

ECS518U - Operating Systems

Week 3

Processes (part B)

Tassos Tombros

Outline

- Review from Week 2
 - System Calls
 - Context switch
- Process scheduling
- Process Control using PHP
 - fork, wait, exec, signals
- Assessed lab(s) info
- **Reading:**
 - **Stallings:** Chapter 3, parts of Chapter 9
 - **Tanebaum:** Chapter 2, sections 2.1, 2.4

Things you will learn today

- **What** are some of the **aims** of process scheduling
- **How** some specific scheduling methods work and **what** are their strengths and problems
 - FCFS, Round Robin, Priority queues, Multi-level feedback queues
- **How** can a process create a process – **fork()**
- **How** can we manage processes using **pcntl extensions** in PHP

Process Scheduling: Story So Far

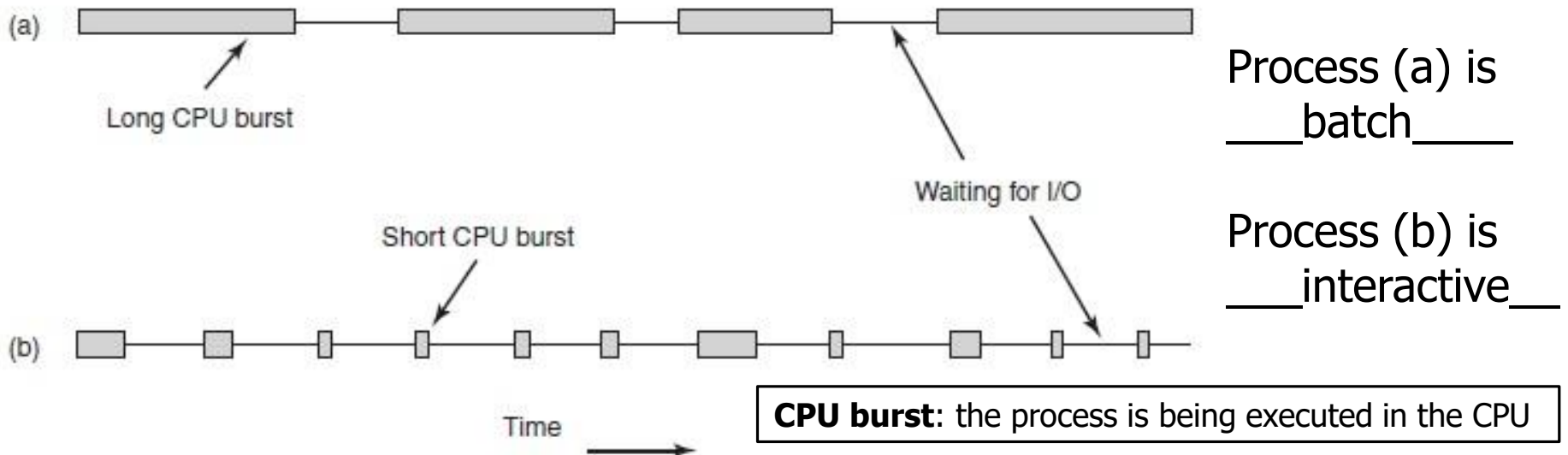
- The OS runs a number of processes
- At any time:
 - Some are ready to run
 - Some are not ready to run (e.g. maybe waiting for I/O)
 - **one is actually running** (per processor)
- How does the OS choose
 - **When** to switch to another process?
 - **Which** process to choose next from among the ones that are ready to run?
- The problem is called **process scheduling**
 - Good analogy: dealing with multiple courseworks
 - **Scheduling can apply not only to processes** but also to: disk requests, network queues, printer jobs

Process Scheduling: Aims

- **Response time:** we want programs to be responsive if we interact with them so scheduling should allow short response times from programs
 - Minimise avg. and max. wait time
- **Throughput: Maximise** the amount of processes completed in a given time frame
- **Efficiency/Utilisation:** To use the resources efficiently (e.g. avoiding idle time for the CPU)
- Prevent **process starvation**
 - Never have a process that does not get executed or is denied resources for too long
- Share time ***fairly*** among processes
 - “Batch” jobs: (lots of computations) throughput matters most
 - Interactive: response time matters most

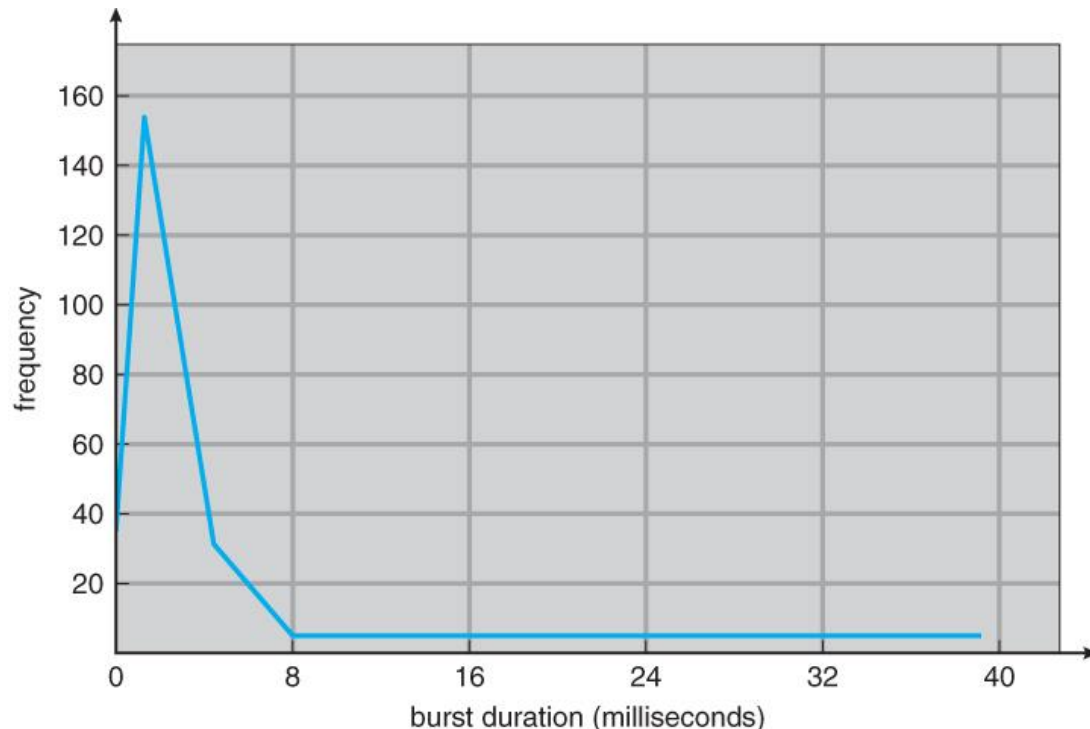
Types of processes

- **Batch processes** (CPU intensive)
 - Computationally intensive, throughput matters most (last digit of π)
- **Interactive processes** (I/O intensive)
 - Response time matters most (e.g. a process that keeps stopping for I/O, perhaps with the user)
- Processes may **change** characteristics over time
- **(Real time)**
 - Guarantees that deadlines will be met (press the brakes before you hit the wall)



Distribution of CPU bursts

- **What does this graph show us?**
 - The vast majority of processes have short CPU bursts
- **What 'types' of processes are we more likely to find on the left vs. the right of the distribution?**
 - Interactive on the left vs. batch (computationally intensive) on the right

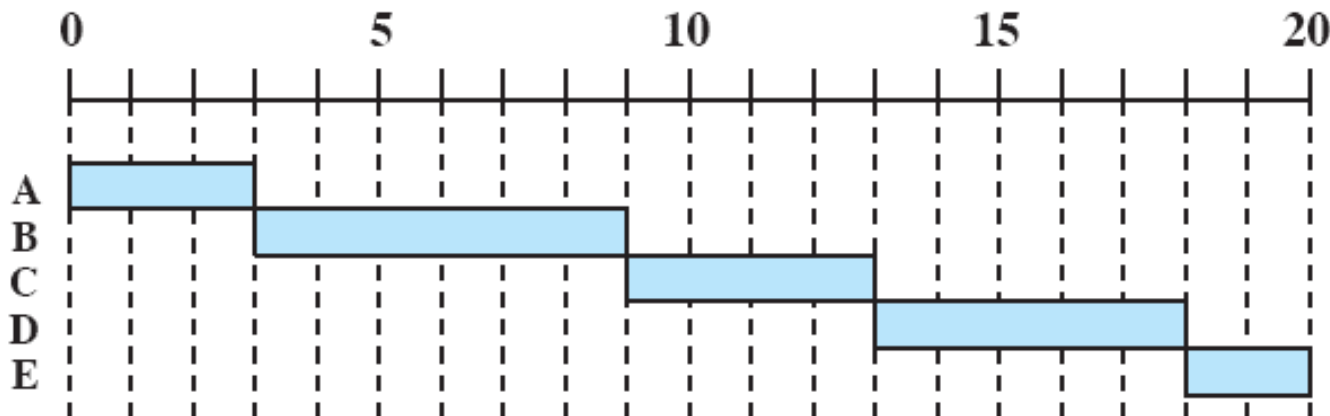


What is the significance of this?

- Modern schedulers exploit this: maybe we want to give precedence to short processes to minimise response times
- The ones that are computationally heavy (right side) will probably not 'notice' that they get interrupted


First-Come-First-Served (FCFS or FIFO)

- Processes are assigned the CPU in the order they request it
- A single Ready queue is used**, a newly arrived process is put on the end of the queue
- The process running is not interrupted because it has run too long
 - It can give up the CPU voluntarily or if interrupted by I/O
 - FCFS favours CPU-intensive processes (little I/O to do so they keep hold of the CPU)
- Problem:** A short process (such as E) waits for longer ones to complete
- It is **simple to implement** (simple linked list queue)
- It works better if all processes are of the same (or similar) lengths



Illustrating the problem with FCFS

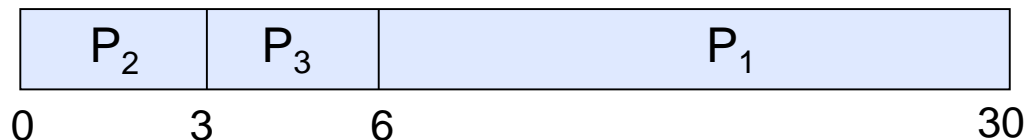
Assume 3 processes P_1 , P_2 , P_3 with burst times:

Process	Burst Time	
P_1	24	
P_2	3	
P_3	3	

Suppose processes are scheduled in the order: P_1 , P_2 , P_3 - the Gantt Chart for the scheduling is as above

- **Waiting time** for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- **Average waiting time:** $(0 + 24 + 27)/3 = 17$
- **Average Completion time:** $(24 + 27 + 30)/3 = 27$

Suppose now that processes are scheduled in the order: P_2 , P_3 , P_1 - now the Gantt chart for the scheduling is:



→ Known as Shortest Job First (SJF)

- **Waiting time** for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- **Average waiting time:** $(6 + 0 + 3)/3 = 3$
- **Average Completion time:** $(3 + 6 + 30)/3 = 13$

Non-preemptive vs. Preemptive scheduling

- **Non-preemptive**

- Once a process is in the running state it continues until
 - completed
 - blocks for I/O (or voluntarily gives up the CPU)
- FCFS is a non-preemptive scheduling method

- **Preemptive**

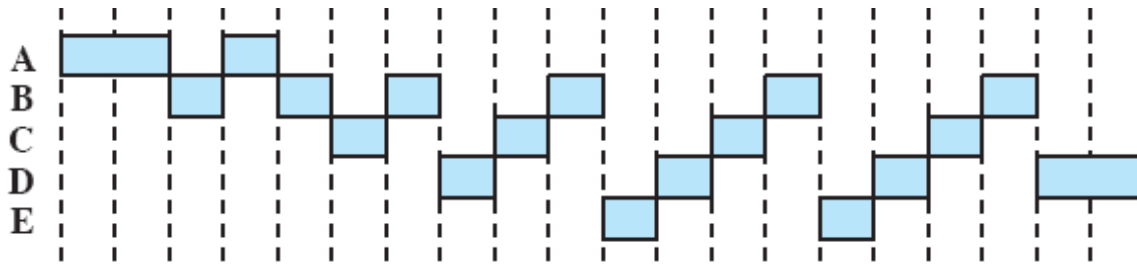
- The currently running process may be interrupted
- Preemption may occur:
 - when a new process arrives
 - on an interrupt
 - periodically based on a timer

Round Robin (RR)

- Uses preemption based on a clock
 - also known as time slicing – each process is given a time slice before being preempted
- Periodic clock interrupt
 - (e.g. in the example below $q=1$ (time quantum))
- Current process then moves to back of queue
 - Next ready process is selected
- **RR is better for shorter jobs**
 - also when jobs are of varying lengths
 - more fair than FCFS

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Round-Robin
(RR), $q = 1$

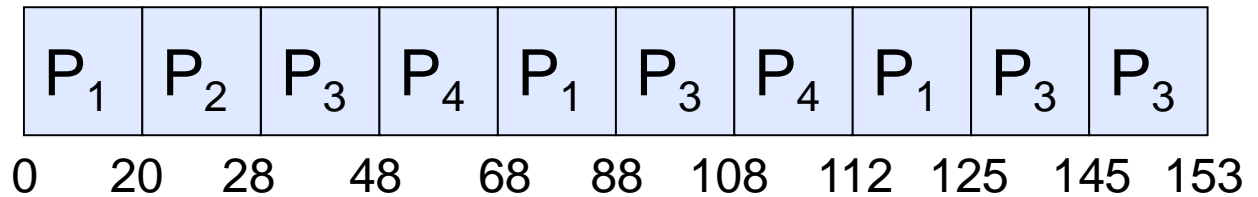


Example of RR with $q=20$

Example:

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	8
P_3	68
P_4	24

– The Gantt chart is:

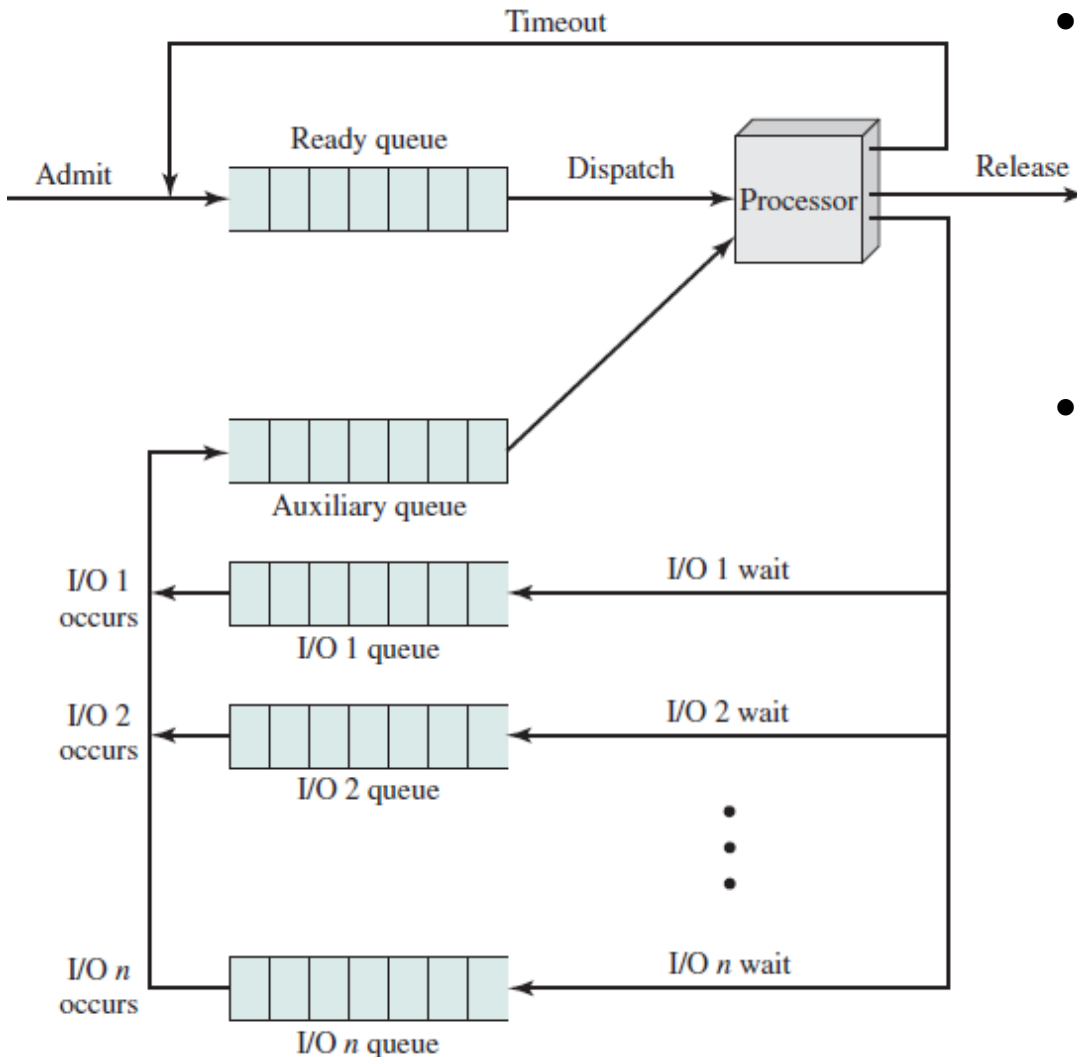


- **Waiting time** for $P_1 = (68-20) + (112-88) = 72$
 $P_2 = (20-0) = 20$
 $P_3 = (28-0) + (88-48) + (125-108) = 85$
 $P_4 = (48-0) + (108-68) = 88$
- **Average waiting time** = $(72+20+85+88)/4 = 66\frac{1}{4}$
- **Average completion time** = $(125+28+153+112)/4 = 104\frac{1}{2}$

Preemption Time Quantum Size

- How long should we make the time quantum (q)?
- If we make it too long?
 - Most processes do I/O at some point so would they use the whole quantum anyway?
 - Computationally heavy processes would be favoured
 - **Poor responsiveness**
- If we make it too short?
 - **Overhead of context switching**, so big proportion of time is spent on switching and not on 'getting work done'
- Typical values can be found in the order of **10ms-100ms**
 - Typical overhead for context switch is 0.1 – 1 ms
 - Roughly 1% overhead due to context switching

'Virtual Round Robin'

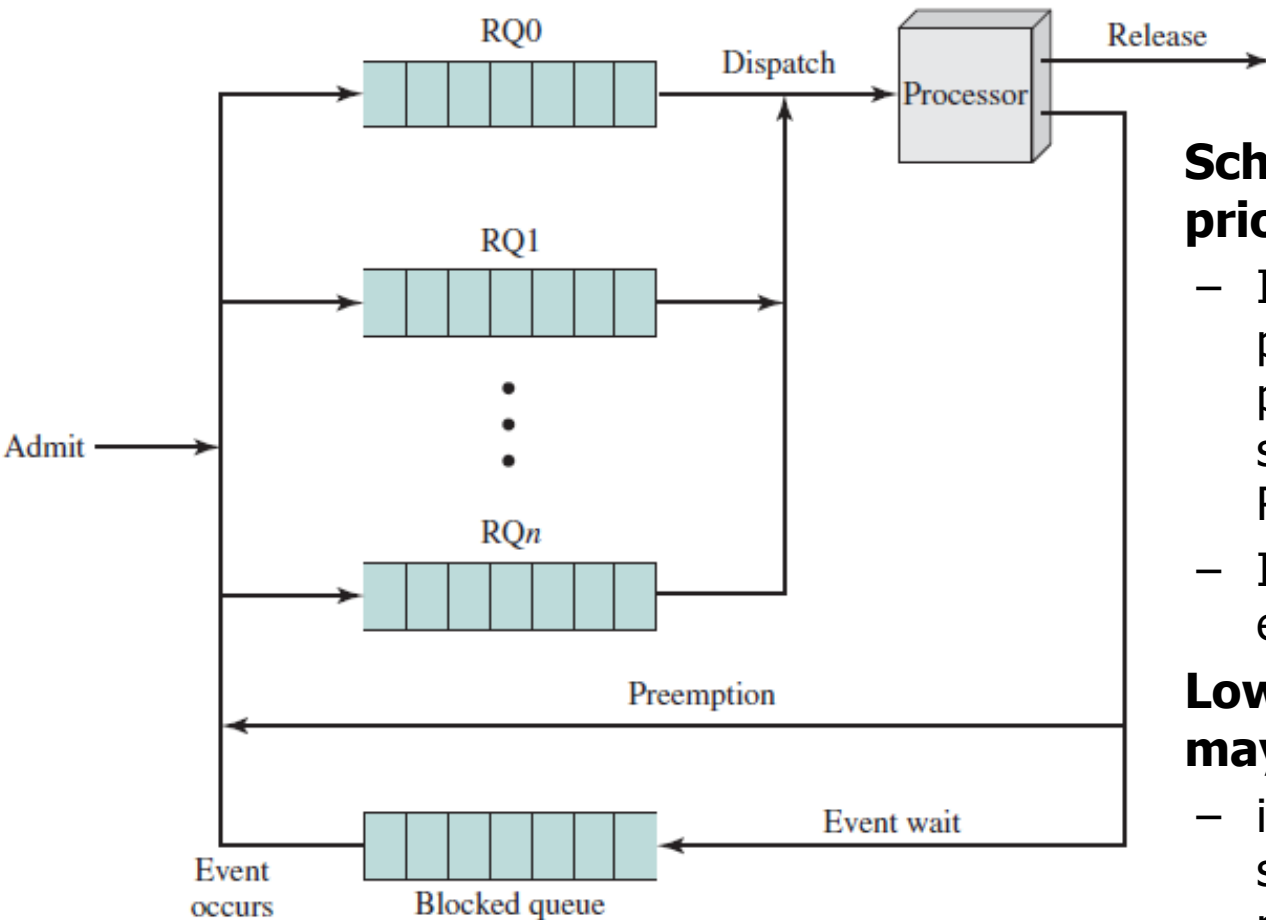


- As per usual:
 - New processes arrive and join the ready queue
 - When a process times out it is returned to the ready queue
 - When a process is blocked for I/O it joins an I/O queue
- New feature: **auxiliary queue**
 - Processes are moved there after being released from an I/O block
 - Processes in the auxiliary queue get preference over those in the main ready queue
 - **BUT** when a process is dispatched from the auxiliary queue, it runs no longer than a time equal to the time quantum minus the total time spent running since it was last selected from the main ready queue (so not for the whole q, but for the remaining q)

Scheduling with Priorities

- Processes are assigned priority values
- Scheduler always chooses a process of higher priority **BEFORE** one of lower priority
- Priorities can be static or dynamic
 - With **dynamic priorities** the OS can adjust priority up or down based on heuristics about interactivity, burst behavior, etc.
 - For example the Unix scheduler dynamically updates priorities in regular intervals based on heuristics
- There are **multiple ready queues** to represent each level of priority

Priority Queuing



Scheduler starts at highest priority queue (RQ0)

- If there are one or more processes in the queue, a process is selected using some scheduling policy (e.g. Round Robin)
- If RQ0 is empty, then RQ1 is examined, etc.

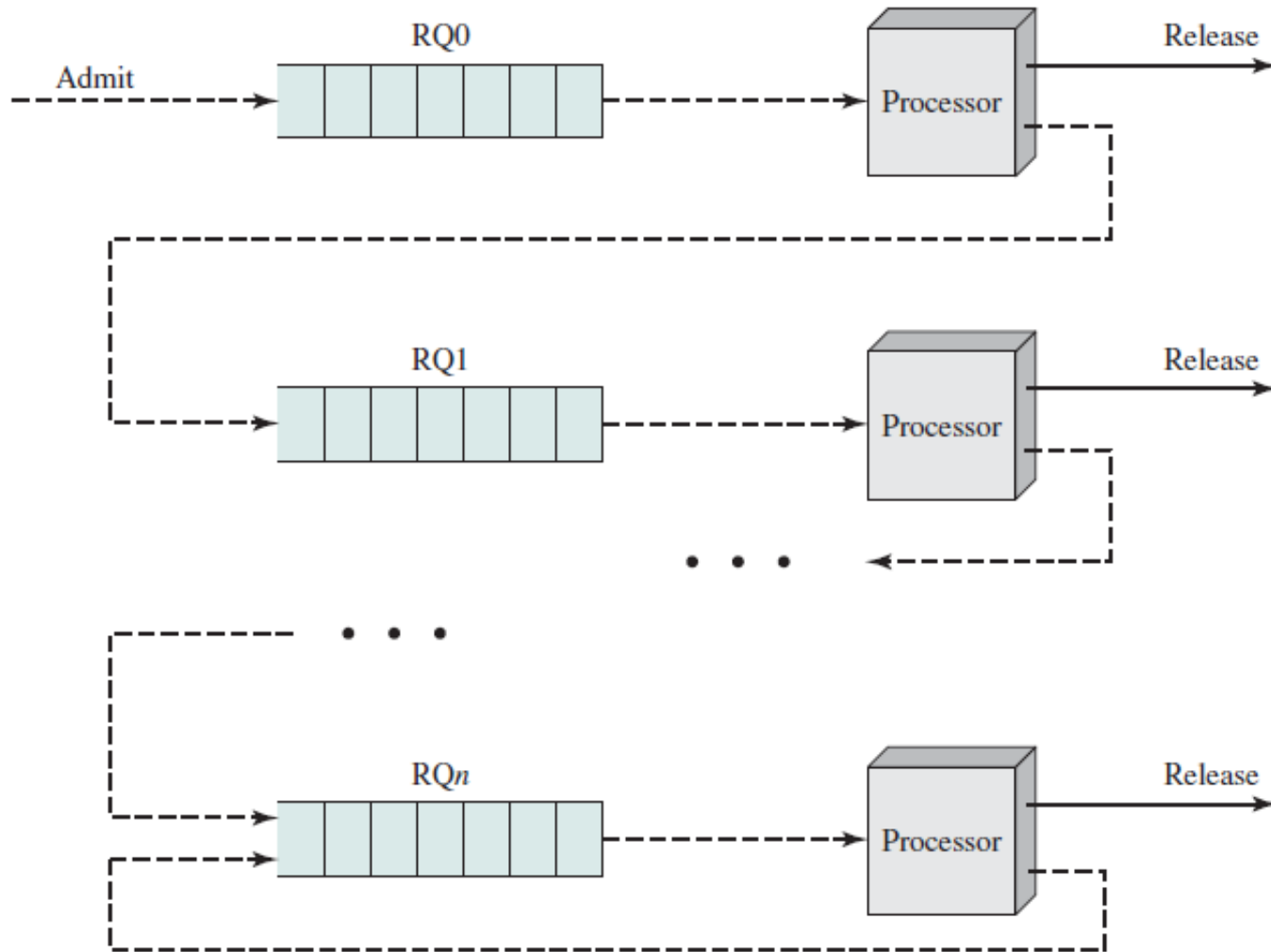
Lower-priority processes may suffer starvation

- if there is always a steady supply of higher-priority ready processes
- the priority of a process can change with its age or execution history (Unix does this)

Multi-Level Feedback Queues

- Variations of this method are used in many desktop OSs
- Multiple priority queues (usually FCFS or RR)
 - A process starts at the top level of priority
 - **On pre-emption: go down a level**
 - (Optionally) On I/O block: go up a level
- Preference to newer and shorter jobs and to I/O bound processes
- Turnaround time of longer processes can stretch out
 - **possible for starvation to occur** if new jobs are entering the system frequently
- To compensate for this **we can vary the preemption times according to the queue**
 - For example a process scheduled from RQ_i is allowed to execute 2^i time units before preemption

Multi-Level Feedback Queues



Some interesting issues

- **How do we evaluate scheduling algorithms?**
 - Most commonly build system which allows actual algorithms to be run against actual data (implement / simulate)
 - Generate measurements on performance metrics and compare
- **Scheduling algorithms that 'know' remaining time of all processes are best**
 - e.g. preemptive: Shorted Remaining Time First (SRTF), non-preemptive: Shortest Job First (SJF)
 - But how can we know in advance how much time a process needs to finish?
 - Mathematical models for approximating based on previous bursts (time series)
- **Scheduling for multi-core architectures**
 - Recent research revealed bugs in the Linux scheduler w.r.t idle times on cores (<https://blog.acolyer.org/2016/04/26/the-linux-scheduler-a-decade-of-wasted-cores/> with link to the research paper)

Lab Feedback

- Do not be afraid to edit code, run it and see what happens
 - What is the worst that can happen?
- You **MUST** develop the curiosity to find things out for yourself
 - PHP functions, Linux commands, etc. etc.
 - If you are not sure what strings you are reading e.g. from command line, print them to screen to see, etc.
- **argc, argv**
 - What was the main benefit of using command line arguments in the last task?
- **ps** is a very useful command to use, also for Lab 3 and more
- **Sample solutions** will be posted on QMPlus with a small delay because of Thursday labs

Process Control

- **Can a process create a process?**
- Yes! Unique identity of each process is the “process ID” (or **PID**)
- The `fork()` library call creates a *copy* of current process with a new PID. The new process is the **child** process and is created by the **parent** process
- Return value from `fork()` : integer
 - When > 0 :
 - Running in (original) **Parent** process
 - return value is **pid** of new child
 - When $= 0$:
 - Running in new **Child** process
 - When < 0 :
 - Error! Must handle somehow
 - Running in original process
- **All state of original process is duplicated in both Parent and Child**
 - Memory, File Descriptors, etc...

PHP – Process Control Extensions

- PHP provides interface to standard UNIX library calls
- The **PCNTL – process control** – extensions provide PHP functions that call POSIX system calls:
 - **pcntl_fork** – system call to create a copy of the current process, and start it running
 - **pcntl_exec** – system call to *change the program* being run by the current process
 - **pcntl_wait** – system call to wait for a process to finish
 - **pcntl_signal** – system call to send a notification to another process
- All these PHP functions correspond to standard UNIX calls, e.g. fork(), exec(), wait(), signal()
- **Use the PHP manual** for the specifics of functions

pcntl_fork – create a new process

```
<?php
```

```
// fork creates a copy of the current process
// the process id of the child process is returned
//     the 'pid' variable is not assigned in the child variable
//     the 'pid' is -1 if the fork fails
$pid = pcntl_fork();
```

```
// both processes continue to this point
```

```
if ($pid == -1) {
    die("could not fork\n");
} else if ($pid) {
    // we are in the parent process, child has PID=$pid
    ...
} else {
    // we are in the child process, child's PID can be
    // obtained by posix_getpid()
    ...
}
?>
```

Example using pcntl_wait

```
$pid = pcntl_fork();
if ($pid == -1) {
    die("could not fork\n");
} else if ($pid) {
    // we are the parent
    print("Parent: child process pid=: $pid \n") ;

    // wait for the child process to exit
    // the 'status' variable is updated with the status
    pcntl_wait($status);

    // this function gets the exit code from the status
    $exitCode = pcntl_wexitstatus($status) ;
    print ("Child exit code is: $exitCode \n") ;
} else {
    // we are the child
    print("Child: my pid=") ;

    // this function get the PID
    print posix_getpid() ;
    print "\n";
    exit (1) ;
}
```

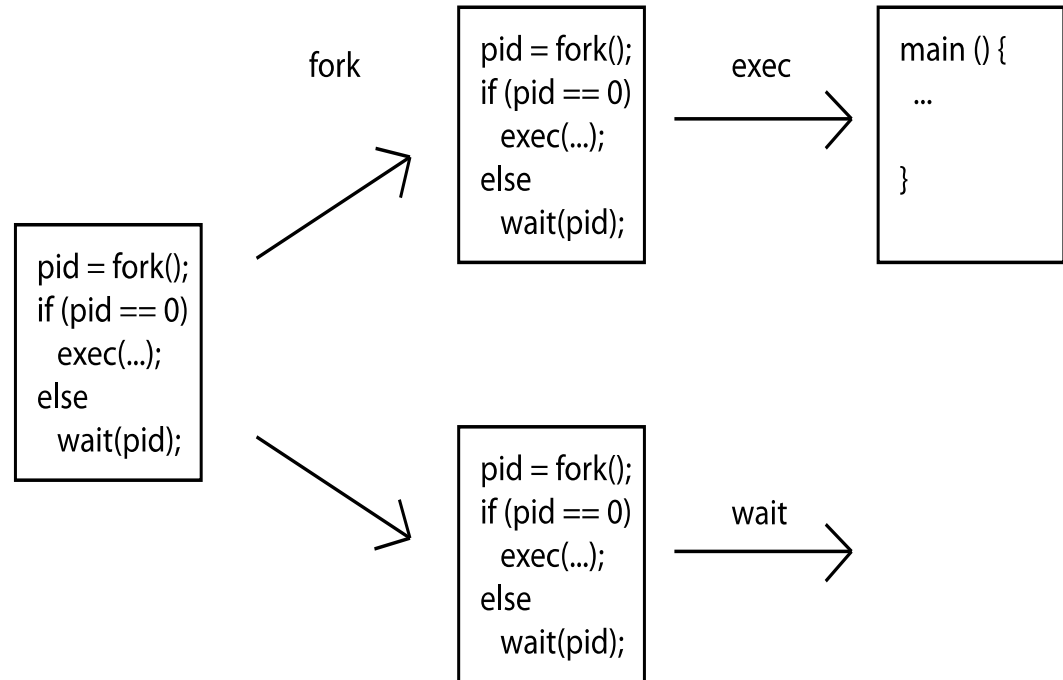
 No zombies

fork() – Copy On Write

Normally upon `fork()` all resources owned by the parent are duplicated and the copy is given to the child.

This is **inefficient**: it copies much data that could be shared.

Also, very often the child process immediately executes a new image by calling **`exec()`**, so all that copying goes to waste (since the new image is now different).



Copy on Write (COW) delays or altogether prevents copying of the data. Rather than duplicate the process address space, parent and child can share a single copy. **The duplication occurs only when there is attempt to write on the shared copy;** until then, it is shared read-only. **COW** delays the copying of each page in the address space until it is actually written to. For example, if `exec()` is called immediately after `fork()` we never need to copy. This happens every time you type a command in a shell.

pcntl_signal example

```
declare(ticks=1) ;

pcntl_signal(SIGINT, "sig_handler") ;
while (1)
{
}

function sig_handler($signum) {
    echo("Phew, signal caught, killing process...\n");
    exit(1) ;
}
```

declare(ticks=1)

the PHP engine needs to be instructed to check for signals and run the pcntl callbacks. To allow this to happen, you need to set the execution directive ticks. ticks instructs the engine to run certain callbacks every N statements in the executor, so setting `declare(ticks=1)` instructs the engine to look for signals on every 1 statement executed. Without this line your code will not catch signals.

Process races

```
else if ($pid) {  
    // we are in the parent process  
    print("Parent: child process pid= $pid \n") ;  
  
    for ($i=0; $i<100; $i++) {  
        echo "Parent, $i\n";  
        //sleep(1);  
    }  
} else {  
    // we are in the child process  
    for ($i=0; $i>-100; $i--) {  
        echo "Child, $i\n";  
        //sleep(1);  
    }  
}
```

Question: What does this program print?
Does it change if you add in one of the **sleep()** statements?

Lab 3: First Assessed Lab

Make sure you know the **rules for assessment** in the ITL (see QMPlus)

- Write PHP scripts that make use of process control system calls (inc. `exec` and `fork`)
- Answer questions about differences between such system calls
- **Your assessment:**
 - Run code, explain code, answer questions about code
 - Answer / explain anything else that is asked on the answer sheet
 - **Readability / comments** of code count in your assessment
- There will be no time to do all the work during the lab – **you MUST do the work before you come in for assessment** – Please read the information on 'rules for assessed labs' on QMPlus

To get you started

- **Sample code** that provides some skeleton functionality is available (lab section QMPlus)
- Sample code with the use of `fork` and `exec` is also available (lab section QMPlus)
- Some extra notes on fork, zombies, signals, etc. (pdf, Lectures on processes section QMPlus)

NOTE:

- **To get the highest grades** (A, A*) you must demonstrate excellent (deep) understanding of the concepts involved, not just the ability to repeat some information you found on Wikipedia, etc.
- To achieve this, **you must spend time trying things** out and to have the curiosity to look for information from different sources and to actually understand it (rather than just memorising it)
- Because of this, inevitably people with 'similar' code solutions will be awarded different grades depending on how they perform in the oral examination during the assessment

Summary

- **Process:** instance of a running program
 - **Process context:** data OS has about a process
 - **Process state:** OS keeps track of state of each process: running, etc.
- **Protection:** user and kernel modes to prevent processes interfering
- **Syscalls:** secure programming interface between apps and OS
- **Context switch:** changing the running process
- **Scheduling:** choosing which (ready) process to run
 - Challenge to meet the different scheduling aims
 - Different process types (I/O vs. CPU –heavy processes)
 - Different scheduling methods favour different types of processes
- **Process control**
 - fork, exec, wait, signal, ...
 - Process family trees (see `pstree`)
 - `fork()` implementation with Copy On Write