Imperial College London

Operating Systems

Processes

Eno Thereska
e.thereska@imperial.ac.uk
http://www.imperial.ac.uk/computing/current-students/courses/211/

Partly based on slides from Julie McCann and Cristian Cadar

Imperial College London

Administrativia

- Lecturer: Dr Eno Thereska
  - Email: e.thereska@imperial.ac.uk
  - Office: Huxley 450

- Class website

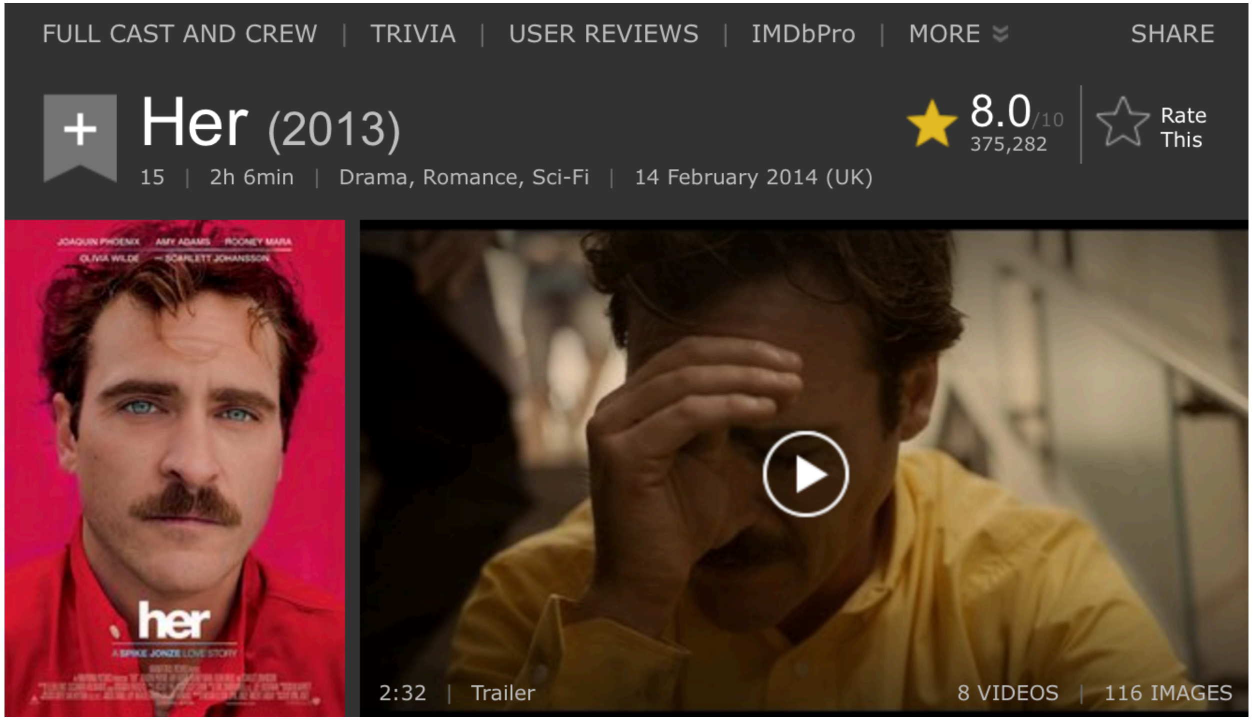
http://www.imperial.ac.uk/computing/current-students/courses/211/

Tutorials

- No separate tutorial slots
  - Tutorials embedded in lectures
- Tutorial exercises, with solutions, distributed on the course website
  - You are responsible for studying on your own the ones that we don't cover during the lectures
  - Let me know if you have any questions or would like to discuss any others in class

Tutorial question & warmup

- List the most important resources that must be managed by an operating system in the following settings:
  - Supercomputer
  - Smartphone
  - "Her"



A lonely writer develops an unlikely relationship with an operating system designed to meet his every need.
Copyright IMDB.com

Introduction to Processes

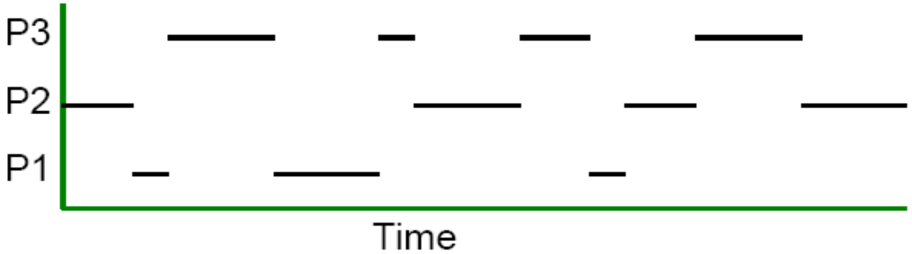
- One of the oldest abstractions in computing
  - An instance of a program being executed, a running program
- Allows a single processor to run multiple programs “simultaneously”
  - Processes turn a single CPU into multiple virtual CPUs
  - Each process runs on a virtual CPU

Why Have Processes?

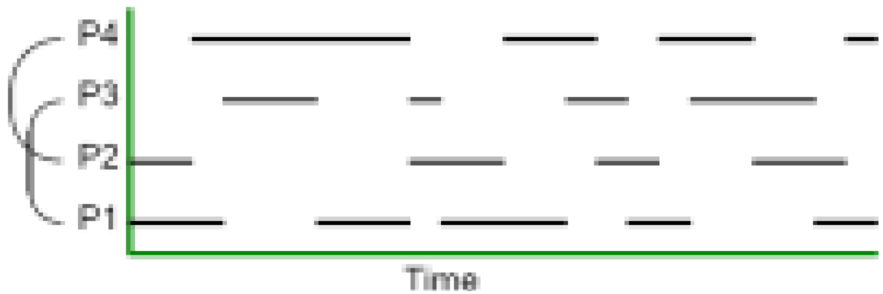
- Provide (the illusion of) concurrency
  - Real vs. apparent concurrency
- Provide isolation
  - Each process has its own address space
- Simplicity of programming
  - Firefox doesn't need to worry about gcc
- Allow better utilisation of machine resources
  - Different processes require different resources at a certain time

Concurrency

Apparent Concurrency (pseudo-concurrency): A single hardware processor which is switched between processes by interleaving. Over a period of time this gives the illusion of concurrent execution.



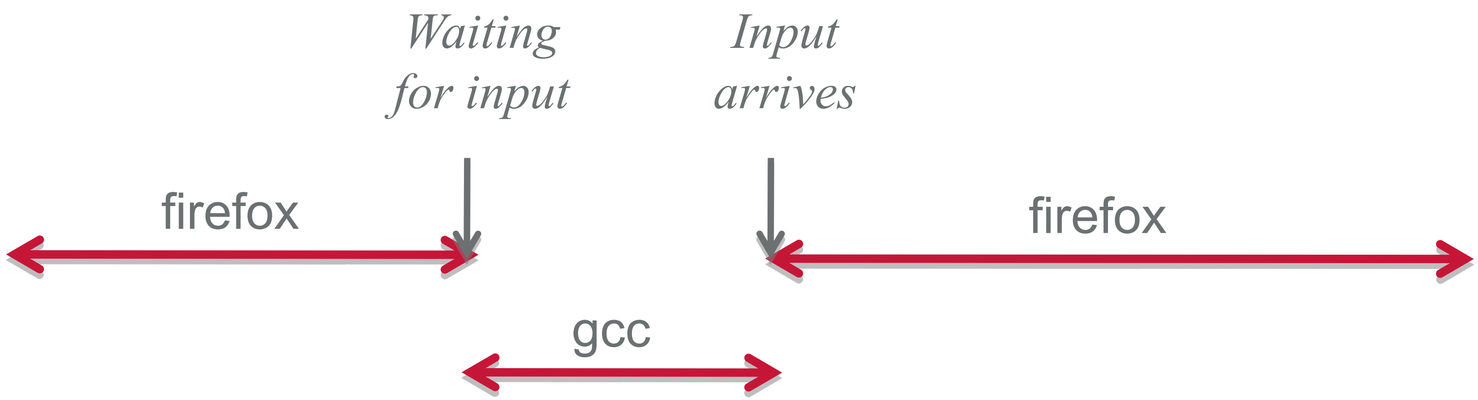
Real Concurrency: Multiple hardware processors; usually fewer processors than processes



Imperial College London

Fairness

Better CPU utilisation



Imperial College London

CPU Utilisation in Multiprogramming (Uniprocessor)

Q: Average process computes 20% time, then with five processes we should have 100% CPU utilization, right?

A: In the ideal case, if the five processes never wait for I/O at the same time

- Better estimate
  - **n** = total number of processes
  - **p** = fraction of time a process is waiting for I/O

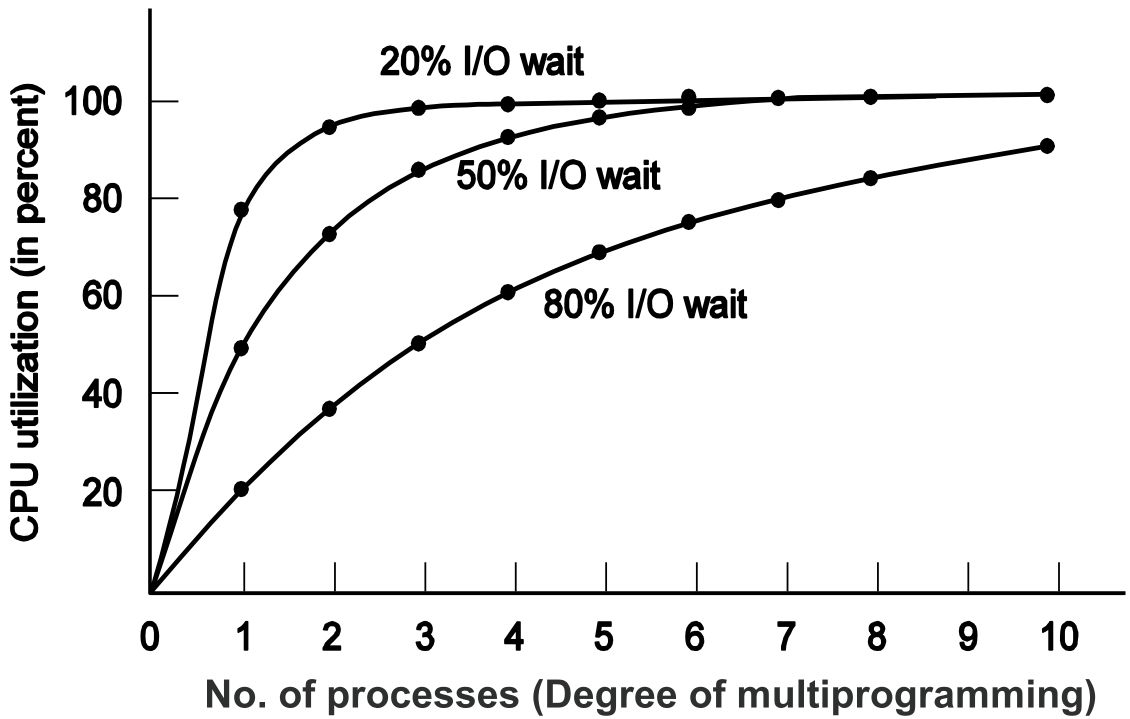
$$\text{Prob(all processes waiting for I/O)} = p^n$$

$$\text{CPU utilization} = 1 - p^n$$

CPU Utilisation = $1 - p^n$

Q: How many processes need to be in memory to only waste 10% of CPU where we know that processes spend 80% waiting for I/O (e.g. data oriented or interactive systems)

A: $1 - 0.8^n = 0.9 \Rightarrow 0.8^n = 0.1 \Rightarrow n = \log_{0.8}0.1 \approx 10$



Context Switches

- On a context switch, the processor switches from executing process A to executing process B

- OS may take periodic scheduling decisions
- OS may switch processes in response to events/interrupts (e.g., I/O completion)
  - The way an OS switches between processes cannot be pre-determined, since the events which cause the switches are non-deterministic

Context Switches

- On a context switch, the processor switches from executing process A to executing process B

- Process A may be restarted later, therefore, all information concerning the process, needed to restart safely, should be stored
- For each process, all this data is stored in a *process descriptor*, or *process control block* (PCB), which is kept in the *process table*

Process Control Block (PCB)

- A process has its own virtual machine, e.g.:
  - Its own virtual CPU
  - Its own address space (stack, heap, text, data etc.)
  - Open file descriptors, etc.
- What information should be stored?
  - Program counter (PC), page table register, stack pointer, etc.
  - Process management info:
    - Process ID (PID), parent process, process group, priority, CPU used, etc.
  - File management info
    - Root directory, working directory, open file descriptors, etc.

Context Switches Are Expensive

- Direct cost: save/restore process state
- Indirect cost: perturbation of memory caches, TLB
  - TLB (translation lookaside buffer) → caches mappings of virtual addresses to physical addresses, and is typically flushed on a context switch
  - More in memory management lectures

- Important to avoid unnecessary context switches

Process Creation

- When are processes created?
  - System initialisation
  - User request
  - System call by a running process

- Processes can be
  - Foreground processes: interact with users
  - Background processes: handle incoming mail, printing requests, etc. (**daemons**)

Process Termination

- **Normal completion:** Process completes execution of body
- **System call:**
  - `exit()` in UNIX
  - `ExitProcess()` in Windows
- **Abnormal exit:** The process has run into an error or an unhandled exception
- **Aborted:** The process stops because another process has overruled its execution (e.g., killed from terminal)
- **Never:** Many real-time processes run in endless loop and never terminate unless error occurs

Imperial College London

Process Hierarchies

- Some OSes (e.g., UNIX) allow processes to create **process hierarchies** e.g. parent, child, child's child, etc.
  - E.g., when UNIX boots it starts running `init`
  - It reads a file saying how many terminals to run, and forks off one process per terminal
  - They wait for someone to login
  - When login successful login process executes a shell to accept commands which in turn may start up more processes etc.
  - All processes in the entire system form a process tree with **init** as the root (***process group***)
- Windows has no notion of hierarchy
  - When a child process is created the parent is given a token (***handle***) to use to control it
  - The handle can be passed to other processes thus no hierarchy

Imperial College London

Case Study: UNIX

Imperial College London

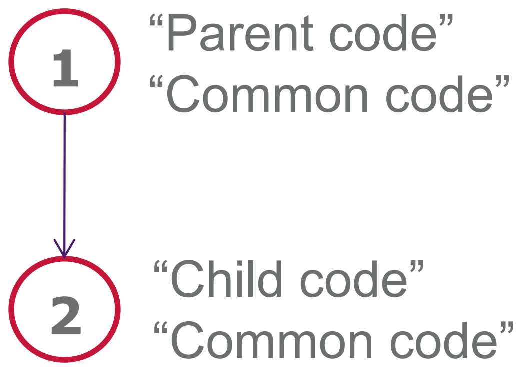
Creating processes

```
int fork(void)
```

- Creates a new child process by making an exact copy of the parent process image
- The child process inherits the resources of the parent process and will be executed concurrently with the parent process
- `fork()` returns twice:
  - In the parent process: **`fork()`** returns the process ID of the child
  - In the child process: **`fork()`** returns 0
- On error, no child is created and -1 is returned in the parent
- How can fork() fail?
  - Global process limit exceeded, per-user limit exceeded, not enough swap space

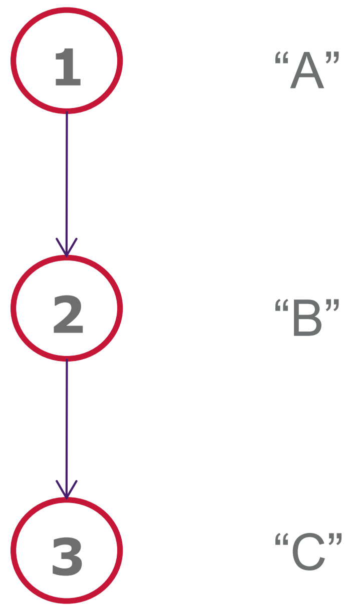
fork() example (1)

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

int main() {
  if (fork() != 0)
    printf("Parent code\n");
  else printf("Child code\n");

  printf("Common code\n");
}
```



fork() example (2)

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

int main() {
  if (fork() != 0)
    printf("A\n");
    else
      if (fork() != 0)
        printf("B\n");
      else printf("C\n");
}
```

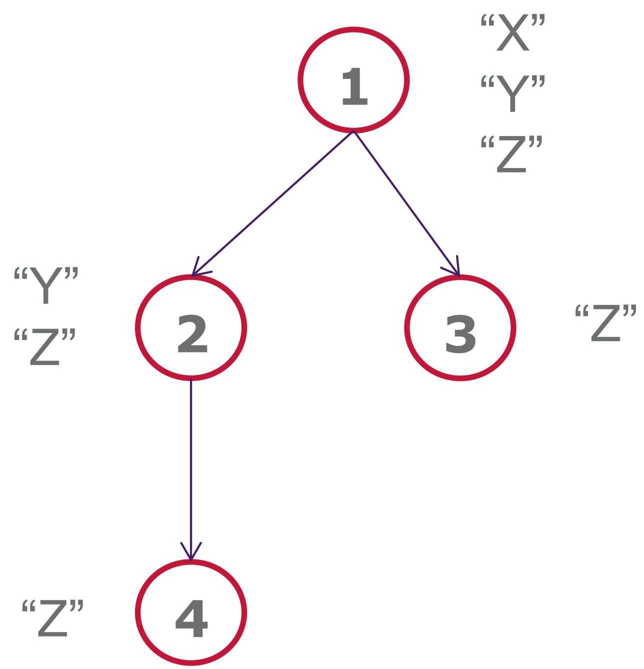


Imperial College London

fork() example (3)

Mentimeter

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

int main() {
  if (fork() != 0)
    printf("X\n");

  if (fork() != 0)
    printf("Y\n");

  printf("Z\n");
}
```



Imperial College London

fork() example (4)

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main() {
  if (fork() != 0)
    printf("%d: X\n", getpid());

  if (fork() != 0)
    printf("%d: Y\n", getpid());

  printf("%d: Z\n", getpid());
}
```

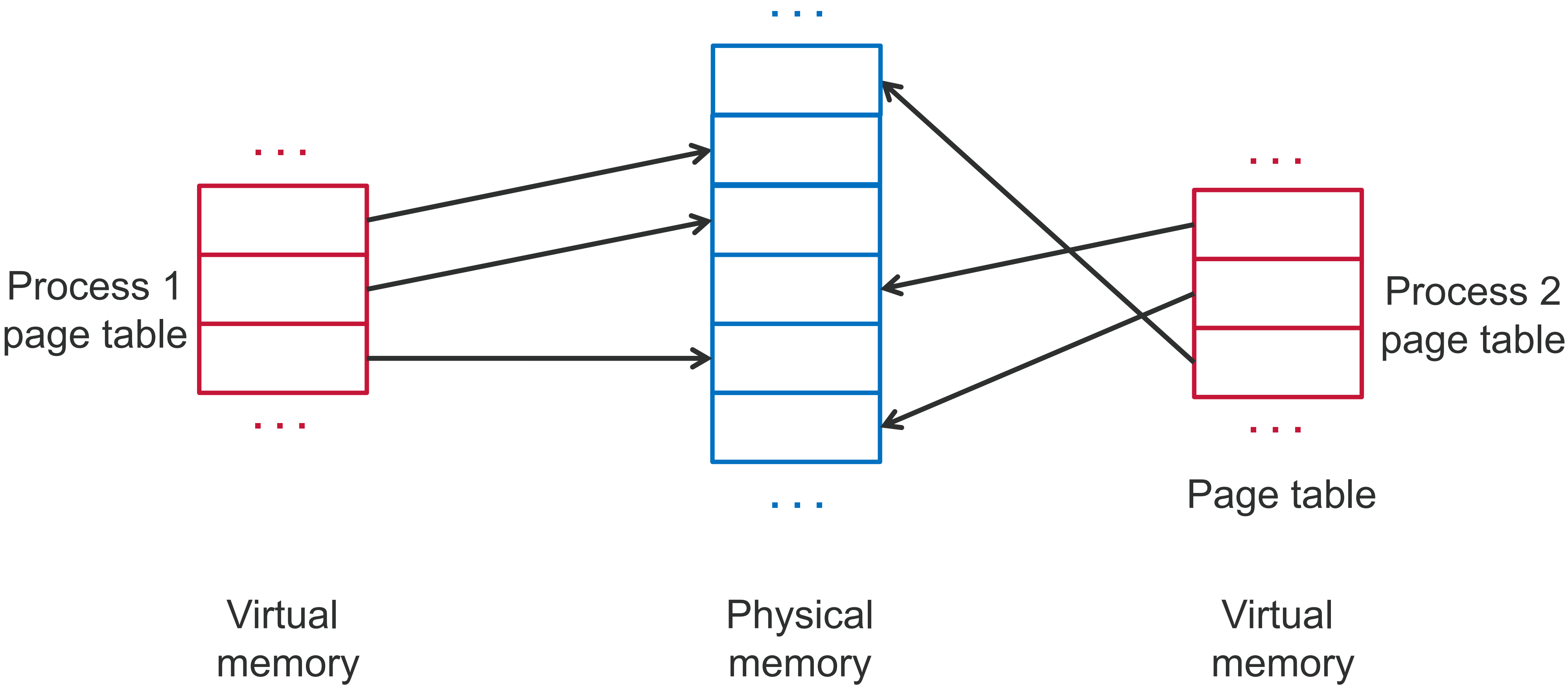
```
$ ./a.out
29221: X
29221: Y
29221: Z
29222: Y
29222: Z
29223: Z
29224: Z
```

Optimizing fork()

- fork(): Creates a new child process by making an exact copy of the parent process image
- Copying the entire address space is expensive!
  - And very few memory pages are going to end up having different values in the two processes

Imperial College London

Page tables: high level view



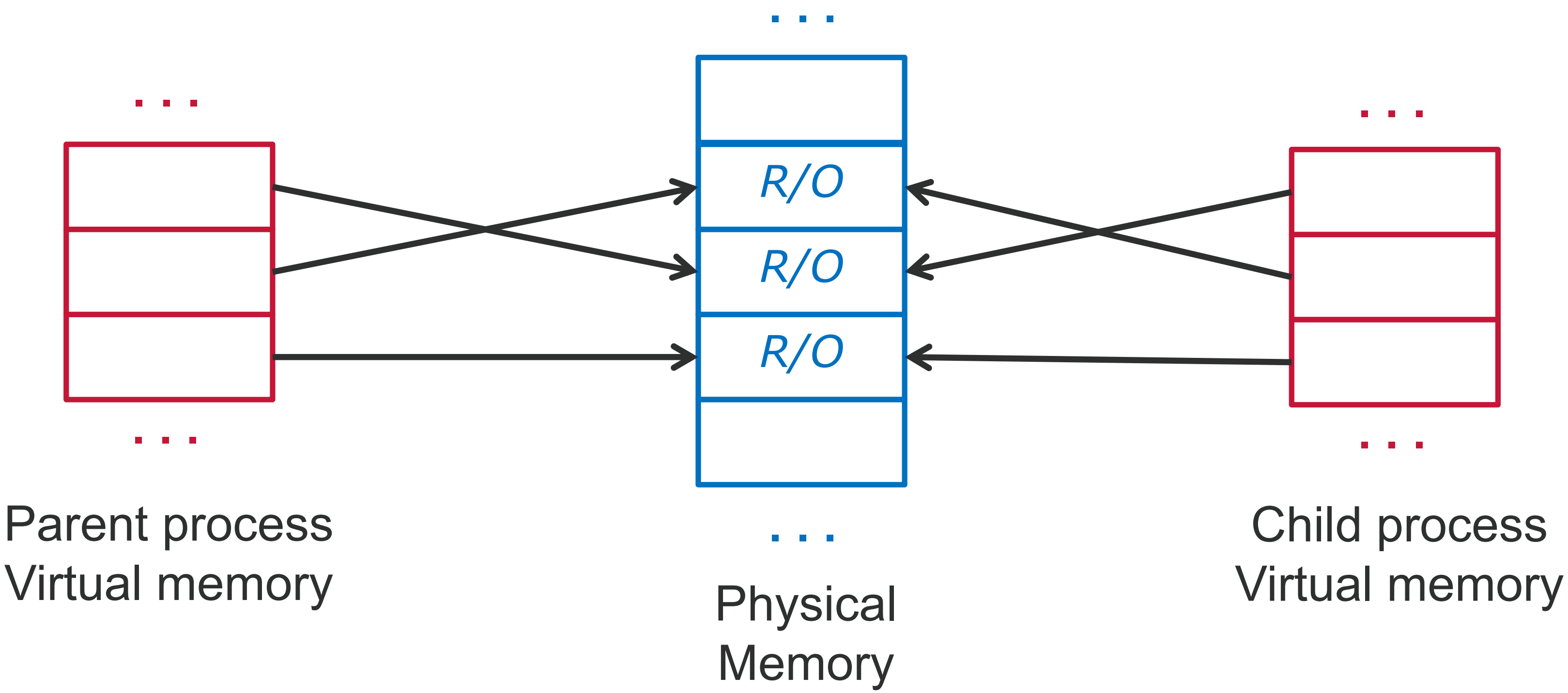
Schematic view: refer to “Memory management” lectures for details

Imperial College London

Copy On Write (COW)

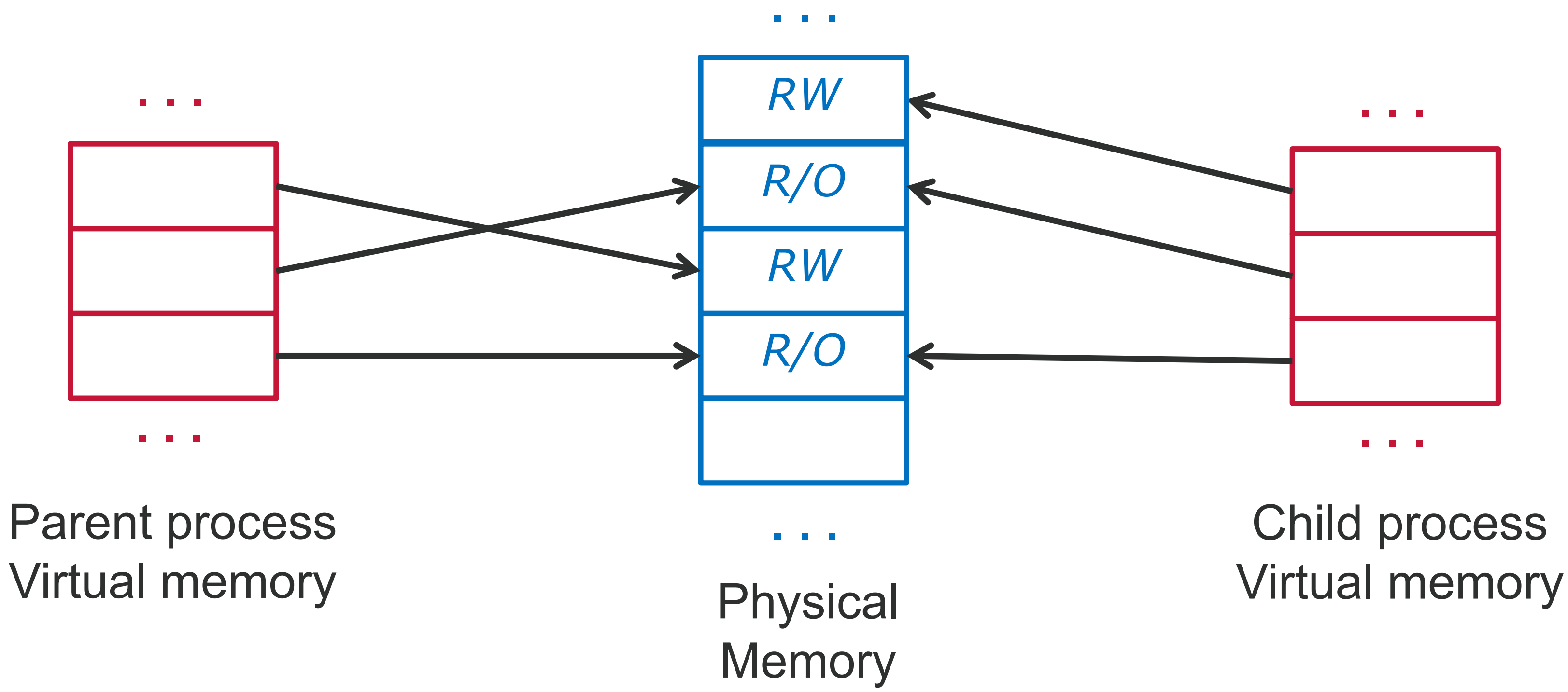
- Give child its own page table pointing to parent's pages which are marked as *read only*
- When any process writes to a page
  - Protection fault causes trap to the kernel
  - Kernel allocates new copy of page so that both processes have their own private copies
  - Both copies are marked *read-write*

COW



Schematic view: refer to “Memory management” lectures for details

COW: Example



Schematic view: refer to “Memory management” lectures for details

Executing processes

```
int execve(const char *path, char *const argv[],
               char *const envp[])
```

- Arguments:
  - **path** – full pathname of program to run
  - **argv** – arguments passed to main
  - **envp** – environment variables (e.g., $PATH, $HOME)
- Changes process image and runs new process
- Lots of useful wrappers:
  - E.g., execl, execle, execvp, execv, etc.

```
man execve
```

Consult man(ual) pages!

Imperial College London

Waiting for Process Termination

```
int waitpid(int pid, int* stat, int options)
```

- Suspends execution of the calling process until the process with PID pid terminates normally or a signal is received
- Can wait for more than one child:
  - pid = -1 wait for any child
  - pid = 0 wait for any child in the same process group as caller
  - pid = -gid wait for any child with process group gid
- Returns:
  - pid of the terminated child process
  - 0 if WNOHANG is set in options (indicating the call should not block) and there are no terminated children
  - -1 on error, with errno set to indicate the error

fork(), execve() and waitpid() in action

A command interpreter could do:

```
while (TRUE)  { /* repeat forever */
  read_command (command,parameters)
  if (fork () != 0)    /* fork off child process */
    waitpid(-1, &status, 0);   /* Parent code */
  else        /* Child code */
    execve (command, parameters, 0);
              /* execute command */
}
```

Why both fork() and execve() ?

- UNIX design philosophy: **simplicity**
  - Simple basic blocks that can be easily combined
- Contrast with Windows:
  - CreateProcess() => equivalent of fork() + execve()
  - Call has 10 parameters!
    - program to be executed
    - parameters
    - security attributes
    - meta data regarding files
    - priority
    - pointer to the structure in which info regarding new process is stored and communicated to the caller
    - …

CreateProcess()

```
BOOL WINAPI CreateProcess(
    __in_opt LPCTSTR lpApplicationName,
    __inout_opt LPTSTR lpCommandLine,
    __in_opt LPSECURITY_ATTRIBUTES lpProcessAttributes,
    __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
    __in BOOL bInheritHandles,
    __in DWORD dwCreationFlags,
    __in_opt LPVOID lpEnvironment,
    __in_opt LPCTSTR lpCurrentDirectory,
    __in LPSTARTUPINFO lpStartupInfo,
    __out LPPROCESS_INFORMATION lpProcessInformation )
```

Process termination

```
void exit(int status)
```

- Terminates a process
  - Called implicitly when program finishes execution
- Never returns in the calling process
  - Returns an exit status to the parent process

```
void kill(int pid, int sig)
```

- Sends signal sig to process pid

Tracing a Process

```
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

How Can Processes Communicate?

- Files
- Signals (UNIX)
- Events, exceptions (Windows)
- Pipes
- Message Queues (UNIX)
- Mailslots (Windows)
- Sockets
- Shared memory
- Semaphores

UNIX Signals

- Inter-Process Communication (IPC) mechanism
- Signal delivery similar to delivery of hardware interrupts
- Used to notify processes when an event occurs
- A process can send a signal to another process if it has permission to do so:
  - *"the real or effective user ID of the receiving process must match that of the sending process or the user must have appropriate privileges (such as given by a set-user-ID program or the user is the super-user)."* (man page)
  - The kernel can send signals to any process

Imperial College London

When Are Signals Generated?

- When an exception occurs
  - e.g., division by zero => `SIGFPE`, segment violation => `SIGSEGV`
- When the kernel wants to notify the process of an event
  - e.g., if process writes to a closed pipe => `SIGPIPE`
- When certain key combinations are typed in a terminal
  - e.g., Ctrl-C => `SIGINT`
- Programmatically using the `kill()` system call

Imperial College London

UNIX Signals – Examples

`SIGINT`	Interrupt from keyboard
`SIGABRT`	Abort signal from `abort`
`SIGFPE`	Floating point exception
`SIGKILL`	Kill signal
`SIGSEGV`	Invalid memory reference
`SIGPIPE`	Broken pipe: write to pipe with no readers
`SIGALRM`	Timer signal from `alarm`
`SIGTERM`	Termination signal

UNIX Signals

- The default action for most signals is to terminate the process
- But the receiving process may choose to
  - Ignore it
  - Handle it by installing a signal handler
  - Two signals cannot be ignored/handled: **`SIGKILL`** and **`SIGSTOP`**

```
signal(SIGINT, my_handler);

void my_handler(int sig) {
    printf("Received SIGINT. Ignoring...")
}
```

Imperial College London

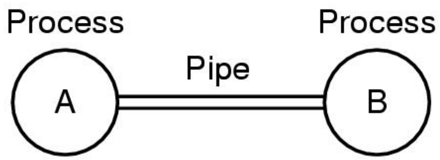
Signal Handlers – Example

```
#include <signal.h>
#include <stdio.h>

void my_handler(int sig) {
  fprintf(stderr, "SIGINT caught!”);
}

int main(int argc, char *argv[])
{
  signal(SIGINT, my_handler);
  while (1) {}
}
```

```
$ ./a.out
[ctrl-C]
SIGINT caught
```

UNIX Pipes

- *A* **pipe** is a method of connecting the *standard output* of one process to the *standard input* of another
  - Allows for **one-way** communication between processes



- Widely-used on the command line and in shell scripts
  - `ls | less`
  - `cat file.txt | grep hello | wc -l`

- Two types of pipes
  - unnamed
  - named

Imperial College London

pipe()

```
int pipe(int fd[2])
```

- Returns two file descriptors in `fd`:
  - `fd[0]` – the read end of the pipe
  - `fd[1]` – the write end of the pipe
- The sender should close the read end
- The receiver should close the write end

- If the receiver reads from an empty pipe, it blocks until data is written at the other end
- If the sender attempts to write to a full pipe, it blocks until data is read at the other end

Imperial College London

pipe() example

```
int main(int argc, char *argv[]) {
  int fd[2]; char buf;
  assert(argc == 2);
  if (pipe(fd) == -1) exit(1);

  if (fork() != 0) {
    close(fd[0]);
    write(fd[1], argv[1], strlen(argv[1]));
    close(fd[1]);
    waitpid(-1, NULL, 0);
  } else {
    close(fd[1]);
    while (read(fd[0], &buf, 1) > 0)
      printf("%c", buf);
    printf("\n");
    close(fd[0]);
  }
}
```

```
$ ./a.out abc
abc
```

UNIX Named Pipes (FIFOs)

- Persistent pipes than outlive the process which created them
- Stored on the file system
- Any process can open it like a regular file
    - *Why ever use named pipes instead of files?*

```
$ mkfifo /tmp/abc
$ echo ABC >/tmp/abc
```

```
$ cat /tmp/abc
ABC
```

Tutorial question

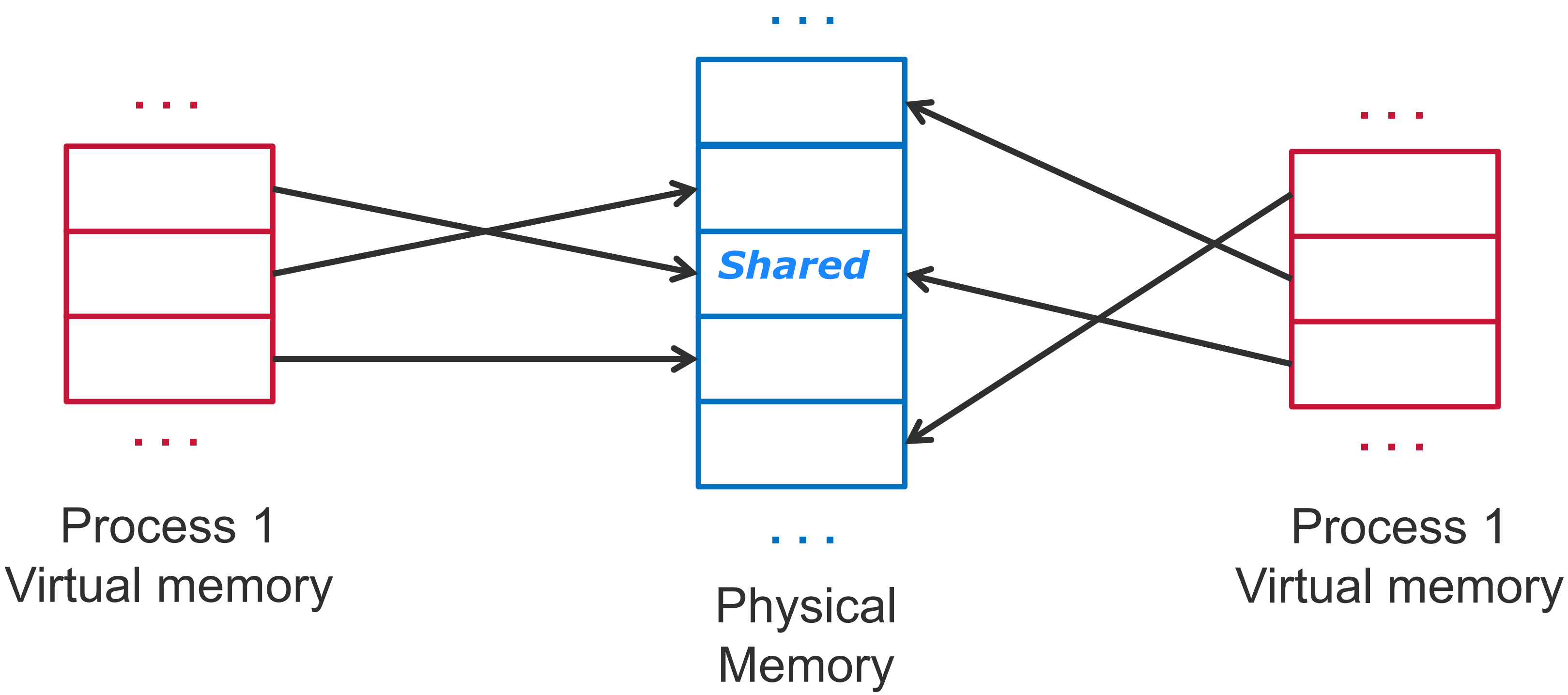
- The process at the write-end of the pipe wants to transmit a linked list data structure (with one integer field and a ``next'' pointer) over a pipe. How can it do this?

Sockets

- Allow bidirectional communication
- Can be used to exchange information both locally and across a network
  - Unlike pipes which are identified by machine specific file descriptors
- Two types of sockets:
  - TCP (stream sockets)
  - UDP (datagram sockets)

Shared Memory

- Processes can set up shared memory areas
  - Implicitly or explicitly mapped to files on disk
- After shared memory is established, no need for kernel involvement



Imperial College London

Shared Memory – System V API

`shmget`	Allocates a shared memory segment
`shmat`	Attaches a shared memory segment to the address space of a process
`shmctl`	Changes the properties associate with a shared memory segment
`shmdt`	Detaches a shared memory segment from a process

Imperial College London

Tutorial question

- When would it be better for two processes to communicate via shared memory instead of pipes? What about the other way around?