Lancaster University

# Lecture 3 – Processes and the Operating System

# Outline

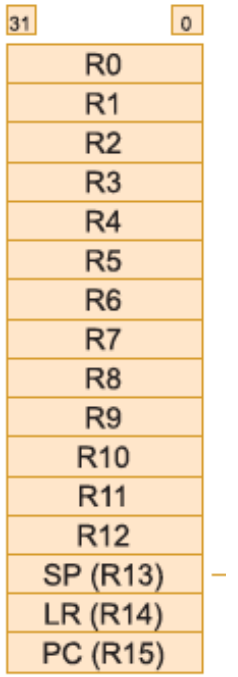
Lancaster University

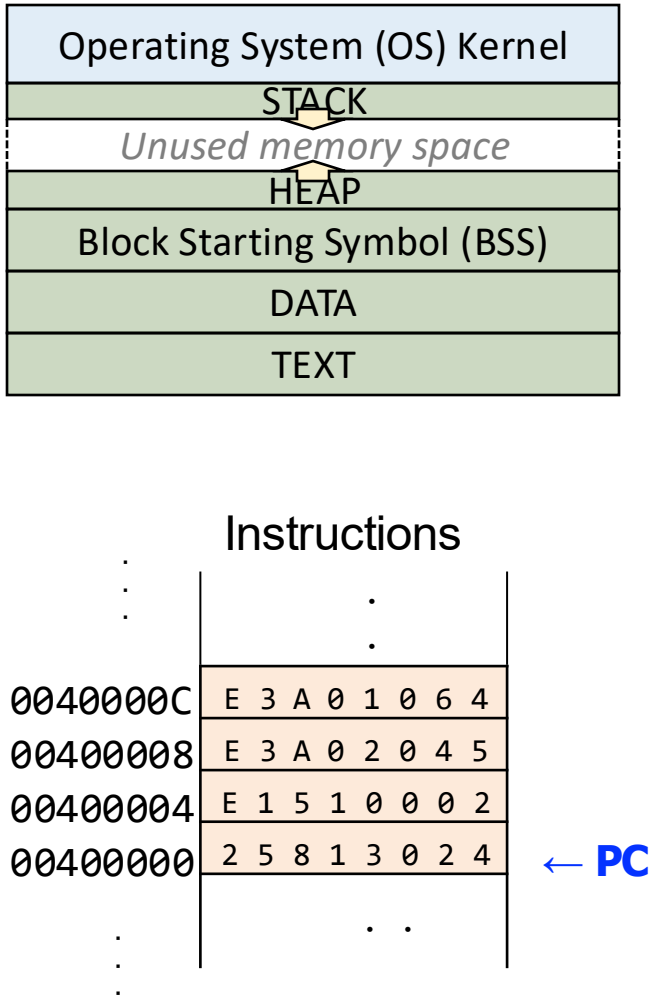
- Recap of computer architecture
- What is the OS?
  - Kernel, process
- CPU Virtualisation
- System Calls (remind what is a CPU exception)
- How do you execute a process over an OS?
- Scheduling

# Computer Architecture Recap

Lancaster University

- Instruction-set architectures is the lowest **interface between computer hardware and software**
  - It defines *how software tells the hardware what to do*
  - Defined by the CPU architecture
  - Examples: ARM Assembly
- Most of our program assumed exclusive access to CPU resources and no OS was used.
  - What happens if two assembly processes want to access the CPU?

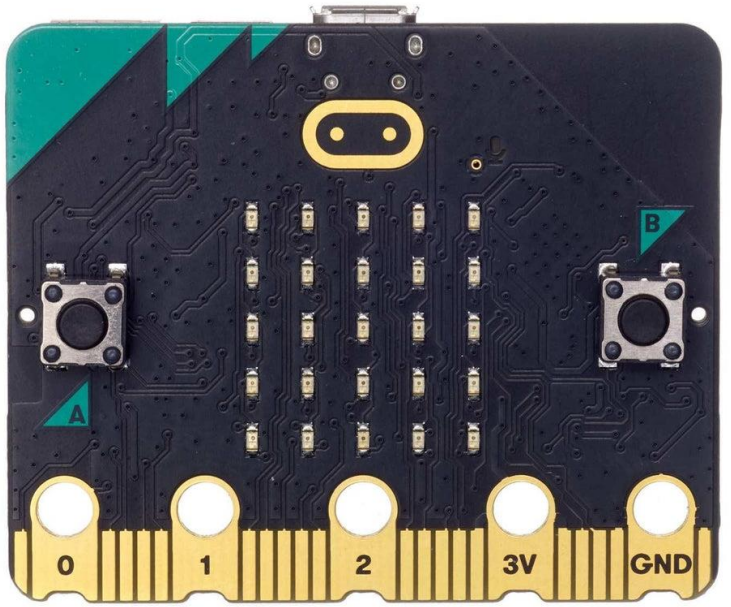


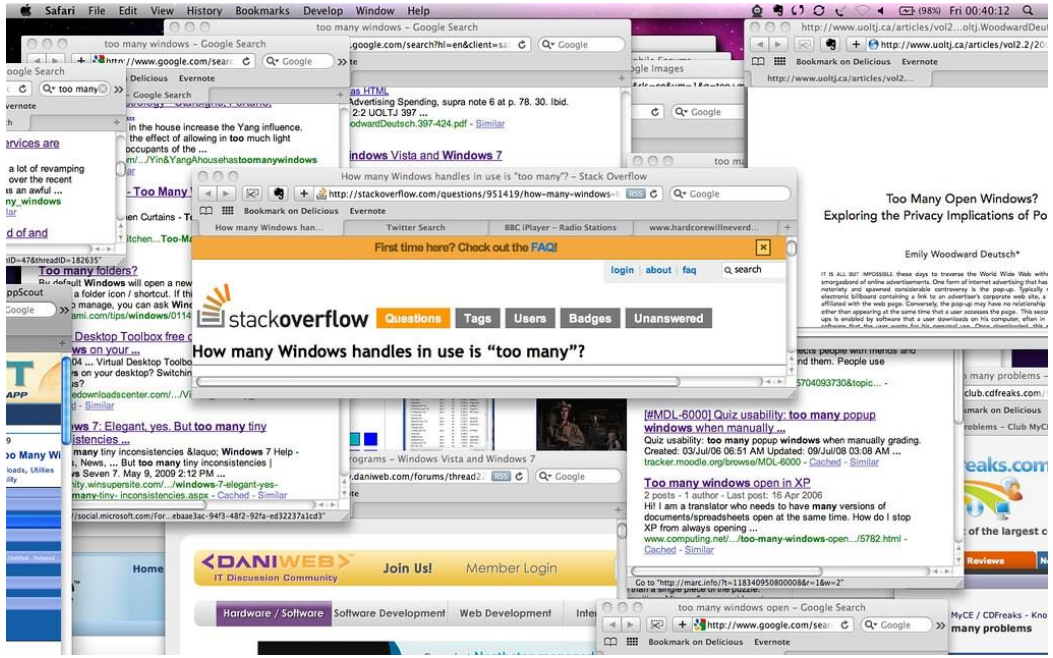
# Single-process computers

Lancaster University

Single program loaded in memory, executed to completion

# Modern computers

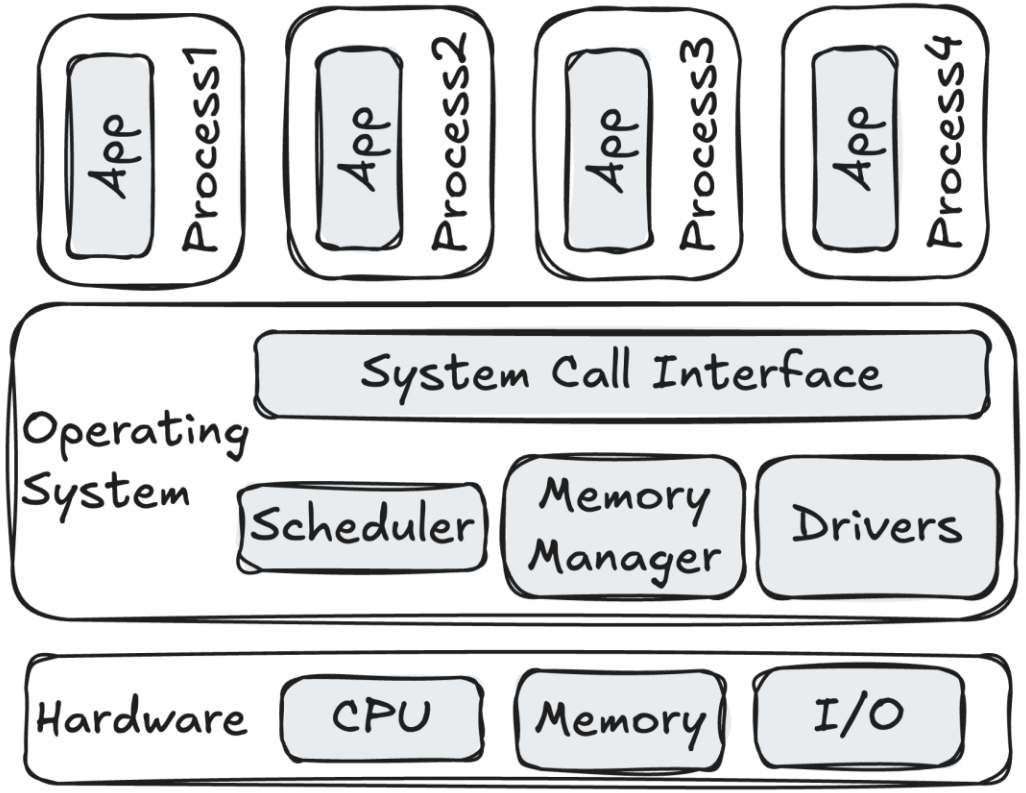



- Lots of processes running in parallel…
- Lots of IO devices shared…
- Programs are responsive…
- How do modern computers achieve parallelism?

# What is an operating system?

Lancaster University

- An **Operating System** is a software layer that sits between hardware and user applications.
  - Manage resources
  - Safe and convenient programming abstractions
  - Services: scheduling, device drivers
- CPU virtualization: control which process uses the CPU



# Virtualization

Lancaster University

- A combination of indirection and multiplexing
- Create a **virtual version of a resource** (hardware, OS, network, or storage) so that multiple independent systems or applications can share it as if each has its own dedicated resource
- Examples:
  - virtual memory
  - virtual modem
  - Cloud computing
  - Santa claus
- Can cleanly and dynamically reconfigure the system

# OS Virtualisation

Lancaster University

- **Goal**: give each process the illusion of exclusive resource access
- **Constraints**: The resources are fixed and shared among all processes, ensuring isolation and security
- **Time-sharing**: exclusive use, one at a time
- **Space-sharing**: everyone gets a small chunk all the time

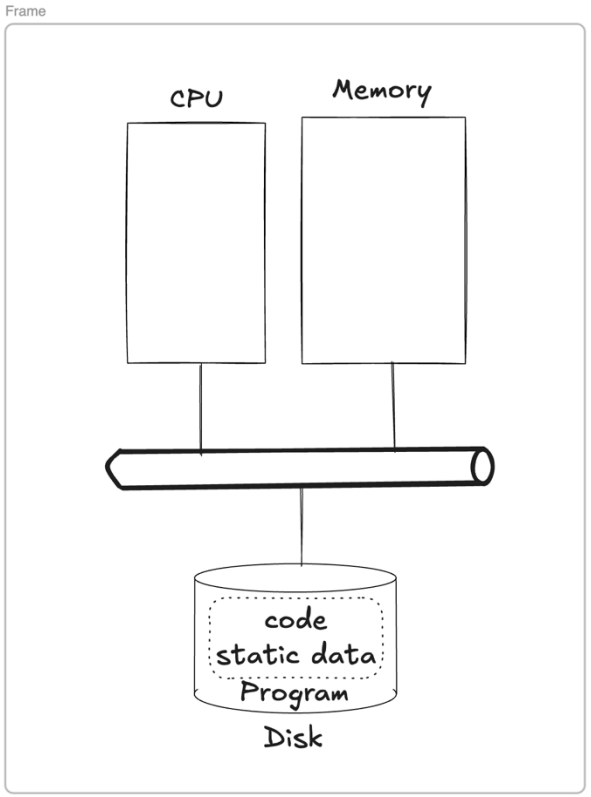
- Different strategies for resources
  - CPU: time sharing, alternate between tasks
  - Memory: space sharing (more later)
  - Disk: space sharing (more later)

# What is a process?

Lancaster University

- A **program** is a bundle of binary instructions and static data, stored on the disk.

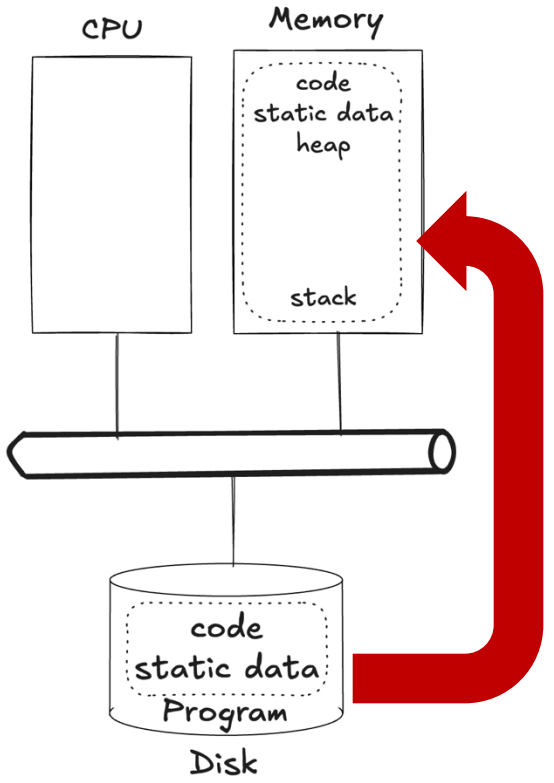
- A **process** is an OS abstraction that represents a program loaded into memory and executed. It includes:
  - Memory resources
  - Execution state (registers)
  - Resources (e.g., open files)
  - Process control block (state maintained by the OS)



# What is a process?

Lancaster University

- A **program** is a bundle of binary instructions and static data, stored on the disk.
- A **process** is a running instance of a program. It includes:
  - Memory resources
  - Execution state (registers)
  - Resources (e.g., open files)
  - Process control block (state maintained by the OS)
- A process can have one or more **threads**.



# Process API

Lancaster University

- UNIX systems offer an extended API system to manage processes:
- *`fork()`*: duplicate the current process (copy memory and register state)
  - The two processes will have different PID
  - The original process is now the *parent process*, and the new process is a *child process*.
- `exec()/execvp()`: replaces the current process's memory image with a new program
- `wait()`: parent process waits for one of its child processes to finish
  - If wait is not called, the child process enters zombie state (terminated, but the state not thoroughly cleaned up)
- Lots of examples during the lab activities this week…

# Process Creation

Lancaster University

- OS allocates internal data structures
- OS allocates an address space
  - Loads code, data from disk
  - Creates runtime stack, heap
- OS opens basic files (STDIN, STDOUT, STDERR)
- OS initializes CPU registers

# CPU virtualisation

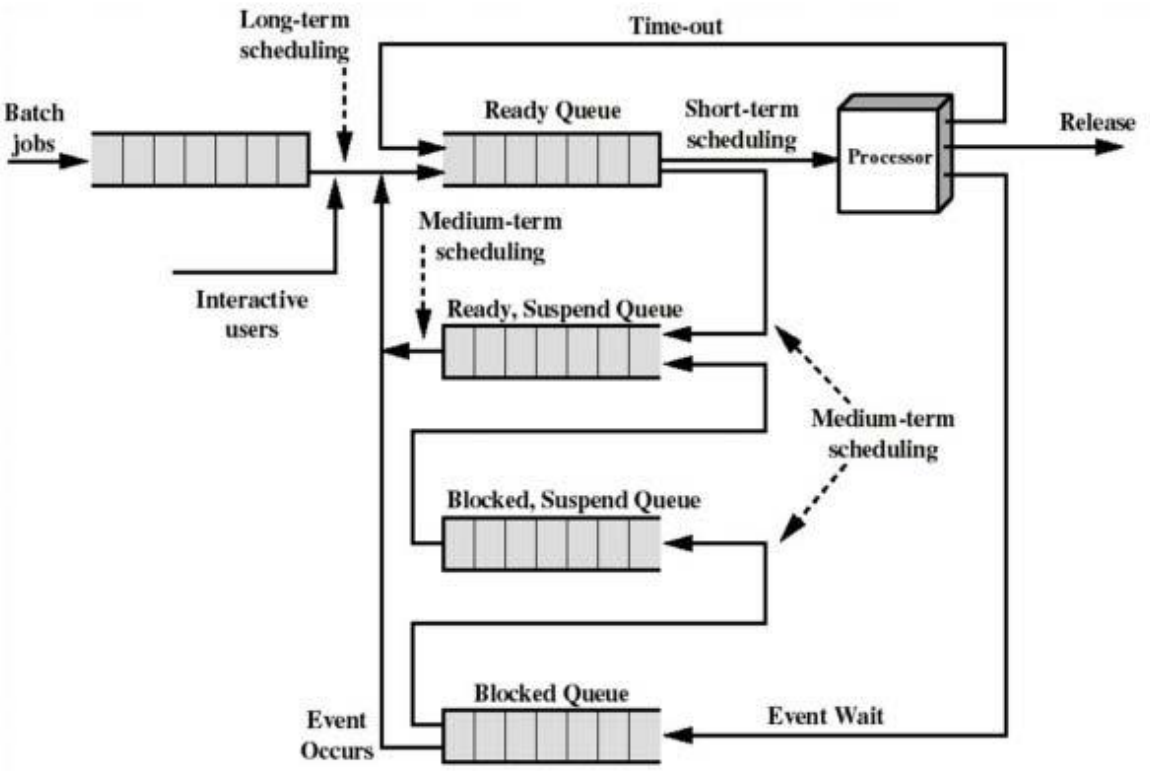
Lancaster University

- When the user executes a program, the OS creates a process.
- OS time-shares CPU across multiple processes.
  - OS scheduler picks one of the executable processes to run.
  - Scheduler must keep a list of processes
  - Scheduler must keep metadata for policy

- **How do you execute code in a secure way, without incurring performance penalties?**

# OS Scheduling

- An **OS scheduler** is the part of the operating system that decides **which process gets to use the CPU (or other resources) at any given time**.
  - It ensures efficient, fair, and responsive execution of multiple processes.
- Extended research explores how to optimize different performance metrics:
  - Throughput, waiting time, response time, fairness…
- More details in SCC.233…



# Direct Execution (no limits)

Lancaster University

| OS | Program |
| --- | --- |
| *Create entry for process list*<br>*Allocate memory for program*<br>*Load program into memory*<br>*Set up stack with argc/argv*<br>*Clear registers*<br>*Execute call main()* | |
| | Run main()<br>*Execute return from main* |
| Free memory of process<br>Remove from process list | |

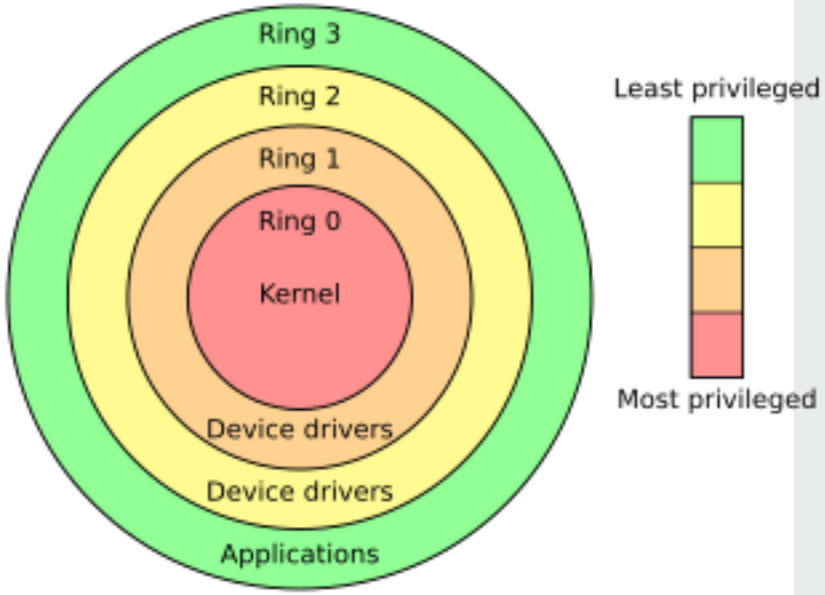
*Problem 1 (security): if we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently?*

*Problem 2 (sharing): when we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?*

# Problem 1: Security

Lancaster University

- A process needs to access the disk.
  - How do you check user permissions?
- Solution: protection rings enforced by the CPU
  - User mode: the code cannot access devices and kernel data (processes)
  - Kernel model: the code has unlimited access to resources (kernel)
- How can a process request a service from the kernel (e.g., access a file?)
  - Special type of exception, called **system call**.



# Exceptions

Lancaster University

- The programmer defines program execution.
  - Nonetheless, events occur external to the program logic.
- An exception is used to model an unexpected event during program execution.
  - It looks like a function call that branches to a new address.
  - Allows a program/OS to implement code that manages external events.
- Exceptions may be caused by hardware or software.
  - User presses a key on a keyboard (hardware exception, input/output (I/O) device);
  - Encounter undefined instructions.
  - Trap instruction (SVC - ARM, INT – x86); a program invokes a service OS, running at a higher privilege level.

# System call

Lancaster University

- A **system call** allows programs to request a service from the kernel.
  - It's the **interface** between user programs (in user space) and the OS (in kernel space).
  - Linux offers a couple of 100's system calls: open, write, read, exit …
- The OS at boot registers the address of a system call handler function with the CPU
- A process executes the trap instruction
- The process context is stored in memory, the state of the OS is loaded and the CPU executes the call handler function (see previous slide)

# x86 System Calls

Lancaster University

Show 10 entries Search:

| # | Name | Registers | | | | | | | Definition |
|---|---|---|---|---|---|---|---|---|---|
| | | eax | ebx | ecx | edx | esi | edi | ebp | |
| 0 | restart_syscall | 0x00 | – | – | – | – | – | – | kernel/signal.c:2501 |
| 1 | exit | 0x01 | int error_code | – | – | – | – | – | kernel/exit.c:1095 |
| 2 | fork | 0x02 | – | – | – | – | – | – | arch/x86/kernel/process.c:271 |
| 3 | read | 0x03 | unsigned int fd | char *buf | size_t count | – | – | – | fs/read_write.c:460 |
| 4 | write | 0x04 | unsigned int fd | const char *buf | size_t count | – | – | – | fs/read_write.c:477 |
| 5 | open | 0x05 | const char *filename | int flags | umode_t mode | – | – | – | fs/open.c:1046 |
| 6 | close | 0x06 | unsigned int fd | – | – | – | – | – | fs/open.c:1117 |
| 7 | waitpid | 0x07 | pid_t pid | int *stat_addr | int options | – | – | – | kernel/exit.c:1879 |
| 8 | creat | 0x08 | const char *pathname | umode_t mode | – | – | – | – | fs/open.c:1079 |
| 9 | link | 0x09 | const char *oldname | const char *newname | – | – | – | – | fs/namei.c:3152 |
| 10 | unlink | 0x0a | const char *pathname | – | – | – | – | – | fs/namei.c:2979 |
| 11 | execve | 0x0b | const char *name | const char *const *argv | const char *const *envp | – | – | – | arch/x86/kernel/process.c:342 |
| 12 | chdir | 0x0c | const char *filename | – | – | – | – | – | fs/open.c:375 |
| 13 | time | 0x0d | time_t *tloc | – | – | – | – | – | kernel/time.c:62 |
| 14 | mknod | 0x0e | const char *filename | umode_t mode | unsigned dev | – | – | – | fs/namei.c:2693 |
| 15 | chmod | 0x0f | const char *filename | umode_t mode | – | – | – | – | fs/open.c:499 |
| 16 | lchown | 0x10 | const char *filename | uid_t user | gid_t group | – | – | – | fs/open.c:586 |

https://syscalls.w3challs.com/?arch=x86

# Problem 2: CPU sharing

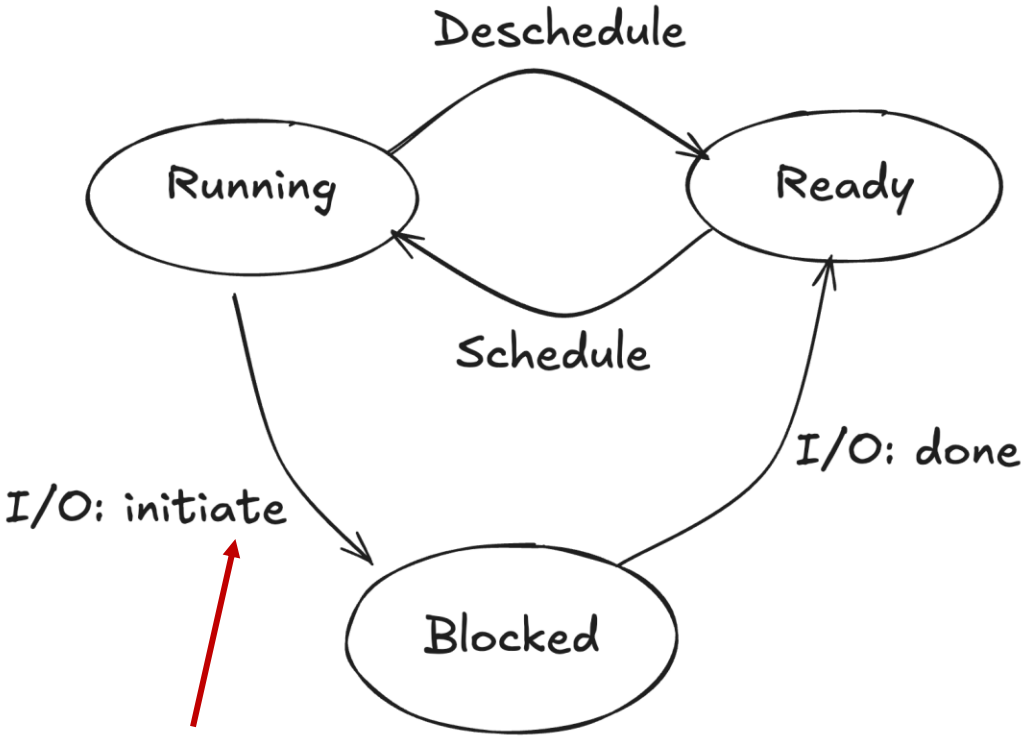
Lancaster University

- If a process runs (i.e., uses the CPU), the OS cannot context switch to share the CPU.
  - *How can the operating system regain control of the CPU so that it can switch between processes?*
- Cooperative scheduling: Long-running processes periodically give up the CPU for the OS to run other tasks.
- Non-cooperative schedule: Schedule the OS over fixed intervals (scheduling quantas) using a timer interrupt.

# Process states

```
top - 12:22:24 up 85 days, 21:26,  4 users,  load average: 0.36, 0.20, 0.13
Tasks: 344 total,   1 running, 343 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.1 sy,  0.0 ni, 99.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  31779.4 total,   1399.9 free,   2895.6 used,  27483.9 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  27698.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1290 root      20   0 3118700  59168  24968 S   6.0   0.2   5816:04 kubelet
20879 root      10 -10 2005512 235888  12528 S   1.3   0.7  10:01.74 ovs-vswitchd
21040 root      20   0 6687464 902008  28176 S   1.3   2.8  36:33.54 qemu
20161 root      20   0 4191320 126976  63104 S   1.0   0.4  49:41.41 dockerd
 5688 root      20   0 1257972  43348  24204 S   0.3   0.1   0:00.53 tailscaled
 6282 tudor     20   0   11088   4632   3752 R   0.3   0.0   0:00.24 top
20692 root      20   0 1597720 147884 106016 S   0.3   0.5   8:07.01 grafana
20721 root      20   0 2710932  80904  49676 S   0.3   0.2 151:22.39 promtail
    1 root      20   0  381372  14112   8552 S   0.0   0.0 178:58.70 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:16.38 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
    5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
    8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_hig
   10 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
   11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
   12 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
   13 root      20   0       0      0      0 S   0.0   0.0   6:10.41 ksoftirqd/0
```




Processes can block because they expect IO to complete from a computer device.

# Limited direct execution (Boot)

Lancaster University

| OS @ boot | Hardware | Program |
| --- | --- | --- |
| initialize trap table | | |
| | remember address of...<br>syscall handler<br>timer handler | |
| Start interrupt timer | | |
| | Start timer<br>Interrupt CPU in X msec | |

# Limited direct execution (Start process)

Lancaster University

| OS@ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| Create entry for process list<br>Allocate memory for program<br>Load program into memory<br>Setup user stack with argv<br>Fill kernel stack with reg/PC<br>return-from-trap | | |
| | restore regs (from kernel stack) move to user mode<br>jump to main | |
| | | Run main()<br>… |

# Limited Direct Execution (System Call)

Lancaster University

| OS@ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | | Call system call<br>trap into OS |
| | save regs (to kernel stack)<br>move to kernel mode, jump to trap handler | |
| Handle trap<br>Do work of syscall<br>**return-from-trap** | | |
| | restore regs (from kernel stack) move to user mode jump to PC after trap | |
| | | ...<br>return from main trap (via exit()) |
| **Free memory of process Remove from process list** | | |

# Limited Direct Execution (Timer interrupt)

Lancaster University

| OS@ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | | Process A |
| | **timer interrupt**<br>save regs(A) to k-stack(A)<br>move to kernel mode, jump to timer handler | |
| Handle the trap<br>Call `switch()` routine<br>save regs(A) to proc-struct(A)<br>restore regs(B) from proc-struct(B)<br>switch to k-stack(B)<br>**return-from-trap (into B)** | | |
| | *restore regs(B) from k-stack(B)*<br>*move to user mode, jump to B's PC* | |
| | | Process B |

# Conclusion

Lancaster University

- An **Operating System** is a software layer that sits between hardware and user applications.
  - Manage resources
  - Safe and convenient programming abstractions
  - Services: scheduling, device drivers
- CPU virtualization allows multiple processes to share a physical CPU.
- Limited direct code execution allows the OS scheduler to control process scheduling.
  - Processes use system calls to access virtualized resources.
  - Timer interrupts enable the OS to schedule processes.
- Next: Memory Virtualisation

# Further reading

Lancaster University

- Chapter 4: Processes
- Chapter 5: Process API
- Chapter 6: Direct Execution