFEXITED (child_status))
        printf("Child process exited normally, with
exit code %d\n", WEXITSTATUS (child_status));
else
        printf("Child exited abnormally");
```

`waitpid()` – Waiting for a child

A parent process can wait for a child process to terminate

- `pid_t waitpid(pid_t pid, int *stat_loc, int options)`
  - block until the process with the specified `pid` terminates
  - the return code from the terminating process is placed in `stat_loc`
  - options control whether the function blocks or not
    - 0 is a good choice for options

Man page

- man 2 wait

Zombie process

```
int main ()
{
  pid_t child_pid;
  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* This is the parent process.  Sleep for a minute.  */
    sleep (60);
  }
  else {
    /* This is the child process.  Exit immediately.  */
    exit (0);
  }
  return 0;
}
```

Multithreading

- Processes typically have their own "isolated" memory space.
  - Memory protection schemes prevent a process from accessing memory of another process (more in general, any memory outside its own space).
  - The catch: if processes need to share data, then there may be some overhead associated with it.
- Threads are a "lighter version" of processes:
  - A process can have multiple threads of execution that all share the same memory space.
  - Sharing data between threads has little or no overhead
    - Good news? Bad news? (both?)

`<pthread.h>`

- Linux implements POSIX standard thread API (*pthreads*).
- Include header file `<pthread.h>`
- The *pthread* functions are not included in the standard C library.
- They are in `libpthread`.
- When linking add `-lpthread`.

thread-create.c

```
void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

int main ()
{
  pthread_t thread_id;
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

```
man 3 pthread_create
```

Waiting for all threads to finish

- What if main thread finishes?
- What if you pass data to threads?
- main thread should wait for all threads to finish.
- Use `pthread_join()`

pthread_join()

```
int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* Create a new thread to print 30000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20000 o's.  */
  …
  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);

  /* Now we can safely return.  */
  return 0;
}
```

UNIVERSITY OF WATERLOO

Concurrency Issues

- **Race condition:**

  A situation where concurrent operations access data in a way that the outcome depends on the order (the timing) in which operations execute.

  - Doesn't necessarily mean a bug! (like in the threads example with the linked list)
  - In general it constitutes a bug when the programmer makes any assumptions (explicit or otherwise) about an order of execution or relative timing between operations in the various threads.

Concurrency Issues

- **Race condition:**

  Example (x is a shared variable):

Thread 1:	Thread 2:
x = x + 1;	x = x – 1;

  (what's the implicit assumption a programmer could make?)

UNIVERSITY OF WATERLOO

Concurrency Issues

- **Race condition:**

  Thread 1:

  x = x + 1;

  Thread 2:

  x = x – 1;

- In assembly code:

  Thread 1:

  R1 ← x
  inc R1
  R1 → x

  Thread 2:

  R1 ← x
  dec R1
  R1 → x

Concurrency Issues

- **And this is how it could go wrong:**

Thread 1:	Thread 2:
x = x + 1;	x = x – 1;

- In assembly code:

$R1 \leftarrow x$	$R1 \leftarrow x$
inc R1	dec R1
$R1 \rightarrow x$	$R1 \rightarrow x$

Concurrency Issues

Atomicity / Atomic operation:

Atomicity is a characteristic of a fragment of a program that exhibits an observable behaviour that is non-interruptible.

It behaves as if it can only execute entirely or not execute at all, such that no other threads deal with any intermediate outcome of the atomic operation.

Concurrency Issues

- **Example of atomic operations in POSIX:**
  - Renaming / moving a file with
    int rename (const char * old, const char * new);
    Any other process can either see the old file, or the new file – not both and no other possible “intermediate” state.

Concurrency Issues

- **Mutual Exclusion:**

  Atomicity is often achieved through mutual exclusion – the constraint that execution of one thread excludes all the others.

  - In general, mutual exclusion is a constraint that is applied to sections of the code.

Concurrency Issues

- **Critical section:**

  A section of code that requires atomicity and that needs to be protected by some mutual exclusion mechanism is referred to as a *critical section*.

  - In general, we say that a program (a thread) *enters* a critical section.

Concurrency Issues

- **Mutual Exclusion – How?**

  Attempt #1: We disable interrupts while in a critical section (and of course avoid any calls to the OS)

  - There are three problems with this approach
    - Not necessarily feasible (privileged operations)
    - Extremely inefficient (you're blocking everything else, including things that wouldn't interfere with what your critical section needs to do)
    - *Doesn't always work!!* (keyword: multicore)

Concurrency Issues

- **Mutual Exclusion – How?**

  Attempt #2: We place a flag (sort of telling others "don't touch this, I'm in the middle of working with it).

```
int locked;         // shared between threads
// ...
if (! locked)
{
        locked = 1;
        // insert to the list (critical section)  locked = 0;
}
```

- Why is this flawed? (there are *several* issues)

Concurrency Issues

- **Mutual Exclusion – How?**

One of the problems: does not really work!

This is what the assembly code could look like:

```
R1 ← locked
tst R1
brnz somewhere_else
R1 ← 1
R1 → locked
```

Concurrency Issues

- **Mutex:**

  A mutex (for MUTual EXclusion) provides a clean solution: In general we have a variable of type mutex, and a program (a thread) attempts to *lock* the mutex. The attempt *atomically* either succeeds (if the mutex is unlocked) or it *blocks* the thread that attempted the lock (if the mutex is already unlocked).

  - As soon as the thread that is holding the lock unlocks the mutex, this thread's state becomes ready.

Concurrency Issues

- **Using a Mutex:**

  lock (mutex)
  critical section
  unlock (mutex)

- For example, with POSIX threads (pthreads):

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// ...
pthread_mutex_lock (&mutex);
// ... critical section
pthread_mutex_unlock (&mutex);
```

mutex

```
void* dequeue_job (void* arg)
{
  while (1) {
    struct job* next_job;
    pthread_mutex_lock (&job_queue_mutex);
    if(job_queue == NULL){
        next_job = NULL;
        break;
    }
    else{
    next_job = job_queue;
    job_queue = job_queue->next;
    }
    pthread_mutex_unlock (&job_queue_mutex);
    process_job ();
    free (next_job);
  }
  return NULL;
}
```

Concurrency and synchronization

- **Another synchronization primitive: Semaphores**
  - Semaphore: A counter with the following properties:
    - Atomic operations that increment and decrement the count
    - Count is initialized with a non-negative value
    - **wait**operation decrements count and causes caller to block if count becomes negative (if it was 0)
    - **signal**(or **post**) operation increments count. If there are threads blocked (waiting) on this semaphore, it unblocks one of them.

Concurrency and synchronization

- **enqueue / dequeue with semaphores**

```
semaphore items    = 0;
mutex_t mutex;     // why also   a  mutex?
```

```
void enqueue()
{
    while (true)
    {
        produce_item();
        lock (mutex);
        add_item();
        unlock (mutex);
        sem_signal (items);
    }
}
```

```
void dequeue()
{
    while (true)
    {
        sem_wait (items);
        lock (mutex);
        retrieve_item();
        unlock (mutex);
        consume_item();
    }
}
```

UNIVERSITY OF WATERLOO

Concurrency and synchronization

- **Mutual Exclusion with semaphores**
  - Interestingly enough – Mutexes can be implemented in terms of semaphores!

```
semaphore lock = 1;

void process ( ... )
{
    while (1)
    {
        /* some processing */
        sem_wait (lock);
            /* critical section */
        sem_signal (lock);
        /* additional processing */
    }
}
```

UNIVERSITY OF WATERLOO

Concurrency and synchronization

- **enqueue / dequeue with semaphores only**

```
semaphore items = 0;
semaphore lock = 1;
```

```
void enqueue()
{
	while (true)
	{
		produce_item();
		sem_wait (lock);
		add_item();
		sem_signal (lock);
		sem_signal (items);
	}
}
```

```
void dequeue()
{
	while (true)
	{
		sem_wait (items);
		sem_wait (lock);
		retrieve_item();
		sem_signal (lock);
		consume_item();
	}
}
```

Concurrency and synchronization

- **POSIX semaphores:**
  - For unnamed semaphores:
    - Declare a (shared – possibly as global variable) sem_t variable
    - Give it an initial value with sem_init
    - Call sem_wait and sem_post as needed.

```
sem_t items;
sem_init (&items, 0, initial_value);
// ...
sem_wait (&items)        or     sem_post (&items)
```

semaphores

```
void* dequeue_job (void* arg)
{
  while (1) {
      struct job* next_job;
      sem_wait (&job_queue_count);
      pthread_mutex_lock (&job_queue_mutex);
      next_job = job_queue;
      job_queue = job_queue->next;
      pthread_mutex_unlock (&job_queue_mutex);
      process_job (next_job);
      free (next_job);
      }…
}
```

Concurrency and synchronization

- **More on locking granularity:**
- Read/Write locks implement this functionality:
  - Threads calling read_lock do not exclude each other.
  - A thread calling write_lock excludes any other threads requesting write_lock and also any other threads requesting read_lock
    - It blocks if some thread is holding a read lock!
  - POSIX R/W Locks:

pthread_rwlock_t
pthread_rwlock_rdlock (...)
pthread_wrlock_wrlock (...)
pthread_rwlock_unlock (...)

Concurrency and synchronization

- **Starvation:**

  One of the important problems we deal with when using concurrency:

  An otherwise ready process or thread is deprived of the CPU (it's *starved*) by other threads due to, for example, the algorithm used for locking resources.

  - Notice that the writer starving is *not* due to a defective scheduler/dispatcher!

Concurrency – Deadlock

- **Deadlock:**
  - Consider the following scenario:
  - A Bank transaction where we transfer money from account A to account B
  - Clearly, there is a (dangerous) race condition
    - Want granularity — can not lock the entire bank so that only one transfer can happen at a time
    - We want to lock at the account level:
      - Lock account A, lock account B, then proceed!

Concurrency – Deadlock

- **Deadlock:**
  - Problem with this?
  - Two concurrent transfers — one from account 100 to account 200, one from account 200 to account 100.
    - If the programming is written as:
      Lock source account
      Lock destination account
      Transfer money
      Unlock both accounts

Concurrency – Deadlock

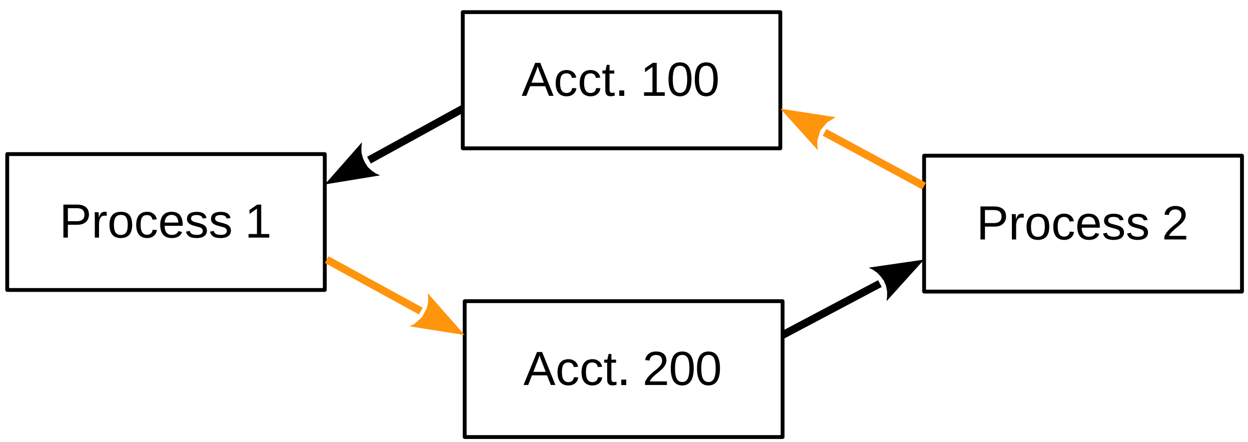
- **Deadlock:**
  - Problem with this?
  - Two concurrent transfers — one from account 100 to account 200, one from account 200 to account 100.
    - Process 1 locks account 100, then locks account 200
    - Process 2 locks account 200, then locks account 100

Concurrency – Deadlock

- **Deadlock:**
  - What about the following interleaving?
    - Process 1 locks account 100
    - Process 2 locks account 200
    - Process 1 attempts to lock account 200 (blocks)
    - Process 2 attempts to lock account 100 (blocks)
  - When do these processes unblock?
  - Answer: under some reasonable assumptions, *never!*

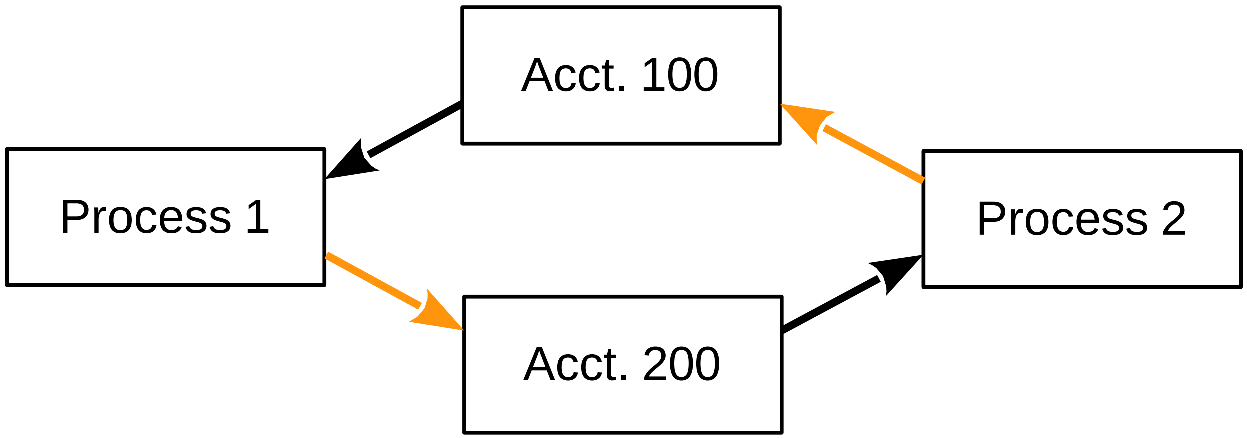
Concurrency – Deadlock

- **Deadlock:**
  - Graphically:



Concurrency – Deadlock

- **Deadlock:**



- Solution in this case is really simple:
  - Lock the resources in a given order (e.g., by ascending account number).

Processes vs Threads

- All threads in a program must run the same executable.
- A child process may run a different executable.

- An errant thread can harm other threads.
- An errant process cannot do so because each process has a copy of the program's memory space.

- Copying memory for a new process adds an additional performance overhead relative to creating a new thread. However, the copy is performed only when the memory is changed, so the penalty is minimal if the child process only reads memory.

Processes vs Threads

- Threads should be used for programs that need fine-grained parallelism. For example, if a problem can be broken into multiple, nearly identical tasks, threads may be a good choice.
- Processes should be used for programs that need coarser parallelism.
- Sharing data among threads is trivial because threads share the same memory.
- Sharing data among processes requires the use of IPC mechanisms and can be more cumbersome but makes multipleprocesses less likely to suffer from concurrency bugs.

Interprocess Communication

- **Sharing data between processes:**
  - Requires synchronization (to avoid race conditions, and to access data when there is data to be accessed!)
  - Typical mechanisms:
    - Through designated files (obvious, but inefficient)
    - Through pipes (very simple, but limited)
    - Through shared memory (efficient, but dangerous!)
    - Through message queues (convenient, though not particularly simple)

Interprocess Communication

- **Sharing data through files:**
  - Not much to say – one process writes data to a file, another process reads data from the file.
    - Still need synchronization

pipe() and dup2() – Interprocess Communication

pipe() creates a ONE directional pipe

- two file descriptors: one to write to and one to read from the pipe
- a process can use the pipe by itself, but this is unusual
- typically, a parent process creates a pipe and shares it with a child, or between multiple children
- some processes read from it, and some write to it
  - there can be multiple writers and multiple readers
    - although multiple writers is more common

dup2() duplicates a file descriptor

- used to redirect standard input, standard output, and standard error to a pipe (or another file)
- STDOUT_FILENO is the number of the standard output

Man pages

- man 2 pipe
- man 2 dup2

UNIVERSITY OF WATERLOO

pipe()

```
int fds[2];
 pid_t pid;
 pipe (fds);
 pid = fork ();
 if (pid == (pid_t) 0) {
   FILE* stream;
   close (fds[1]);
   stream = fdopen (fds[0], "r");
   reader (stream);
   close (fds[0]);
 }
 else {
   FILE* stream;
   close (fds[0]);
   stream = fdopen (fds[1], "w");
   writer ("Hello, world.", 5, stream);
   close (fds[1]);
 }
```

dup2()

```
int fds[2];
  pid_t pid;
  pipe (fds);
  pid = fork ();
  if (pid == (pid_t) 0) {
    close (fds[1]);
    dup2 (fds[0], STDIN_FILENO);
    execlp ("sort", "sort", 0);
  }
  else {
    FILE* stream;
    close (fds[0]);
    stream = fdopen (fds[1], "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "One fish, two fish.\n");
    fflush (stream);
    close (fds[1]);
    waitpid (pid, NULL, 0);
```

Interprocess Communication

- **Pipes:**
  - A pipe is a mechanism to set up a “conduit” for data from one process to another.
  - It is unidirectional (i.e., we have to predefine who transmits and who receives data)
  - Simplest form is with **popen**:
    - It executes a given command (created as a child process) and returns a stream (a FILE *) to the calling process:
    - It then connects either the standard output of that command to the (input) stream, or the standard input of that command to the (output) stream.

Interprocess Communication

- **Pipes**
  - To read the output from a program:

    FILE * child = popen (“/path/command”, “r”);
    if (child == NULL)   { /* handle error condition */ }

    Now read data with, e.g., fread (… , … , … , child); and NEVER forget to pclose (child);

  - Whatever data the child process sends to its standard output (e.g., with printf) will be read by the parent.
  - Conversely, if we popen (…. , “w”), then whatever data we write to it (e.g., with fprintf or fwrite) will appear through the standard input of the child.

Interprocess Communication

- **Shared memory:**
  - Mechanism to create a segment of memory and give multiple processes access to it.
  - **shmget**creates the segment and returns a handle to it (just an integer value)
  - **shmat** creates a logical address that maps to the beginning of the segment so that this process can use that memory area

    - If we call **fork()**, the shared memory segment is inherited shared (unlike the rest of the memory, for which the child gets an independent copy)

Interprocess Communication

- **Message queues:**
  - Mechanism to create a queue or “mailbox” where processes can send messages to or read messages from.
  - **mq_open**opens (creating if necessary) a message queue with the specified name.
  - **mq_send and mq_receive** are used to transmit or receive (receive by default blocks if the queue is empty) from the specified message queue.

Interprocess Communication

- **Message queues:**
  - Big advantages:
    - Allows multiple processes to communicate with other multiple processes
    - Synchronization is somewhat implicit!
  - See **man mq_overview**for details.

getopt() – processing CLI options

switch

switch with an argument

positional argument

```
$ foo -s -t 10 bar.txt baz.txt
```

At a start of the program, main(argc, argv) is called, where

- argc is the number of CLI arguments
- argv is an array of 0 terminated strings for arguments
  - e.g., argv[0] is “foo”, argv[1] is “-s”, argv[2] is “-t”, argv[2] is “10”, …

getopt() is a library function to parse CLI arguments

- `getopt(argc, argv, “st:”)`
- input: arguments and a string describing desired format
- output: returns the next argument and an option value
- see example in `using_getopt.cpp`

/dev/urandom – Really Random Numbers

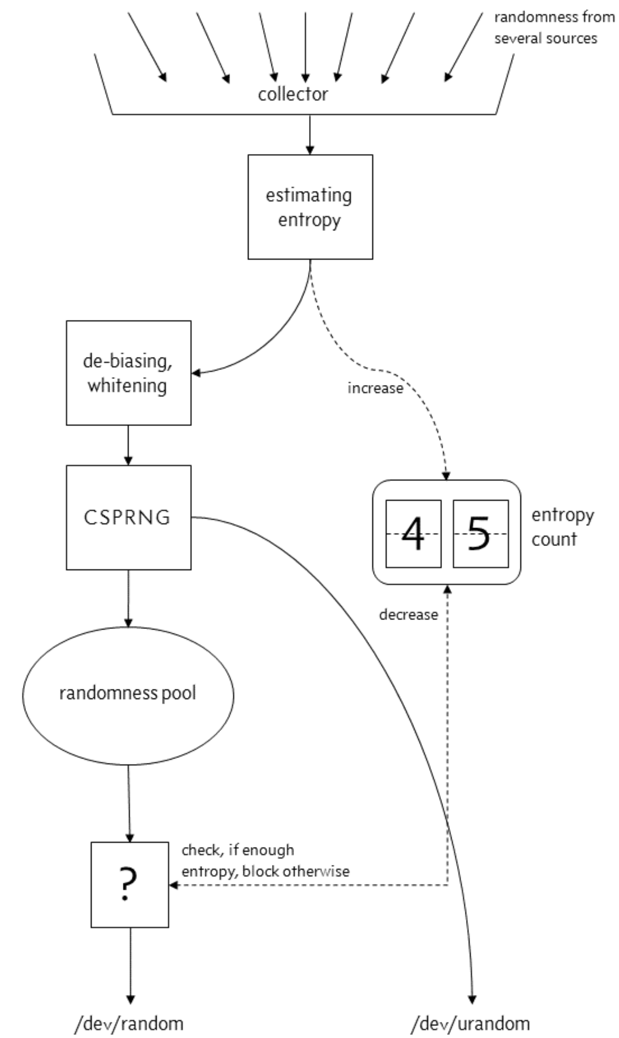
/dev/urandom is a special file (device) that provides supply of "truly" random numbers

"infinite size file" – every read returns a new random value

To get a random value, read a byte/word from the file

see `using_rand.cpp` for an example

Have to use it for Assignment 3!



https://www.2uo.de/myths-about-urandom/