Overview

- We have looked at what an operating system does, how it gets started and the interfaces to it (both the system service and hardware interfaces)
- Now we take a first look inside an OS at the algorithms and data structures that go into its implementation
- We start off with a fundamental OS concept: the process
- This will lead us on to cover multitasking and the process lifecycle

What is a Process?

- There is nothing dynamic about a program, it is simply a collection of instructions and data sitting on the disk
- When a program is launched the OS sets up an environment in which that program can be run i.e. it provides the address space in which the program will execute
- Once an address space is set up program execution can begin
- A process is the combination of the address space and the activity (or thread) of execution

What is a Process?

- We can think of a process as “a program in action” or, slightly more formally, a process can be thought of as a “sequence of states”
- If a process consists of a sequence of states from the initial to the final one, what makes up a state?
- State variables include values in CPU registers, the values a process sees in each byte of memory, the priority of the process, remaining timeslice etc.
- As instructions are executed, this state changes (at the very least the instruction pointer changes)

What is a Process?

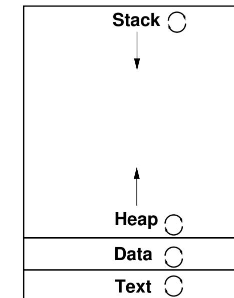
- Thus a process is a dynamic entity: if a program is a set of assembly instructions for a table, then a process is the act of following the instructions to assemble the table
- An OS is typically dealing with hundreds of processes at any instant and thus it needs a data structure to track the state of each one
- This data structure is used by the OS to represent, run and manage the runtime needs (in terms of memory, open files etc.) of a process
- This data structure is called a process control block or PCB and the OS keeps all of them together on a linked list in kernel memory

What is a Process?

- Compartmentalising each process in its own address space (region of memory) gives significant benefits:
  1. One process cannot see the memory belonging to another process and so confidentiality is assured
  2. If a process goes haywire it can only crash itself since it is locked into its personal address space
  3. Living alone in its own address space, to the process it looks like it has complete control of the (virtual) machine
- What does a process look like in memory?
- Or, what would a process see if it looked in the mirror?

Process Layout in Memory

- You can think of a process as existing in a box in memory with the following structure:
  - Text is instructions
  - Data is data
  - Stack supports procedure call and return and local variable allocation
  - The heap supports dynamic memory requirements
- How the box is implemented and enforced is dealt with under memory management



Process Creation

- Once a process is created we add the corresponding process control block to the list and the operating system manages it thereafter but where do processes come from?
- The system service which creates a new process is called `fork` (can you think why?)
- A call to `fork` duplicates the current process so immediately after the call there are two identical processes, each executing the same program at the same point
- Running two instances of the same program is usually not very useful so after the `fork` we need to do something else. . .

Process Creation

- One process, usually the new one, the child, overwrites its address space with a new program
- It does this by invoking another system call, `exec`
- So `fork` creates an environment for a new process by duplicating the current one, `exec` initialises it with a new program and starts its execution
- In Linux, `fork` calls `clone` which allows fine grained control over new process creation

Multitasking

- A processor runs a process, executing the instructions of the associated program
- Assume we are dealing with a uniprocessor and thus only one process can be executing at any instant
- The processor must be concurrently shared between processes if they are all to make progress: this is multitasking
- A process is given the CPU and before removal its state (register contents etc.) is saved in its PCB and another process is selected to run
- When its turn comes round again, its state is restored from the PCB and the process executes again

Multitasking

When should a process relinquish the CPU?

- If a process has held the CPU for too long; the decision to let go may come from the process itself or be forced on it by the OS through assigning it a CPU timeslice
- If a process must wait on an event (e.g. data to be read from the disk) it makes sense for it to voluntarily give up the CPU until that event completes
- We might also wish that a process give up the CPU when some urgent work needs doing i.e. a higher priority process wants the CPU
- When the decision to relinquish the CPU always comes from the process we have cooperative multitasking

Multitasking

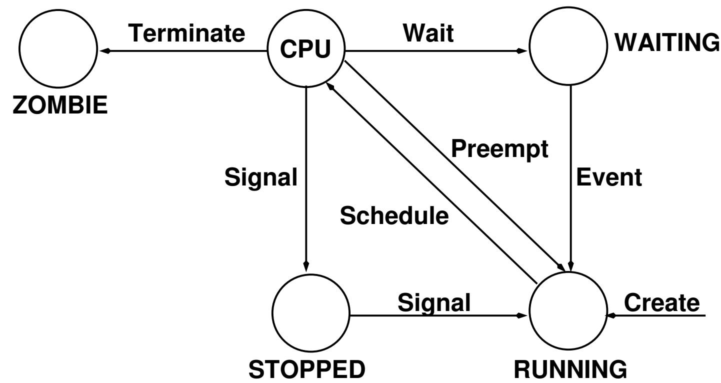
- When the decision may also come from the operating system we have preemptive multitasking
- Consider two processes, A and B, A periodically reads data from from drive X while B reads data from drive Y
- When A is blocked while it waits for the requested data to arrive B may be in a position to do some useful work
- So we mark A as non-runnable, remove it from the CPU and make a note to only mark it runnable again when the data it requested has been delivered to its address space
- While A sleeps we give the CPU to B
- The combined completion time for A and B is shortened

The Process Lifecycle

A process will be in one of 4 states

1. RUNNING: On the run queue, ready to do some work, there will be many like this (RUNNABLE might be a better description)
2. WAITING: On an event queue waiting for some event to occur before it can proceed
3. ZOMBIE: Terminated but PCB still around in memory
4. STOPPED: Paused for debugging purposes

The Process Lifecycle



The Process Lifecycle

A process may lose the CPU for one of 4 reasons:

- Voluntarily: a requested resource is unavailable or a process must wait on some event; it gives up the CPU so a runnable process can do some work
- Death: a process may finish, becoming a zombie
- Preemption: the operating system may step in to forcefully take the processor away; occurs when a process has used up its timeslice or a higher priority process wants the CPU
- Signal: the process may receive a signal causing it to halt and move into a stopped state

Context Switching

- Context switching is the procedure of reallocating the processor from one process to another
- It is essential the current state of the process/virtual machine be saved: registers, memory allocations, timeslice, priority etc.
- When a process becomes active again its state is restored and to the process everything is exactly as it was before it was context switched
- If a process was to watch the time on a wall clock it would occasionally notice time jump forward but apart from that it would look like it alone controlled the machine

So Where Are We?

- We have processes with each living in its own address space and we understand the associated benefits
- We have multitasking and can run multiple processes concurrently
- But things are not perfect:
  1. Context switches are expensive and waste CPU cycles
  2. Processes cannot easily talk to each other or share information
- What can we do?