Concurrent Systems

Concurrency

Concurrent vs sequential

Sequential	Concurrent
One stream of execution	>1 stream of execution
At any point in time execution is taking place from a single specific piece of code	At any point in time execution is taking place from multiple pieces of code
When a piece of code has been executed, the next piece to execute is chosen (next in sequential order, or jumping to some other line of code)	When a piece is executed, then next piece to run may be chosen from any of the multiple "streams"

Concurrent vs parallel

Concurrent	Parallel
Design approach to make use of available computing resources	**Optimisation** approach to use multiple hardware units to speed up computation
Can work with 1+ hardware units	Multiple hardware units!
Parallelism is a specialised form of concurrency	

Types of Concurrency

Extent to which “streams” interact	
Minimal/Managed Interaction	**Maximal Interaction**
No direct access to memory, communication carefully managed	Easy access to shared memory
Expensive to implement	Cheap
“Process”	“Thread”
Most applications run like this	Many complex applications use threads

	internally

Types of Parallelism

(Flynn's taxonomy)

1. Single instruction single data (SISD)
    - Single core processor
2. Single instruction multiple data (SIMD)
    - Vector/array processor
3. Multiple instruction single data (MISD)
    - Not sure, pipelined?
4. Multiple instruction multiple data (MIMD)
    - Shared memory (tightly coupled)
        i. Asymmetric (the questionable one lol)
        ii. Symmetric (SMP) - multi-core processor
    - Distributed memory (loosely coupled) - cluster

Massively Parallel

If there is almost no interaction between parallel threads calculating independent parts of solution.

Bad write-set interaction (no mutex)

A	Thread A	G	Thread B	B	Time
-	-	0	-	-	|
0	A=G	0	-	-	|
1	A++	0	B=G	0	|
1	G=A	1	B++	1	|
1	-	1	G=B	1	v

Concurrency (mutex) Example

```c
#include <stdio.h>
#define NUM_SLICES 1000
#define H 4.0/NUM_SLICES

double answer;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

double f(double x) {
      return (16.0 - x*x);
}

double trapezoid(double a, double b) {
      return H*(f(a)+f(b))/2.0;
}

void *IntegratePart(void *i) {
      double a,b,area;
      int rc;

      a = (int)i * H;
      b = a + H;
      area = trapezoid(a,b);

      rc = pthread_mutex_lock(&mutex);
      answer = answer + area;
      rc = pthread_mutex_unlock(&mutex);
      pthread_exit(NULL);
}


int main(int argc, const char * argv[]) {
      pthread_t threads[NUM_SLICES];
      long rc, t;

      answer = 0.0;

      for (t = 0; t < NUM_SLICES; t++) {
            pthread_create(&threads[t], NULL, IntegratePart,
               (void *)t);
            if (rc) {
                 printf("ERROR return from pthread_create(): %ld\n",
                   rc);
                 exit(-1);
            }
       }

       for (t=0; t<NUM_SLICES; t++) {
            pthread_join( threads[t], NULL);
       }
       printf("%f\n", answer);
       return 0;
}
```

Atomic Instructions

Atomically - no exception will take place until the instruction has completed.

Test-and-Set (TAS)

```
local = global; global = 1
```

Exchange

```
local = global1; global1 = global2; global2 = local;
```

Compare-and-Swap (CAS)

```
if local == global1
     then global1 = global2; return true
     else local = global1; return false
```

How fast can we go?

Correctness & Fairness

Correctness

In sequential problems

Partial correctness: The answer is correct if the program halts

Total correctness: The program does not halt and the answer is correct.

In concurrent programs

Safety property: A property must *always* be true.

Liveness property: A property must be *eventually* true.

Fairness property: All threads ready to run will do so eventually.

Starvation-Freedom: Any thread waiting for a resource will eventually be granted access.

Checking correctness/Verification

General idea

- States/state transitions
- Construct a state transition diagram (model) of every possible state of concurrent program
- Reason with model (model checking) to prove certain property

How to verify

- Manual construction and inspection of state diagrams - only for most trivial programs

- Automated construction and inspection of state diagrams
    - Constructing and using state diagram can be automated using model checker
- Formal specification and verification using temporal logic

Model-checking

Automated checking of whether a logical property is upheld by a finite-state model of a system, given an initial state.

Automatically checks whether M ⊧ φ holds, where M is a finite state model of a system, and property φ is stated in some formal notation.

Weak Fairness

If a statement is ready for execution and it is *guaranteed* that it will *eventually* be executed, then the system is **weakly fair**. We often assume weak fairness.

Closed world assumption

- A model checker proves by **negation** or **exhaustion**
- These proofs are only valid if the model checker knows *everything*
- Property deemed true or false by model checker may not truly be in real world

Promela & SPIN

SPIN

Is a model checker using Linear Temporal Logic.

Ways to run

Simulation

- Performs **one** possible run of the system
- Runs aka **scenarios**
- Choices are random, can control randomness from command line

- Can also do guided simulation from trail file

```
spin -u1000 dekker.pml
```

Verification

- Systematically searches over **all** possible runs of system
- Checking for truth of desired properties
- If check fails, it outputs a **counter-example**
  - A run of the system that leads to a property failure
  - To trail file

```
spin -a dekker.pml; cc -o pan pan.c; ./pan
```

Shrinking a counter-example

- A failure is noted and a .trail file is created
  - depth = number of steps taken to failure
  - To run the trail

  (given counter example found by `spin -run mutex_flaw.pml`)
  `spin -p -t mutex_flaw.pml`

- Two ways to find shorter counter-example
  - Keeping track of depth and iteratively search for minimal depth:
    `cc -DREACH -o pan pan.c ; ./pan -i -mDD` where DD is known depth.
  - Use breadth-first search instead:
    `cc -DBFS -o pan pan.c ; ./pan`

Promela Elements

Process

- Defined by **proctype** definition (which has…)
  - Name

    - List of formal parameters
    - Local variable declarations
    - Body of process
- Executes concurrently with other processes
- Communicates with other processes by:
    - Global variables
    - Channels
- Has local state
    - Process counter (where in the process it is)
    - Contents of local variables
- Creation and execution
    - Created using **run** statement (returns process id)
    - Processes can be created at any point in execution
    - Can also be created by adding **active** in front of **proctype** declaration
    - Start execution immediately after their creation
    - There is an upper bound to number of processes

Types and Type Declarations

- Basic types
    - bit [0..1]
    - bool [0..1] or true,false
    - byte [0..255]
    - short [-2^15..2^15]
- Arrays - with zero-based indexing and only 1D
- Records (like structs) using typedef keyword:

```
typedef Record {
     short f1;
     byte f2;
}
```

- Mtypes (enumeration type) - set of names treated as distinct values

```
Mtype = {name1, name2, …,nameN }
```

Statements

Basic statements (are atomic)

- Statement is either
    - **Executable**: statement can be executed immediately
    - **Blocked**: statement cannot be executed
- Assignment
    - var = 3
    - ALWAYS executable
- Expression
    - Executable if it evaluates to non-zero
    - 2 < 3 is always executable
    - X < 27 is only executable if x is smaller than 27
    - 3 + x is only executable if x is not equal to -3
- **skip** - always executable, does nothing (updates process counter)
- **run** - executable if new process can be created
- **printf** - always executable (ignored)
- **assert(<expr>)** - always executable, if <expr>=0, returns with error

Non-atomic statements

- Sequencing statements
    - List statements separated by semicolons
    - stat1; stat2; stat3; …; statN
    - Do not execute atomically
- If statement

```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0) -> n=n-2
:: (n % 3 == 0) -> n=3
:: else -> skip
fi
```

-
    - Each :: introduces alternative started with a guard statement
        - If any guards are executable, one is non-deterministically chosen
        - Once the alternative has been run, the conditional is over
        - Else only selected if none of the guard statements are true

  - If no guard is executable, the statement blocks until at least one is executable
  - -> is the same as ;
- Do statement (loop)

```
do
 :: choice1 -> stat1.1; stat1.2; stat1.3; …
 :: choice2 -> stat2.1; stat2.2; stat2.3; …
 :: …
 :: choicej -> break;
 :: choicen -> statn.1; statn.2; statn.3; …
od;
```

  - Similar to if but loops around to choice at end until it breaks
  - Use break to escape
  - Strange notation from Guarded Command Language (Dijkstra) to reason about non-determinism

Atomic statements

```
atomic { stat1; stat2; ... statn }
```

- Used to group statements into atomic sequence
- All statements are executed in ONE sequence (no interleaving)
- Executable if stat1 is executable
- If a stat_i (with i > 1) is blocked, atomicity is temporarily lost!

#define and inline

- Can use #define, #if, #ifdef, …
- Also use **inline**

```
inline name(arg1, …, argN){
      stat1; stat2; ... statn
}
```

- Can only be called like a statement
- Can contain calls to other inlines, but NOT recursive
- Behaviour
  - It is **textual substitution**
  - Does not represent function/procedure
  - If a statement declares a variable, it has global scope

- inline better than #define for errors

Promela Verification Constructs

SPIN does standard verification checks by default but Promela can also do others without Linear Temporal Logic.

Assertions

Already discussed

Promela Labels

Labels are allowed to be attached to statements.

Labels with specific prefixes trigger specific correctness checks.

End-State label

- For when there is a valid end-state not at the end of a process' code (so dead-lock error not caught)
- Label with anything starting with **end**
- Eg: when there's a program that should not terminate

Progress-State label

- For when loops in program
    - If they're desirable, progress is made each time (progress cycles)
    - Otherwise, undesirable non-progress cycles
- A given loop may never get executed, and if it is required this is starvation
- Label these states starting with **progress** (to indicate they should occur in an infinite loop
- Then use special verifier optimised for non-progress (NP) cycles

```
cc -DNP -o pan pan.c; ./pan -l
```

Dekker's Algorithm

```
bool turn;
bool flag[2];
byte count;

active [2] proctype mutex()
{
  pid i, j;
  i = _pid;
  j = 1 = _pid;
  again:
    //Acquire lock
    flag[i] = true;
    do
    :: flag[j] ->
        if
        :: turn == j ->
            flag[i] = false;
            !(turn == j);
            flag[i] = true
        :: else
        fi
    :: else ->
        break
    od;

    count++;
    assert(count==1);
    count--;

    //Release lock
    turn = j;
    flag[i] = false;

    goto again
}
```

pthreads

pthreads vs processes

Processes	pthreads
Separate memory spaces from each other	Share memory space (in same process)
Expensive inter-process communication & scheduling	Cheap inter-thread communication & scheduling.
pthreads live within **processes**	

pthreads

pthread_create()

```
int pthread_create( pthread_t *, const pthread_attr_t *, void
*(*)(void *), void *);
```

```
rc = pthread_create(&thread_data, NULL, ThreadCode, threadarg);
```

Precondition: N/A

Postcondition: `thread_data` contains pthread ID, `ThreadCode` is ready to execute

pthread_join()

```
int pthread_join(pthread_t, void **);
```

```
rc = pthread_join(thread_data, NULL);
```

Precondition: `thread_data` refers to a created thread, there is no other live call to `pthread_join()` with the same `thread_data` value.

Postcondition: thread identified by `thread_data` has terminated.

`pthread_join()` does not return until thread has terminated

pthread_mutex_init()

```
int pthread_mutex_init(pthread_mutex_t *, const
pthread_mutexattr_t *);
```

```
Rc = pthread_mutex_init(&mymutex, NULL);
```

Precondition: none

Postcondition: `mymutex` is initialised and unlocked.

pthread_mutex_lock()

```
int pthread_mutex_lock(pthread_mutex_t *);
```

```
rc = pthread_mutex_lock(&mymutex);
```

Precondition: `mymutex` is initialised.

Postcondition: `mymutex` is locked and the calling thread owns that lock.

pthread_mutex_unlock()

```
int pthread_mutex_unlock(pthread_mutex_t *);
```

```
rc = pthread_mutex_unlock(&mymutex);
```

Precondition: `mymutex` is locked, and the calling thread owns the lock.

Postcondition: `mymutex` is unlocked and no thread owns that lock.

pthread_cond_init()

```
int pthread_cond_init(pthread_cond_t *, const pthread_condattr_t
*);
```

```
Rc = pthread_cond_init(&myconvar, NULL);
```

Precondition: none

Postcondition: `myconvar` is initialised and has not been signalled.

pthread_cond_wait()

```
int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *);
```

```
rc = pthread_cond_init(&myconvar, &mymutex);
```

Precondition: `myconvar` is initialised and `mymutex` is locked, and the calling thread owns that lock.

Postcondition: `mymutex` is locked and the calling thread owns the lock. Time may have passed.

pthread_cond_signal()

```
int pthread_cond_signal(pthread_cond_t *);
```

```
rc = pthread_cond_signal(&myconvar);
```

Precondition: a previous call to `pthread_cond_signal` with associated `myconvar` and its `mymutex`. `mymutex` is locked and owned by the calling thread.

Postcondition: A signal has been sent to `myconvar` to wake the thread waiting on it. `mymutex` is locked and owned by the calling thread.

pthread_exit()

```
void pthread_exit(void *);
```

```
pthread_exit(NULL);
```

Precondition: formally none, but should not own any mutex locks.

Postcondition: terminates the calling thread. Any mutex locks it holds are still held. Values of local variables are undefined.

Called implicitly on the normal end of `ThreadCode`.

Operating System

Complementary Views

Extended machine

- Essentially a virtual machine for programs to run on
- Easier to program than real machine
  - Virtual memory gives uniform supply of memory regardless of individual machine
- Provides useful services bare machine doesn't have
  - File system, media handling, communications, …

Resource Manager

- OS is responsible for fair and efficiency management of resources of computer system
  - CPU (scheduling, …)
  - Memory (Memory management, virtual memory, paging, …)
  - Devices (Disks, special-purpose devices)
  - Files & directories
  - Network connections (ports)

OS Responsibilities

Resource allocation

- OS is responsible for sharing resources among the processes that exist at any given time on computer
- **Processor time** is allocated to a process (or its threads) by a **scheduler**
- **Physical memory** is allocated to processes by a **memory manager**
  - File storage
  - Many programs need to share file storage in structured way

  - File I/O managed by OS to prevent corruption
- Many devices (e.g. I/O devices) are allocated on an obtain/release basis using queues
  - Running program needs to perform I/O operations with 1+ devices
  - For efficiency, safety, fairness, I/O ops performed by OS on behalf of requesting process

Other things

Communication

When one process needs to communicate with another, either locally or over network

Error Detection

OS should handle unexpected error events in appropriate manner.
Including loss of power, prohibited memory access, stack overflow, I/O error, etc…

Accounting

Recording important system events + gather statistics on computer system.

Protection

Concurrent processes must be prevented from interfering with each other.

Important/sensitive information must be protected and hidden from unauthorised access.

Processes

Processes: AKA **tasks** - entities that run as self-contained programs in computer under control of OS. Given ownership of resources like memory/devices by OS.

Multitasking

OS can grant processor to a ready process easily, more difficult to get processor back.

Co-operative Multitasking

- Process relinquishes the processor
- Advantages

    - Lower overhead - processes can be prepared to yield the processor
    - No danger that processor will be given up at a critical point
- Disadvantages
    - Hard to program yield calls into a program
    - If yield calls are done badly, program hogs the processor
- Generally inadvisable

Preemptive multitasking

- Processor removed from process by OS (usually by timer interrupt exception handler)
- Advantages
    - Robust against badly written/malicious programs
    - Easier to write applications for
- Disadvantages
    - Can be expensive as full process contexts must be stored and restored
    - May cause problems if process preempted at a bad time

Hardware Support of OSes

Basic

- CPU
- Memory
- I/O

Standard (adds support for concurrency)

- Interrupts
- Privilege modes
- Atomic instructions
- Memory management

Understanding a program

Basic Program execution

Has key components

Memory:

- Program machine-code lives here with static global variables
- Space reserved for stack
- Dynamic memory allocated from the heap

CPU

- Program counter (PC) register with address of next instruction
- Stack pointer (SP) register pointing to current top of stack
- (maybe heap pointer (HP) pointing to boundary of heap)

I/O

- Accessed either via special IO instructions or special memory locations

Program Context

Defines all information needed for a program to run

- Values in all CPU registers and global static variables
- Contents of stack and heap

Program Lifetime

1. Initialisation/setup phase
2. Running phase, alternates between
    a. CPU doing useful computations
    b. CPU waiting while *slow IO operations compute*
3. Teardown phase at the end (if program should terminate)

Slow IO is awful and motivates interrupts.

Interrupts

A hardware feature which allows external signals to break the flow of normal program execution.

When an interrupt occurs and is accepted:

1. CPU saves critical registers on the stack (PC, plus maybe others)

2. Loads PC with address of interrupt handler
3. When handler has run, executes a return from interrupt (RTI) instruction which restores the PC and other saved registers
4. **Note interrupt does not save whole program context.**

Interrupt sources

1. IO devices
   - IO completion interrupt signal
2. Timers
   - Timer is special purpose IO device
   - Produces timer interrupt signal
3. Interrupting interrupts
   - **Exceptions**: interrupts on serious error
   - **Traps**:special CPU instruction which acts as interrupt

Exception number

Many kinds of exception/interrupt/trap, so each kind has a number which identifies its handler code (in the exception table)

Using interrupts to switch program

Access IO code using trap instructions which

Gets IO device going AND

- Arrange for relevant IO interrupt to occur
- Save all PC registers of current program (which becomes **suspended**)
- Loads all registers (incl. PC) of the previously suspended program
- Now the previously suspended program is running

Essentially, the interrupt handler does its job then calls the scheduler to decide what runs next.

Context switching

Context switching is the process of

- Saving critical context of running code
- Restoring critical context of some suspended program

Privilege Modes

Idea

- Certain instructions will only execute if the CPU is at a high enough privilege level
    - Instructions that configure key hardware components or give direct access to IO devices
    - Executing without the right privilege leads to Privilege violation exception
- Increase privilege by causing exception
- Assume User (low privilege) and Kernel (high privilege)

Contexts with privilege

- CPU with privilege modes will have a status register with bits encoding those levels
- This register is *always* critical part of context and will always be saved/restored from stack on exception
- Memory management can also be used to protect large code chunks

Unix

Architecture Elements (from most basic down)

1. Hardware
    - Processors
    - Memory
    - Peripheral devices
2. Kernel - most basic facilities provided by OS
    - Memory management
    - Message passing
    - Device handling
    - Might be in privileged mode

3. System calls
    - Access to OS facilities via this
4. Shell, utilities, and libraries
    - Provide facilities for users and other programs
    - Not privileged in general
5. Applications

Scheduling

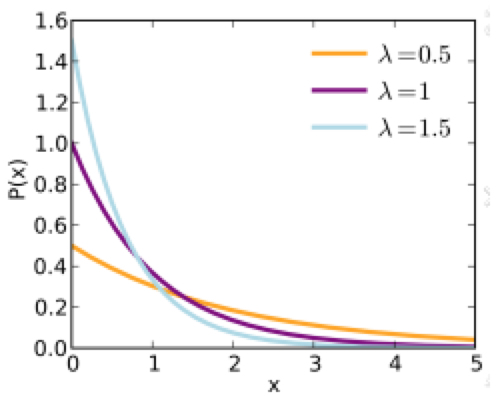
As only one processor can execute at a time on a single processor system. Executed until it is required to wait for some event (I/O, timer, resource). When it is stalled, we want something else to use the CPU.

Scheduler: component of OS that gives time to processes

Process Behaviour

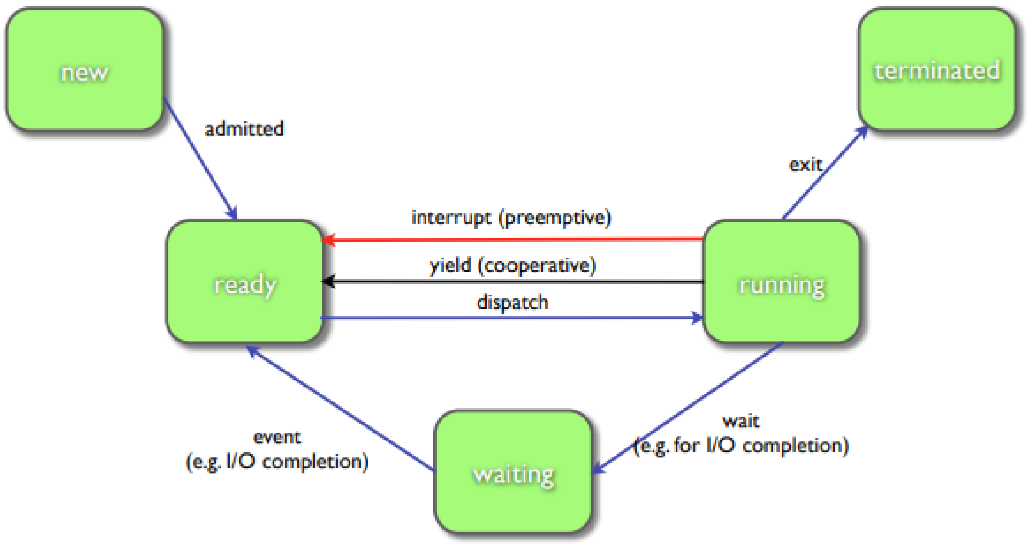
CPU/IO burst cycle

- Processes often alternate between bursts of CPU and IO activity
- CPU burst times have exponential frequency curve
    - Many short burst, few very long bursts
    - Occur independently at a constant average rate
    - Memoryless



Process state transitions

- Only one process can be running on a single processor at any tim
- Multiple processes may be in the waiting or ready state



CPU Scheduler (short term)

- When the CPU goes idle, OS must select another process to execute
- **Candidate processes**
    - Processes in memory which are ready to execute
    - Stored on ready pool
- Ready pool is not necessarily FIFO, depends on scheduler
- Longer term - number of processes eligible for execution must be managed

Scheduling goals

1. Fast response time
    - Look at waiting time (latency)
    - Look at turnaround time (time from arrival to end of burst)
2. High throughput
    - Avoid wasting time
    - Avoid too many context switches

3. Fairness
    - Avoid starvation!

Scheduling issues

1. Fairness
    - Every process gets processor time eventually
2. Efficiency
    - Make best use of resources available
3. Timeliness
    - Respond quickly to events
    - Provide processor time reliably, in **real-time**, over extended periods

These requirements may conflict

Real Time

Real time means correctness/utility of program depends on BOTH logical correctness and deadlines being met

- **Hard real-time:** If a deadline is missed, the system fails totally
- **Firm real-time:** If a deadline is missed, the delayed service is useless, but the *overall* system can tolerate the occasional failure
- **Soft real-time:** If a deadline is missed, the delayed service is *degraded* not useless, and the overall system is degraded but still useful/usable

Scheduling example

Processes which arrive in order, with burst times of 8, 4, 9, and 5 ms

Process	**Arrival Time**	**Next Burst Time**	**Priority (lower is better)**
P1	0	8	2
P2	1	4	4
P3	2	9	1
P4	3	5	3

Burst time: the amount of time the process will use in a continuous run without a break (eg for IO)

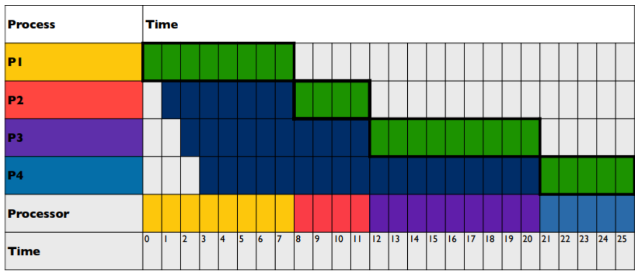
Waiting time/latency: time waited before the process runs, mean and variance are both important.

Turnaround time: waiting time + burst time

Scheduling approaches

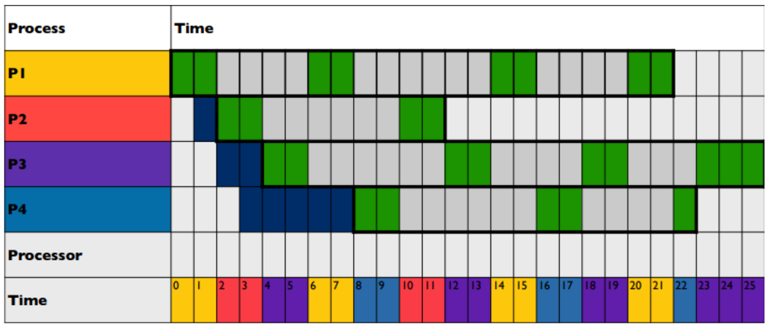
First-come First-served (FCFS)

1. First process to request CPU is first to be scheduled
2. Implemented with FIFO ready queue
3. When a process enters ready state, put at tail of ready queue
4. When CPU is free, process from head of the queue is dispatched and removed from queue
5. Average waiting time can be long and depends on CPU burst times



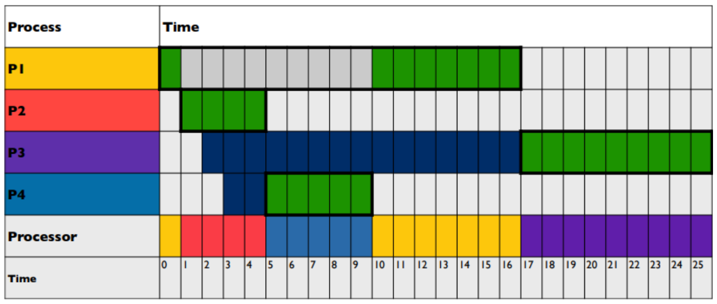
Round Robin

1. Process that requests CPU first is first to be scheduled
2. Implemented with FIFO ready queue
3. When process enters ready state, put at tail of of ready queue
4. When CPU is **interrupted**, a scheduling event occurs
    - Next process at head of queue is dispatched
    - Interrupted process put at end of queue
5. Shorter waiting time, longer turnaround time



Shortest Job First (pre-emptive) (SJF)

1. Process that requests CPU first is first to be scheduled
2. Implemented with FIFO ready queue
3. When process enters ready state
   - If shorter than current job, process put at head of queue, CPU is interrupted
   - If not shorter than current job, process is placed in ready queue **at location just ahead of any longer jobs already in the queue**
4. When CPU interrupted, a scheduling event occurs
   - Next process at head of queue is dispatched
   - Interrupted process is placed on the queue **at location just ahead of any longer jobs already in the queue**
5. Waiting time generally shorter, turnaround time generally longer



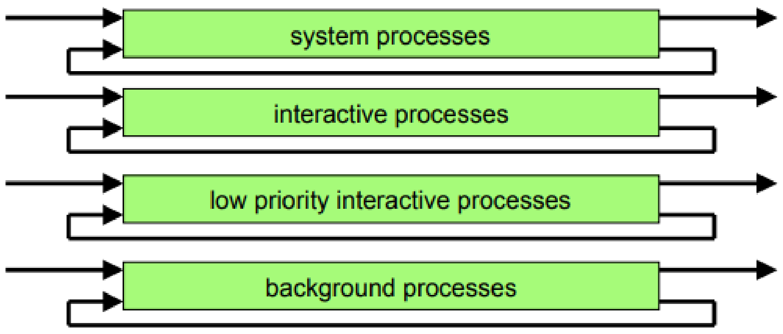
Example with all approaches

			First-come first-served		Round Robin		SJF	
Proces	**Arrival**	**Next Burst**	**Waiting**	**Turnaround**	**Waiting**	**Turnaround**	**Waiting**	**Turnaround**

s	Time	Time	Time	Time	Time	Time	Time	Time
P1	0	8	0	8	0, then 14	22	0, then 9	17
P2	1	4	7	11	1, then 6	11	0, then 0	4
P3	2	9	10	19	2, then 13	24	15, then 0	26
P4	3	5	18	23	5, then 15	20	2, then 0	7

Multi-level queue scheduling

Group processes together based on their characteristics or purpose:



Basic multi-level queue scheduling

- Processes are permanently assigned to each of these groups
- Each group has an associated queue with its own queuing discipline
  - FCFS, SJF, priority queuing, Round-Robin
- Additional scheduling between queues
  - Often implemented as *fixed-priority pre-emptive* scheduling

Multi-level feedback queue scheduling

- Allows processes to move between queues
- Allows IO bound processes to be separated from CPU bound processes for scheduling
  - Processes using a lot of CPU time on lower priority queues
  - Processes which are I/O bound remain in higher priority queues
- Processes with long waiting times are moved to higher priority queues

    - Prevents starvation
- Higher priority queues have absolute priority over lower priority queues
- We can change
    - Number of queues
    - Queueing discipline
    - Rules for moving process to higher/lower priority queues

Example - Linux

Scheduling goals

- Fast response time
- High throughput
- Avoid starvation
- Satisfy needs of high and low priority processes
- Etc...

Scheduler type:

Pre-emptive scheduling - when the **quantum** of the process expires, another process may be executed in its place

Process ranking

- According to priorities
- Priorities are dynamic
    - OS scheduler adjusts priorities based on behaviour
    - Decrease priority of long-running processes
    - Boost priority when on I/O completion
- Complex schemes for determining current priority of a process
- Simple to select next process

Interactive/Batch Process priority

- Every normal process has a static priority (100..139) (smaller is higher priority)
- Static priority gives base time quantum

```
if (static priority < 120) then
      base time quantum = (140 − static priority) x 20
else
      base time quantum = (140 − static priority) x 5
```

- Dynamic priority determines which process to execute
    - `dynamic_priority = max(100, min(139, static_priority − bonus + 5))`
    - Bonus in range (0..10), but if <5, it lowers the dynamic priority
    - Bonus based on average sleep time, generated from lookup table that gives bigger bonus to processes which sleep for longer

Scheduling classification

- Differentiates between interactive and batch
- Interactive if `(bonus-5 >= static_priority/4 - 28)`
    - Easier for high priority process to become interactive
    - Very low priority processes *cannot* become interactive
- Active or expired (avoids starvation of low-priority processes)
    - **Active:** have not exhausted their quantum, are allowed to run
    - **Expired:** have exhausted their quantum, are not allowed to run until there are no more active processes
    - (In practice more complex - interactive processes can be made active as long as nothing is starving)

Real-time processes

- Assigned priorities in range 1..99 (lower is higher priority), not dynamic
- Two classes of real-time process
- FIFO process
    - Continue executing until they
        - Block OR
        - Are pre-empted by a higher priority process
    - Time slice is irrelevant
- Round-Robin process

    - Continue until they
        - Block,
        - Are pre-empted by a higher priority process, OR
        - Time slice expires
    - When time-slice expires, placed at tail of priority list

Time-slicing

- Periodically, time-slice of currently executing process is decremented
- If time slice is exhausted
    - Process removed from active queue
    - Time slice restored
    - Then placed either
        - On active list - if it is interactive and there are no starving expired processes
        - On expired list - otherwise
- Schedule next process to execute from the head of the highest priority active queue
- **If all processes are expired, expired and active lists are swapped (by changing pointers)**

Example - xv6

Scheduler

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns. It loops, doing:
// - choose a process to run
// - swtch to start running that process
// - eventually that process transfers control
// via swtch back to the scheduler.
void
scheduler(void)
{
        struct proc *p;
        struct cpu *c = mycpu();
        c->proc = 0;

        for(;;){
                // Enable interrupts on this processor.
                sti();

                // Loop over process table looking for process to run.
                acquire(&ptable.lock);
                for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                        if(p->state != RUNNABLE)
                                continue;

                        // Switch to chosen process. It is the process's job
                        // to release ptable.lock and then reacquire it
                        // before jumping back to us.
                        c->proc = p;
                        switchuvm(p);
                        p->state = RUNNING;

                        swtch(&(c->scheduler), p->context);
                        switchkvm();

                        // Process is done running for now.
                        // It should have changed its p->state before coming back.
                        c->proc = 0;
                }
                release(&ptable.lock);

         }
}
```

Notes

- It works with multicore
- Key variables
    - Pointer c to selected process
    - Pointer p to walk the process table (ptable)
- Process table is locked
- Skip non-runnable processes, run the first runnable process

- Preparation for running process
    - Mark it as running
    - Set processor mode to user (switchuvm)
- Run by calling swtch
- When it returns to scheduler
    - Set processor mode to kernel (switchkvm)
    - Set selected process pointer to 0

Process Table

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

struct {
       struct spinlock lock;
       struct proc proc[NPROC];
} ptable;
```

```
// Per-process state
struct proc {
       uint sz; // Size of process memory (bytes)
       pde_t* pgdir; // Page table
       char *kstack; // Bottom of kernel stack for this process
       enum procstate state; // Process state
       int pid; // Process ID
       struct proc *parent; // Parent process
       struct trapframe *tf; // Trap frame for current syscall
       struct context *context; // swtch() here to run process
       void *chan; // If non-zero, sleeping on chan
       int killed; // If non-zero, have been killed
       struct file *ofile[NOFILE]; // Open files
       struct inode *cwd; // Current directory
```

```
        char name[16]; // Process name (debugging)
};
```

Notes

- Table has a spinlock to manage access
- Table has array of struct proc with metadata of each process
- Most relevant parts of proc
    - state
    - pid
    - context
    - chan

Process Context

```
//unnecessary comments lol
struct context {
        uint edi;
        uint esi;
        uint ebx;
        uint ebp;
        uint eip;
}
```

Context Switching

In ASM!

- Switches out a process or the kernel
- Switches in the kernel or a process

```
# Context switch
#
# void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.
```

```
.globl swtch
swtch:
movl 4(%esp), %eax
movl 8(%esp), %edx

# Save old callee-saved registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacks
movl %esp, (%eax)
movl %edx, %esp

# Load new callee-saved registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

Memory Management

1. **Memory protection:** processes can rely on not having their memory interfered with by other processes
2. **Virtual memory:** processes can use memory without knowing about how other processes are using it

Memory protection unit (MPU)

- Watches traffic on bus between CPU and main memory
- Can be configured (different operations allowed in different areas)
    - CPU instructions to do it

    - Must be done in high privilege mode
- Generates **memory access violation** interrupt if forbidden memory accessed

Virtual Memory Hardware

MPU combines with virtual memory hardware to form Memory Management Unit (MMU)

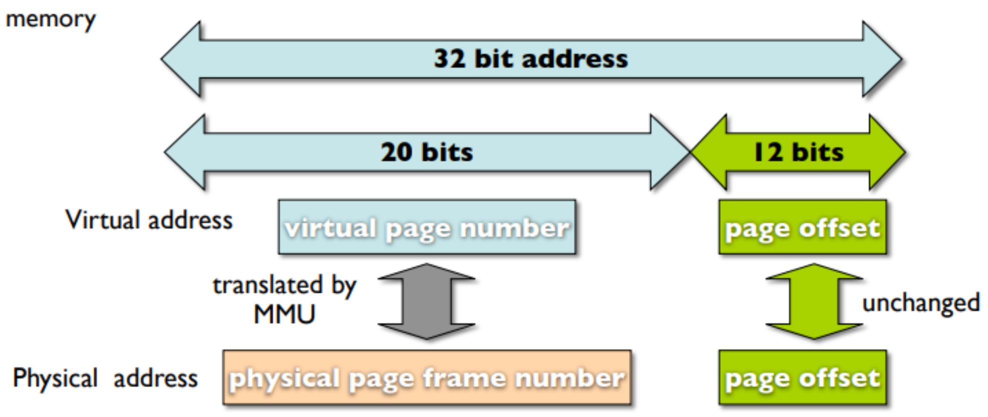
Necessary as the OS runs more processes than fit in memory - waiting processes can't fit.

Process memory management

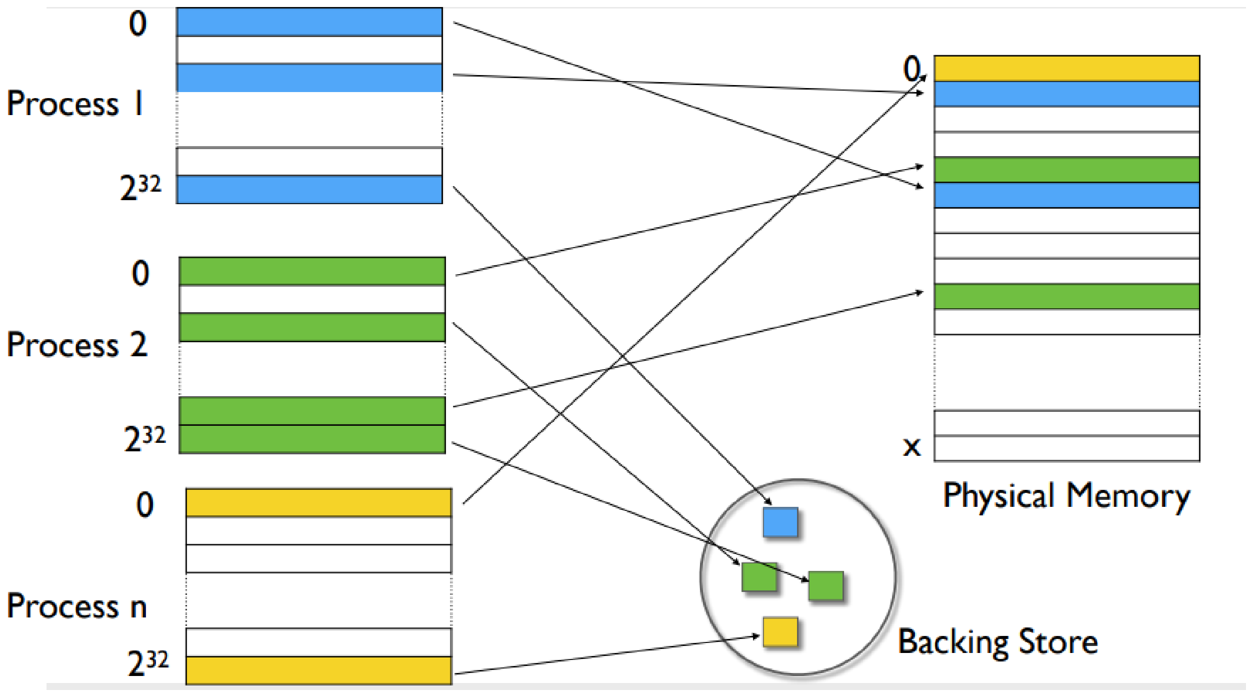
- Use another memory device to increase memory capacity for processes
- Called **swap device/page file**
- Programmer doesn't know where program is AND it moves over time
- Treat memory used by process as a set of fixed size memory chunks (pages)
    - Simplifies moving from/to swap device
    - Leads to fragmentation (process memory not contiguous)

Virtual memory

- Give each process its own virtual address space
    - (divide the space into fixed-size pages)
- Programs refer to addresses in their own virtual memory space – virtual addresses
- On the fly conversion of virtual addresses into physical addresses



- How much space?
  - Using 32-bit addresses, with address space divided into 4KB pages
  - Total addressable memory is 2^32 = 4GB
  - Each page contains 2^12 = 4KB, requiring 12 bit offset
  - 2^20 pages, each with a 20-bit page number
- Per-process virtual address space
- Size of virtual address space is usually >>>> than physical address space
  - Not every virtual page will be mapped to a physical page
  - Some virtual pages not used
  - Some pages may be mapped to swap device/page file



Virtual memory solves issues with...

Location

- In multiprogramming environment, processes and OS must share physical memory
- Programs must be paged in/out of memory
- Programmers have no way of knowing where in physical memory their processes will be located
- Processes will be fragmented

- Virtual memory gives each process its own contiguous virtual memory space, within which the program works.

Protection

- In a multiprogramming environment, user process must be prevented from interfering with memory
  - Belonging to other processes
  - Belonging to the OS
  - This protection must be implemented at run-time
- Each process only has access to its own virtual memory space, which should never map to the physical memory of another process (except under specific circumstances!)

Sharing

- Protection must be flexible enough to allow portions of memory to be shared between processes
- We can map the virtual pages of different processes to the same physical memory page frame (or same frame on disk)

Logical organisation

- Programs are organised as modules with different properties
  - Executable code
  - Data
  - Shared data
- Assign properties to virtual memory pages, e.g. read-only, executable, non-executable, etc.

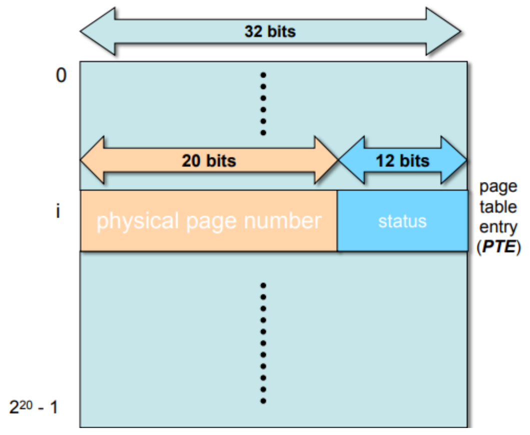
Physical organisation

- Program should be isolated from implementation of the memory hierarchy (e.g. swapping)
- Program shouldn’t be affected by amount of physical memory available
- Virtual memory hides this from the program
- Program can enquire about properties of interest, can ask for pages to be “wired” or locked into physical memory for performance or response reasons

Implementation

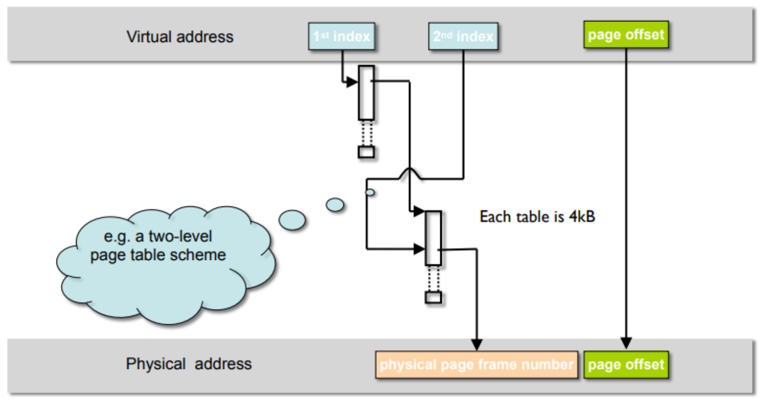
Basic

- Each process has its own page table
  - Used to map virtual pages to physical page frames
  - Page tables also stored in memory
- To translate a virtual page number to a physical page frame number
  - Use the virtual page number as an index to the page table
  - Read the physical page frame number from the table
- Example



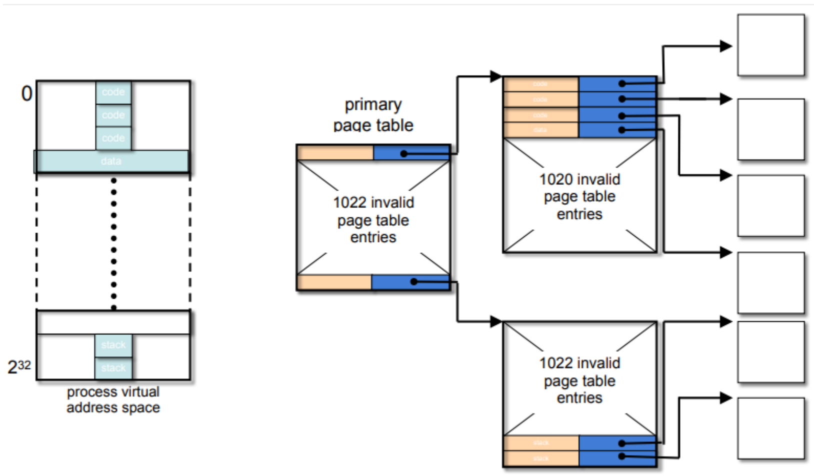
  - Only 20 bits used for address translation
  - Use lower 12 bits to store info about page
    - Valid (is the mapping correct)
    - Modified (has the page been modified)
    - Protection (read only? Read-write? executable)
    - Etc…
- Wasteful!! - with 32 bit addresses and 4KB page size
  - 1,048,576 page table entries x 20+ bits bytes each ≈ 3 MB
  - i.e. every process requires a 3 MB page table!!
  - Wasteful, when you consider that most processes will use only a small portion of the virtual address space
  - Most of the page table will be unused

Multi-level page tables



Two-level page tables

- One primary page table
- As many secondary page tables as needed
  - Created (and removed) as needed
  - Depending on how much of the virtual memory space is required by the process


0
code
code
code
data
stack
stack
2^{32}
process virtual address space
primary page table
1022 invalid page table entries
code
code
code
data
1020 invalid page table entries
1022 invalid page table entries
stack
stack

- What is the total size of all page tables in the preceding example, if two primary PTEs are used?
  - We have one primary table with two PTEs
  - each pointing to its own secondary page table
  - A total of three page-tables = 12kB

- In the preceding example, each page table was 4 kB in size so each page table fitted in exactly one page frame
    - Different schemes will use different size bit-fields for 1st and 2nd indices
    - Different schemes will use three (or more?) levels of page tables
- More efficient than a single-level scheme
- The page offset still remains un-translated.

Translation Look-Aside buffer

- Each virtual-to-physical memory address translation requires one memory access for each level of page tables
    - 2 memory accesses in a 2-level scheme
- To reduce the number of memory access required for virtual-to-physical address translation, the MMU uses a **Translation Look-Aside Buffer** (TLB)
    - An on-chip cache
    - Stores the results of the m most recent address translations
- Before walking the page tables to translate a virtual address to a physical address, the MMU checks each entry in the TLB
- If the virtual page number is found
    - Cached secondary PTE is used by the MMU to translate address,
    - No need to walk the page tables in memory
- If a TLB match is not found …
    - The MMU walks the page tables and locates the required PTE
    - The least recently used (LRU) TLB entry is replaced with
    - Current virtual page number
    - PTE from the secondary page table

Page faults

When MMU accesses PTE (page table entry), checks Valid bit of status field to determine if the entry is valid

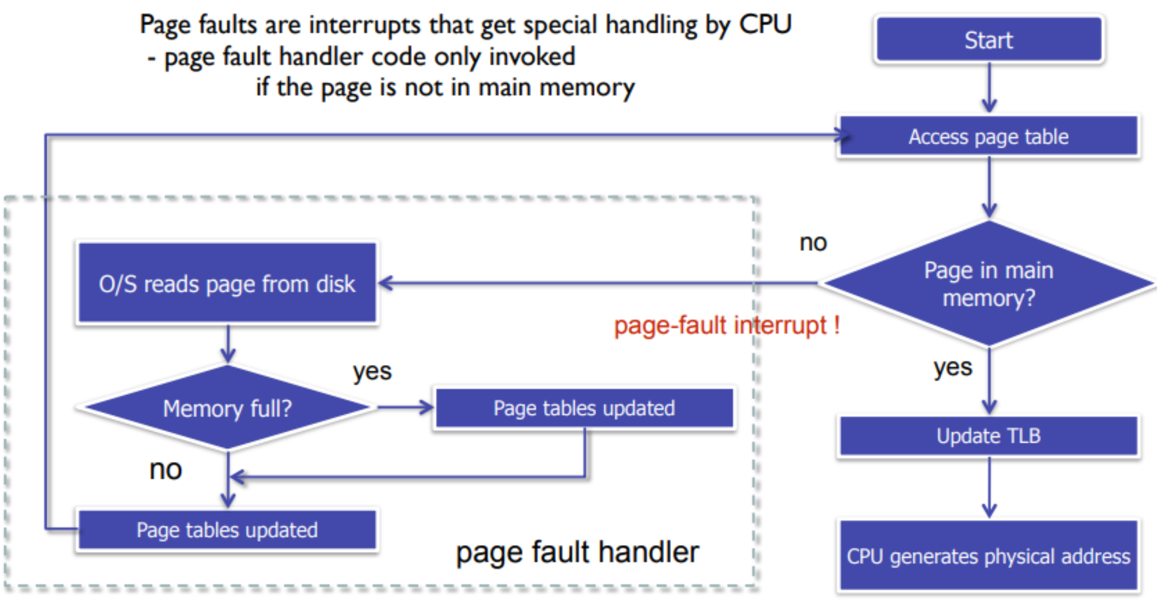
- if V==0 and a primary PTE is being accessed then a secondary page table has not been allocated physical memory

- if V==0 and a secondary PTE is being accessed then no physical memory has been allocated to the referenced page
- Either case, a page fault occurs

OS is responsible for handling page faults by

- Allocating a page of physical memory for a secondary page frame
- Allocating a page of physical memory for the referenced page
- Updating page table entries
- Writing a replaced page to disk if it is modified ("dirty")
- Reading code or data from disk (another thread will be scheduled pending the I/O operation)
- Signalling an access violation
- **Retrying** to execute the faulting instruction

Page fault handling



Non-Volatile Storage

Volatile storage: loses its memory when powered down (RAM)

Non-volatile storage: records data permanently

- persistent storage

- Tech based on magnetised surfaces (tape, hard disk) or solid-state devices based on Flash memory
- What file systems are based on

Non-volatile storage hardware

Made from

1. NV storage medium itself
2. Control hardware to manage flow of data to/from storage
    a. Data buffering
    b. Performance optimisation

Example - Disk Drive

Overview

- Rotation rate - typically 5400, 7200, 10000, 15000 rpm.
- Each platter can be considered to hold a large number of concentric tracks.
    - Each track is broken up into sectors.
    - Sectors typically 512 bytes.
- Worst case time for a byte on a track to reappear under the read/write head:
    - 1/120 second at 7200 rpm

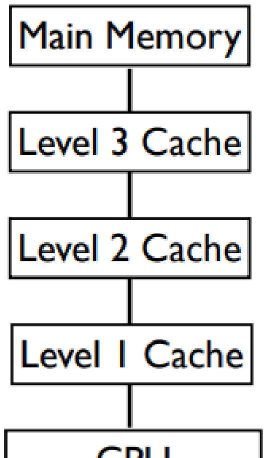
I/O

- Almost invariably, disk I/O is interrupt-driven.
- Request is made to the disk driver, and when complete, interrupt generated.
- Disk interrupt handler interacts with the device queue,
- When appropriate, the handler will call a specialised disk I/O scheduler.
- Disk I/O Schedulers try to improve performance by taking device characteristics into account:
    - Latency
    - Transfer Rate
    - Seek Time
    - Position of the data required.

Other

Memory in Different Types of Parallelism

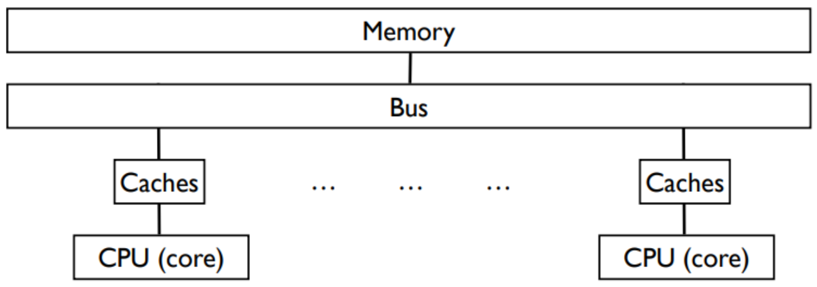
Single Processor and Memory



- Random access memory (RAM)
  - ~slow
  - ~cheap
- Hierarchy of caches between CPU and RAM
  - Level 1 - smallest, fastest
  - …

- Level 3, largest, slowest
- Register memory in CPU
    - Very fast
    - Very expensive

Multicore and Memory



Level 3 cache often shared between all CPUs.

Shared memory machine

- Each processor has equal access to each main memory location - Uniform Memory Access (UMA)
    - As opposed to Non-Uniform Memory Access (NUMA)
- Each processor may have its own cache, or may share caches
    - Leads to cache issues - consistency, line-sharing, etc.

Shells

Overview

- Program which allows you to give commands to a computer and get responses
- GUI based
    - File Explorer for Windows, etc
    - Easy to learn, hard to automate
- Command-line text-based
    - Terminal, cmd, powershell
    - Harder to learn, support high degree of automation

- Shell languages exist (SH, BASH, CSH, TCSH, ASH, ZSH) (for UNIX)
- We use bash

Principles

- Shell commands invoke other programs to do work (eg ls finds program called ls)
- Most programs work with standard character-based input sources and sinks
    - stdin, stdout, stderr
    - Pipes which can be connected to many things
    - By default
        - stdin -> keyboard
        - stdout & stderr -> terminal window

Command stuff

1. Multiple commands
    - Sequence of commands on one line separated by ;
2. Piping
    - Using | direct standard input of one **command** into the standard input of another
3. Redirection
    - Redirect standard input to accept file using <, output to file by >
4. Automation
    - Text file with sequence of commands, executed as shell script
    -