3 – Processes

3 – Processes

- To understand the concept of processes
- Process operations
- Process management

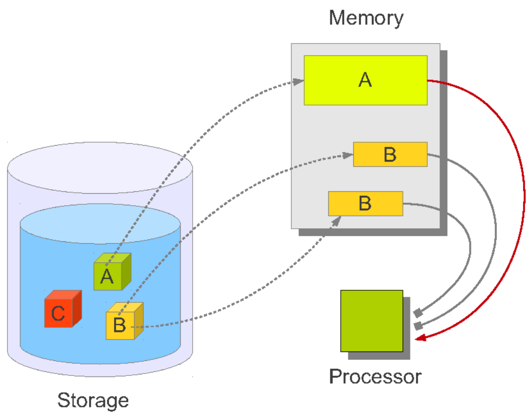
Processes ?

- Early computer systems allowed only one program to be executed at a time.
- Today computer systems allow multiple programs to be loaded into memory and to be executed *concurrently*.
- This requires the notion of a what to call all the CPU activities
  - A batch system executes *jobs*.
  - A time-sharing system has *user programs* or *tasks*.
  - Most of us called them *programs*.
  - ...

(cont'd.)

- All of these are very similar. We even use these terms interchangeably. To make it more clearer, let's call all of them *processes*.
- The most widely accepted definition is that "*a process is a program in execution*".
  - Note that a program itself is **not** a process since it is a passive entity stored on a storage device.
- The CPU executes one instruction after another until the process completes.

(cont'd.)



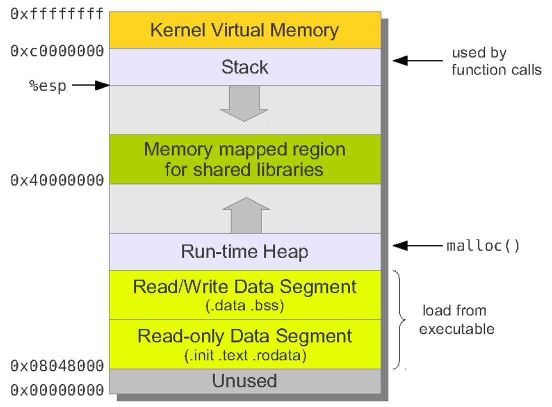
(cont'd.)

- A process is the unit of work in a modern time-sharing system.
  - It is more than the program code, i.e., the *text section*.
  - It includes *stack*, which contains temporary data, and *data section*, which contains global variables.
  - It also includes current activity represented by the value of the program counter and the content of the processor's registers.
  - Additionally, a process may hold a number of resources, e.g,
    - I/O devices (keyboard, mouse, ...)
    - Files
    - Network connections

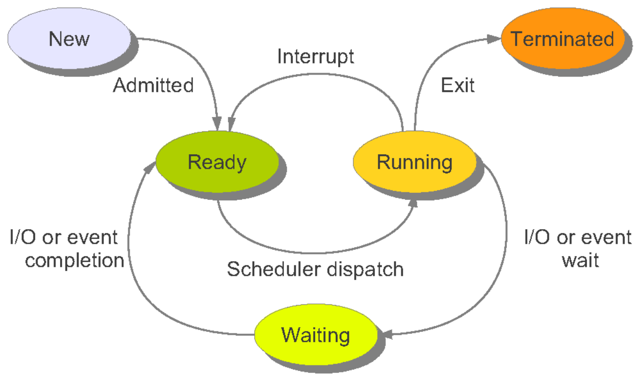
(cont'd.)

- Two processes may be associated with the same programs, but they are considered two separate execution sequences.
  - The text sections are equivalent, but the data sections differ.
  - In some systems, the text sections may be shared among the processes.

(cont'd.)



Process States

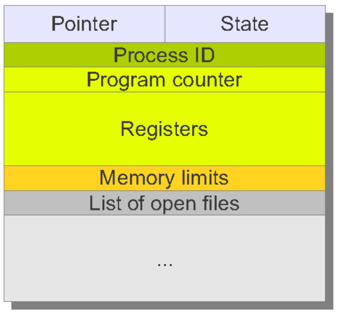


Context Switches

- The states imply that a process does not always run on a processor.
  - More or less, it has to perform I/O operations. That means the processor is idle, and another process may utilize it.
- When a processor switches from one process to another, the operating system must save states of the old process, and load saved states of the new process. This is called *context switch*.
- The context (the process state) is represented in the *Process Control Block*.

Process Control Blocks

- A data structure containing all information required by the OS to execute the process properly, e.g.,



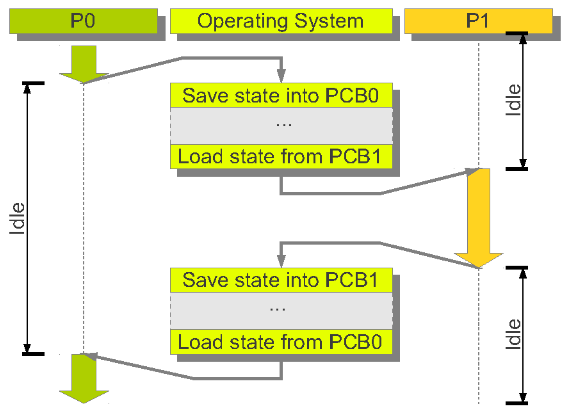
(cont'd.)

- **Process state**: new, ready, running, ...
- **Program counter**: the address of the next instruction to be executed for this process
- **Registers**: state of processor's registers
- **CPU-scheduling information**: priority, pointer to scheduling queue, other scheduling parameters
- **Memory-management information**: base and limit registers, page or segment table, ...
- **Accounting information**: CPU usage and limit, account number, process number, ...

(cont'd.)

- **I/O-management information**: I/O device allocations, list of open files, ...
- For example, on Linux, this is the `task_struct`, defined in `include/linux/sched.h`.
  - It is quite a large structure, the definition alone is about 300 SLOCs, containing many other structures
  - The register part is in the `thread_struct`,
    - In `arch/*/include/asm/processor.h`
  - Let's see the source.

Switching from one process to another



(cont'd.)

- On Linux, `schedule()` calls `context_switch()` to switch context.
  - Described in in `kernel/sched.c`
  - Literally, `context_switch()` has two parts
    - Memory management part to prepare memory for the new context.
    - Architecture-dependent part, `switch_to()`, to prepare registers and stack so that the new context could start.

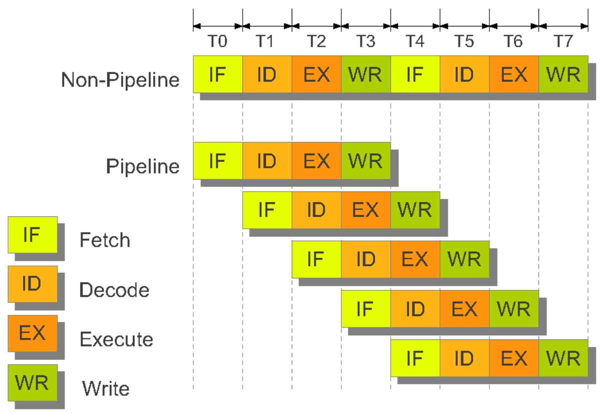
Process Scheduling

- So, the context switch is a **pure overhead**, totally wasteful, and becomes performance bottleneck.
  - Can we avoid context switch ?
  - If processors > processes ?
- Optimally, we want to maximize utilization and interactivity.
  - The objective of *multiprogramming* is to have some process running at all times.
    - Maximize CPU utilization → less context switch.
  - The objective of *time-sharing* is to switch among processes frequently.
    - More interactivity → more context switch.
  - On Linux, see `vmstat`.

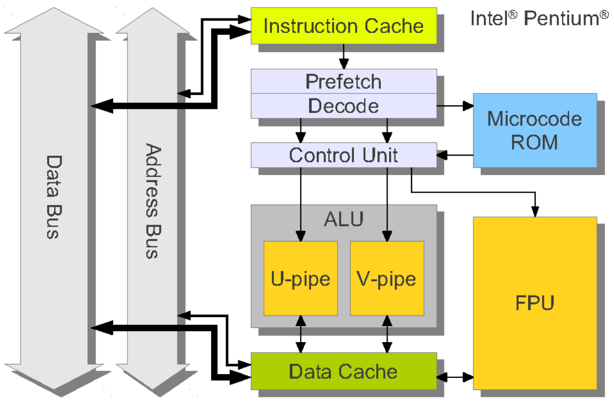
(cont'd.)

- On a **uniprocessor** system,only **one** process can be running at a time.
  - Because there is only one processing unit ?
  - Yes, there are parallelism techniques like, e.g.,
    - Instruction pipeline
    - Superscalar
    - Intel® Hyper-Threading™ technology
    - Multi-core technology
    - ...
  - Doesn't that allows us to execute multiple instructions at a time ?
    - Does it allow us to run multiple processes at a time ?
    - Let's see ...

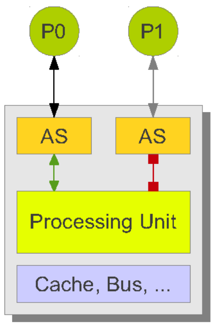
Pipelines



Superscalar

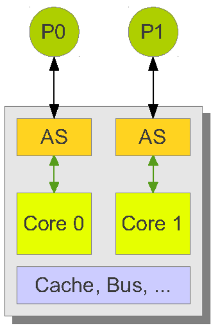


Intel® Hyper-Threading™ Technology



- An Architecture State (AS) holds states of the process in execution.
  - Intel HT™ processors contain multiple ASes so that a processor could hold multiple processes at a time.
  - This creates an *illusion* image of multiprocessors.
- Multiple ASes increase the switching performance.
  - Sacrifice 5% in size.
  - Gain 30% in performance.

Multi-core Processors



- Multiple processing units in a single package.
  - It may share other units in a chip, e.g., caches, buses, ...
- Intel Core™, Core 2™
- AMD Athlon™ X2
- Sony PS3's cell processor
  - 1 x 64-bit POWER
  - 8 x 128-bit SIMD RISC
- In 2006, Intel has showed a wafer of 80-core processors.
  - Commercialized in 5 years.

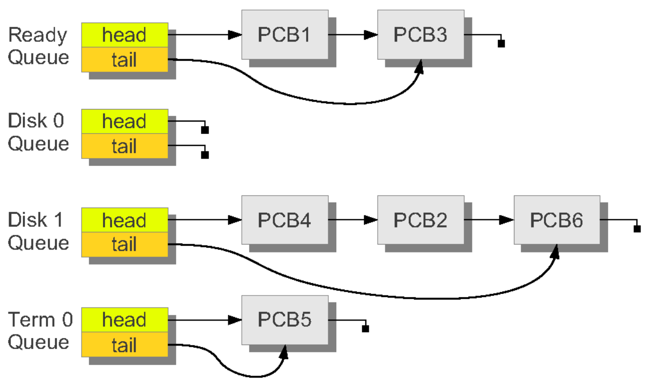
Levels of Parallelism

- Pipelines and superscalar are *instruction-level parallelism*.
  - There are also techniques like branch prediction, out-of-order execution, ...
  - It's that two or more instructions **in the same context** can be executed in parallel, not multiple processes.
- Those Intel HT™ or multi-core technologies allow a processor to hold multiple processes at a time.
  - These are *multiprocessing* or *multithreading* – a much higher-level of parallelism.
- Can process scheduling exploit instruction-level parallelism, multiprocessing, or multithreading ?

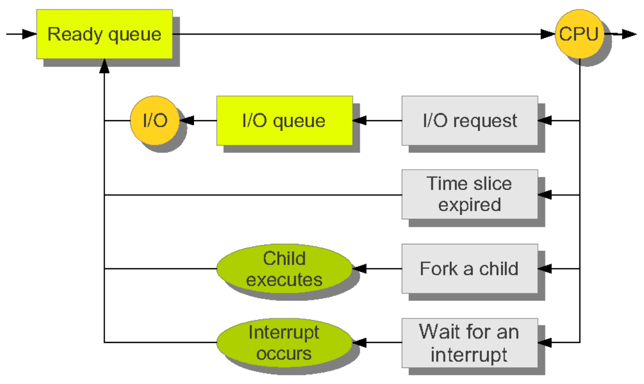
Scheduling Queues

- *Job queue* contains all processes in the system.
- *Ready queue* contains all processes residing in the memory, and are ready to execute.
- *Device queue* contains all processes waiting for a particular I/O device.
  - Each device has it own queue.

(cont'd.)



Queuing Diagram



Schedulers

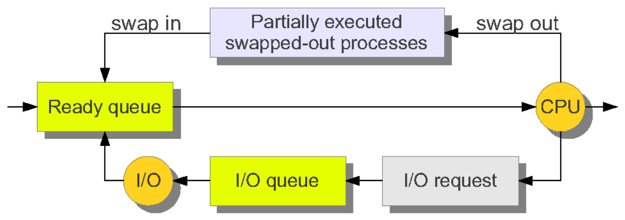
- A component to select processes from queues in some fashion.
- *Short-term scheduler* (or CPU scheduler) selects a process from ready queue to be executed next.
  - It is executed frequently, e.g., once every 100 msec, so it must be fast.
  - If it takes 10 msec, then it takes 10/(10 + 100) ~ 9% of CPU time.
- *Long-term scheduler* (or job scheduler) selects processes spooled in a mass-storage device, and loads them into memory (ready queue) for execution.

(cont'd.)

- The long-term scheduler executes much less frequently, e.g., minutes. This kind of scheduler controls the *degree of multiprogramming* – the number of processes in memory.
- The process can be
  - I/O-bound process
  - CPU-bound process
- The long-term scheduler must select a good mix of I/O-bound and CPU-bound processes.
- The long-term scheduler can be minimum or event absent, e.g., UNIX.

(cont'd.)

- Some systems introduce the *medium-term scheduler* to remove processes from memory.
  - The process may be reintroduced into memory and its execution can be continued.
  - This scheme is called *swapping*.



Creating a Process

- Processes are like human: they are born, live, may give a birth to children, then die.
- The creating process, called a *parent* process, creates a new process, which is a *child* of the creating process.
- A child process may create other processes, forming a tree of processes
  - Try `pstree`.
- How do those children live ?
  - With their parent or separately ?
  - Cooperative or in parallel ?

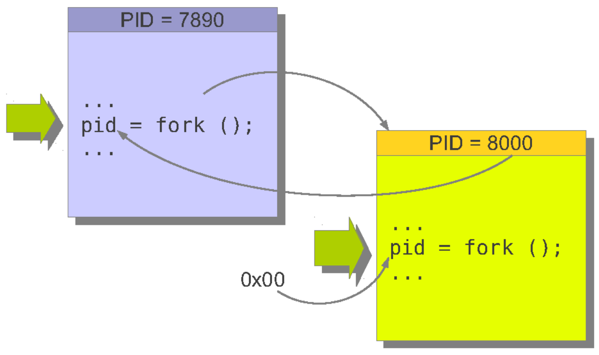
(cont'd.)

- Resource sharing ?
  - Parent and children share all resources.
  - Children share subsets of parent's resources.
  - Parent and children share no resources.
- Concurrent execution ?
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- Executable and address space ?
  - The child is a duplication of the parent.
    - e.g., `fork`
  - Child has a program loaded into it.
    - e.g., `exec`

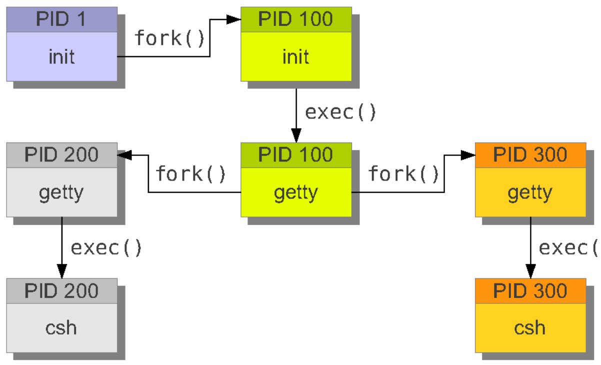
(cont'd.)

```
int main () {
   int pid;
   pid = fork ();
   if (pid < 0) {
      exit (-1);
   } else if (pid == 0) {
      execlp (“/bin/ls”, “ls”, NULL);
   } else {
      wait (NULL);
      printf (“child exit\n”);
      exit (0);
   }
}
```

(cont'd.)



(cont'd.)



Terminating a Process

- Process executes the last statement and asks the operating system to delete it (`exit`).
  - Child may return data to its parent via `wait`.
  - Resources are deallocated by the operating system.
- Parent may calls `abort` to terminate children.
  - Parent detects that a child has exceeded allocated resources.
  - Task assigned to the child is no longer required.
  - Parent is exiting
    - Operating system may not allow any child to continue.
      - All children are terminated – *cascading termination*.

Cooperating Processes

- The concurrent processes may either be
  - *Independent process* – cannot affect or be affected by execution of other processes.
  - *Cooperating process* – can affect or be affected by execution of other processes
- Advantages of cooperating processes
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

(cont'd.)

- There are problems we need to solved when multiple processes cooperatively are running on a single system:
  - Producer-Consumer Problems
  - Interprocess Communications
  - Deadlocks
  - etc.