**OPERATING SYSTEMS LEC 02**

**Processes/Jobs**

In a batch system we call the program **Jobs** but in a time-shared system they are called **user programs** or **tasks**.

A **Process** is a program in execution. A program (executable file containing list of instructions) is passive. A **process** has a Program Counter (PC) and specifies the next instruction to execute, which it does sequentially. It also requires resources to complete its task.

A process includes:

* Text section: program code
* A close up of text on a white background

  Description automatically generatedProgram counter: pointer to the next instruction to execute.
* Contents of processor registers.
* Stack: contains temporary data
* Data section: contains global variables
* Heap: memory that is dynamically allocated at run time.

A process can be **user** level or **kernel** level and can be in the following states:

**New** – process is being created.

**Running** – instructions are being executed.

**Waiting** – waiting for some event to occur

**Ready** – waiting to be assigned to a processor

**Terminated** – finished execution.

A process is represented by a data structure called a **Process Control Block** (PCB) which contains information about the process such as the process number (pid) and counter.

A **parent** process can create other processes (**children**).

**CPU Scheduler**

Its job is to select one of the available processes to be executed. The aim is to maximise the utilisation of the CPU by sharing the time between processes.

The OS keeps several queues.

**Job Queue:** All processes entering the system are put in a job queue.

**Ready Queue:** Set of processed in main memory, ready and waiting for execution.

L**ong term scheduler** selects which processes be moved into ready queue.

**Device Queue:** Processes waiting for an I/O device.

**Short term scheduler** dispatches processes to be executed next.

Can have I/O bound processes or CPU-bound processes. I/O has many short processes while CPU has few long processes.

**Context Switch**

When the CPU switches to a new process, the system must save the context of the old one and load the new processes context. This time is an overhead as no useful work is being done by the system and is a bottleneck usually dependent on hardware support.

**Process Creation**

A tree of processes can be formed when a parent process creates child process. This tree is formed with fork (LINUX), with the root processes having a PID of 1.

**For Resources:** A parent and child can share all resources, the child may share a subset of parent’s resources or they may not share resources at all.

**For Execution:** Parent and child may execute concurrently (LINUX) or parent may wait for child to terminate.

**For Address Space:** Child process address space is duplicate of parent’s or child has a program loaded in it.

**Process Termination**

A process terminates when the last statement is executed and/or calls exit. This will cause the child’s resources to be deallocated by the OS unless it doesn’t call wait and the child does – in which case the child will become an orphan and will have root as its parent.

Parents can also abort child processes in cases where the child has exceeded resources, or it is no longer required. Some OS does not allow child to continue if parent terminates (Not LINUX).

**Cooperating Processes**

These processes can affect or be affected by the execution of another process.

Compare to an **Independent process** which cannot be affected or affect other process executions.

Cooperating can communicate via shared-memory or message passing.

Advantages for cooperating process is:

* Information sharing – users may need same information
* Computation speed-up – break task into subtasks and run in parallel
* Modularity – Divide system function into separate processes
* Convenience – User may want to edit, print in parallel

**Producer/Consumer Problem**

There are processes that produces information and another that will consume the data. In order for this to run concurrently a buffer needs to be in the middle. Can be **unbounded** or **bounded** and can be in the form of shared memory or message passing.

**Shared Memory**

A process uses a system call to create a shared memory region. Shared process must attach to the memory region’s address space. Then access to this region is the same as normal, thus not requiring a system call (but need to be synchronised).

**Message Passing**

Uses a system call to exchange message between processes. More time consuming, OS handles synchronisation. This is good for distributed system with small amount of data exchanged and good for multicore systems.

Has two operations: Send and receive. In order for two processes to communicate they need to establish a connection and then exchange with send/receive.

**Direct Communication (Message Passing)**

* Each process must explicitly name the recipient/sender of communication.
* Links are formed automatically
* Associated with only one pair of processes with only one link between them.

**Indirect Communication (Message Passing)**

* Message are sent and received form buffers (mailboxes).
* Processes can only communicate if they share a mailbox
* Link can be associated with many processes
* A pair of processes may share several links

To create:

1. Create new buffer
2. Send and receive messages though buffer
3. Destroy buffer

**Synchronisation (Message Passing)**

Message passing can be blocking (synchronous) or nonblocking (asynchronous).

* Blocking send – sender is blocked until message is received
* Nonblocking send – sender resumes operation after sending the message
* Blocking receive – receiver blocks until message is available.
* Nonblocking receive – retriever receives either valid message or NULL.

**Buffering (Message Passing)**

* A link between processes can have a buffer of messages.
* Can be bounded to have a max number of messages and if full the sender must wait for the receiver to read.
* Can be unbounded with infinite number of messages, in which the sender will never have to wait.
* For a non-zero capacity buffer the sender does not know if message is received.
  + Can use asynchronous communication so receiver tells sender its read it.
* There are 3 main errors:
  + A process is terminated
    - Fixed by OS telling other process a process is terminated
  + Message is lost due to hardware error
    - Uses detections or acknowledgement of receive is required to fix
  + Scrambled message
    - Just resend the message to solve.

**Threads**

* A thread of control is an independent sequence of execution of program code in a process and is a basic unit of CPU utilisation.
* A traditional process has a single thread that is in sole possession of a process’s memory and other resources.
  + Context switches become bottleneck
  + Threads share all process memory and other resources so used to avoid this bottleneck.
* Threads in a process are:
  + Usually invisible from outside the process
  + Scheduled/executed independently in the same way as different single-threaded processes.
* On a multiprocessor, different threads are executed on different processors.
* Threads are similar as processes:
  + Can be in different states
  + Share CPU – only one thread is running at a time
  + Thread executes sequentially, has its own PC and stack.
  + Thread can create child threats and can block waiting for system calls
* Major difference of processes is that they are not independent of each other
  + All threads can access (and read/write stacks) of every address in task
  + No protection between threads (in a process)
    - Not needed as threads (in same process) are written by same programmer whilst processes are not.

**Benefits of Threads**

* Responsiveness: When one thread in a program is blocked and waiting a second thread in the same task can run. (In an interactive application)
* Resource Sharing: By default, threads share memory/resources in a process
  + Applications using common buffer benefit from thread utilisation
* Economy: More economical to create and context switch threads than processes
* Utilisation of multiprocessors: Threads can run in parallel on different processors

**User and Kernel Threads**

* **Kernel-Supported** threads are a set of system calls similar to processes.
  + Kernel doesn’t context between threads - > time consuming
* **User-level** threads are support via a set of library calls at user level.
  + Kernel doesn’t know that there are user-level threads
  + Thread management is done by user-level threads library
  + Faster than kernel-support threads
  + However, a set of user-level threads are a single thread which means if one is blocked then the other cannot run.

**Many-to-one Model**

* Many user-level threads are mapped to one kernel thread
* Threads within a process cannot run in parallel on multiprocessors

**One-To-One Model**

* Each user thread is mapped into a kernel thread
* Allows more concurrency; allowing other threads to run when one is blocked
* Allows multiple threads running in parallel on multiprocessors
* (Windows OS, LINUX)

**Many-To-Many Model**

* Allows *M* user-level threads to be mapped to *N* kernel threads
* Allows OS to create a sufficient number of kernel threads
* Many user threads can be created as necessary and corresponding kernel threads can run in parallel on multiprocessors

**Thread Libraries**

* Programmers user API of thread library to create and manage threads
* POSIX Pthreads: either kernel or user threads
* Win32: kernel threads
* Java: implemented using thread library of host system
  + Windows uses Win32, Linux uses Pthreads

**Pthreads**

* Primary goal is to allow multithreaded programs to be portable across multiple OS platforms
* API specifies behaviour of thread library, implementation is up to development of the library
  + Can be kernel or user level threads
* Uses POSIX:
  + Each thread maintains processor registers, stack and signal mask
  + Other resources must be globally accessible to all threads (in process)

**Threading issues**

* If a thread invokes exec, the program specified will replace entire process including all threads.
* If a thread uses fork it will duplicate only the thread that invoked the call (if exec is called immediately) OR may duplicate all threads (if the new process does not call exec).

**Thread Cancellation**

* Two scenarios if a thread is terminated before completion:
  + Asynchronous cancellation – one thread immediately terminates target thread.
  + Deferred cancellation – target thread periodically checks if it should terminate
* If resources have been allocated to a cancelled thread the OS may not be able to reclaim all of the resources (asynchronous cancellation especially).
* A thread cancelled in middle of updating data shared with other threads

**Signal Handling**

* Signals in Linux is used to notify processes that a particular event has occurred
* Synchronous signals: delivered to same process performing operation causing the signal.
* Asynchronous signal: generated by an event external to running process.
* When a process has more than one thread where should the signal be delivered?
  + To the thread to which it applies (for synchronous)
  + Every thread in process (for some asynchronous)
  + Certain threads in the process
  + Specified thread to receive all signals for process
  + POSIX: signals are sent to process and each thread has a signal ask to allow the thread to disable signals of a particular type.

**Thread Pools**

* In a multithreaded web server when a server receives a request a separate thread is created. This can be bad because:
  + Time consuming for creating/destroying a thread for each request
  + No limit on number of threads created
  + Solution is to make a thread pool
* A thread pool creates a number of threads at process start-up and places them into a pool to wait for work.
* Benefits:
  + Faster to service a request
  + Limits number of threads that exist at any one point
* Number of threads in pool can be set:
  + Heuristically: based on system resources (CPU, memory, number of requests).
  + Dynamically: adjusted based on usage patterns

**Thread specific Data**

* By default, threads within a process share data of the process
* For some applications, each thread might need its own copy of certain data.
  + Need thread-specific data.
  + Supported by Win32, Pthreads and java.

**Windows Threads**

* Implement Windows API
* Uses one-to-one mapping
* A process contains one or more threads
* Each thread contains:
  + Thread id
  + Register set
  + Separate user and kernel stacks
  + Private data storage area

**Linux Threads**

* No difference between processes and threads
  + Known as tasks rather than threads/processes
* Linux provides fork and clone calls
  + Fork use to create a traditional process
  + Clone uses to create a thread
* Uses one-to-one thread mapping.