Operating Systems and Concurrency - 5CCS2OSC

Lecture 1 Processes

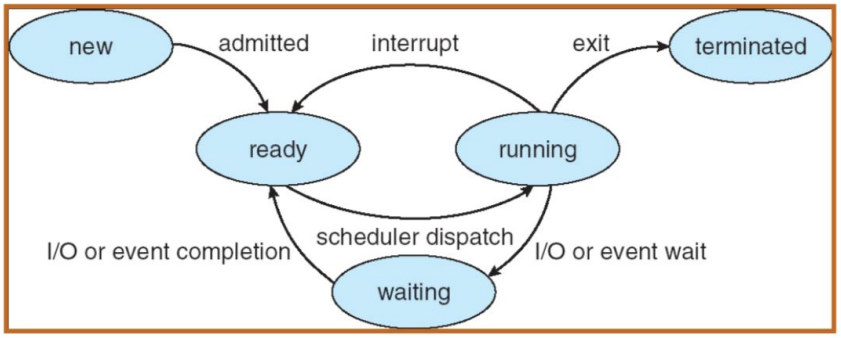
Note: teacher is obsessed with program counter. Mention it.
Process: a program that is executed. It's made up of the Stack, Heap, Data, Text.
We need to know the:

- **Program Counter**: what is the next instruction to execute.
- The current values of variables at this point.

Program: a set of machine code instructions stored on disk (or memory [if running]) that can be executed.
A user may run several instances of a single program, creating multiple processes for a single program.

- If you run 3 copies of notepad: 1 copy of the program will be loaded into memory and 3 processes are created each maintaining their own position in the source code and the current values of variables.

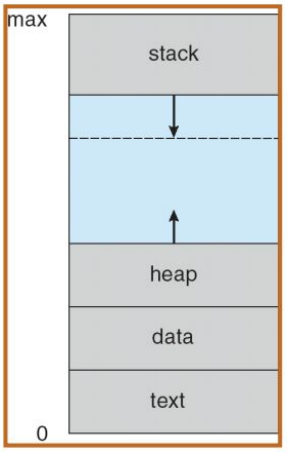
Process States



Processes in Memory

Components of a process: (*mnemonic "Some Highly Destructive Tornadoes")*

- **Stack:** contains temporary data:
  Function parameters, return addresses, local variables.
- **Heap:** dynamically allocated memory (e.g. Objects created using new). The memory the process can use.
- **Data:** global variables (e.g. Java Static variables)
- **Text:** code (compiled binary) for the program the process is executing.



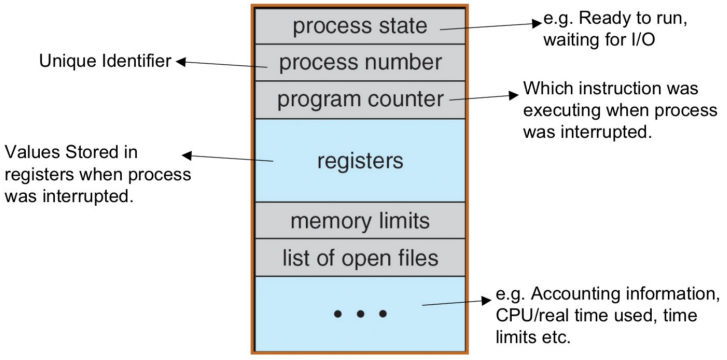
***NB,* register values and programme counters** are **stored** either **in the registers (when the process is executing)** or in the **PCB** (when its not)

Process Control Block

Process Control Block: a structure maintained by the operating system in order to keep track of where a process is up to in execution. Stored separately from the process itself (in the OS).

Contains:

- Program Counter: stores which instruction was executing when the process was interrupted.
- Process Number
- Process State
- Registers
- Memory Limits
- List of open files



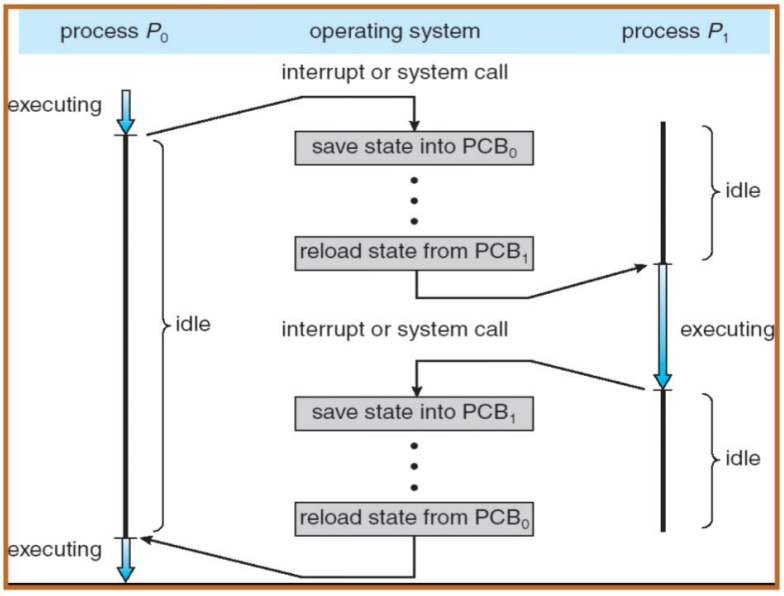
NB, a processor core can only run one process at a time.

Context Switching

Context Switching: the process of taking one process that's running on the CPU, stopping that process and starting another process.
What happens:

- Stores values of a process in the PCB so that it can be stopped (including the counter).
- Then takes another PCB from another process, loads in the CPU registers of it and resumes execution of that process.

Context switching takes time thus creates an overhead.



Time consumed when switching is time where the CPU is idle thus it's not doing useful work. However, context switching is important because:

+ it allows us to "simulate" that a computer has multiple applications running concurrently on a single CPU.
+ Improves productivity since **I/O bound processes** can run at the same time as **CPU bound processes** since most processes only require a short CPU burst
  E.g. I can listen to music whilst doing my PPT.

I/O process: input/output means communicating information between the computer and the outside world (human or other computer).

- I/O process might be a disc defragmenter.
- CPU bound a video encoder.

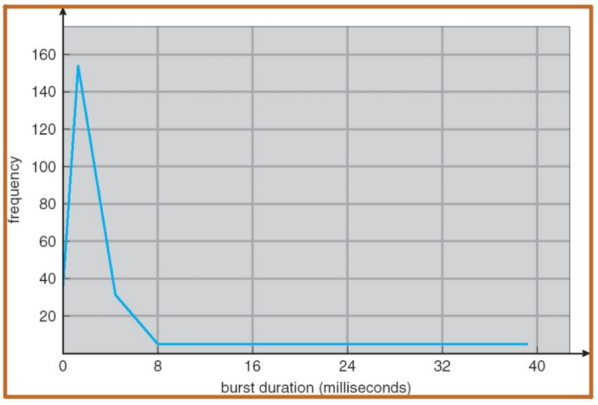
NB, *whilst you can only run one process per core at a time however a hard drive can work independently from the CPU. Example:*

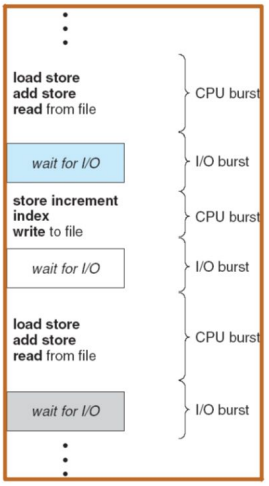
- Process 1 runs on CPU: requests to start reading from hard disk.
- Process 2 now runs on CPU, and whilst process 2 is running on the CPU the requested data from the hard drive is transferred to memory for process 2: this does not require the CPU to do once it's started.
- Process 1 stops running on the CPU and process 2 now runs to do some computations on the CPU based on the data that was loaded earlier.

So no two processes can run on the CPU at once, but copying data from a hard disk to memory doesn't need the CPU, so one process can do this, whilst the other uses the CPU.

CPU Burst Time and Process Behaviour (respectively)

Each process will do a very short CPU burst, it will then do some I/O and then do a CPU burst again.





Process Scheduling

Different Queues in an OS

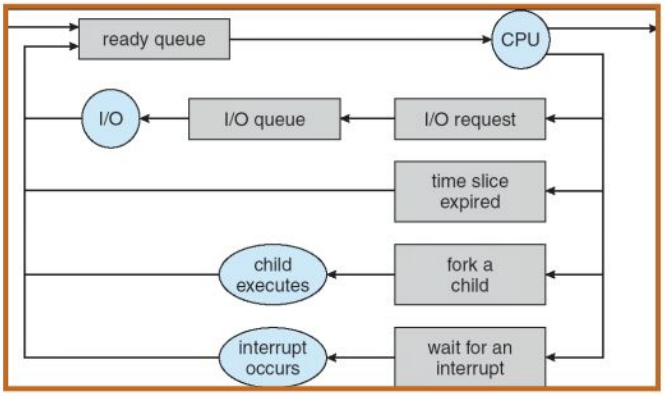
Job Queue: all processes in the system.
Ready Queue: processes waiting for the CPU.
Device Queues: one per device, processes waiting for that device.
During execution of a process it will change queues.

Phases of Scheduling

There are two separate stages for scheduling.

- **Long Term Scheduler** determines how many processes can run at once by determining which ones go to the ready queue, runs infrequently (order of seconds).
- **Short Term Scheduler** determines which processes from the ready queue get to run in the CPU right now, runs very frequently (order of milliseconds).

Every process starts on a ready queue (added by the long-term scheduler), executes in the CPU for some time and then is moves to other queues. If the queue has a mix of CPU and IO bound processes good efficiency can be achieved by running them simultaneously.



CPU Scheduling Algorithms

Efficient CPU Scheduling is essential for good utilisation of the processor and responsiveness of the OS.
Criteria for Scheduling Algorithm Performance:

- **CPU Utilisation:** percentage of CPU used when executing a process
- **Waiting time:** time a process spends waiting in the ready queue
- **Turnaround time:** time between a process arrives in the queue and finishes executing.
- **Response time:** time between arrival of the process and the production of the first response (it may not need to complete in order to respond)
- **Throughput:** number of processes completed by unit of time

We can also ask for **average** waiting/turnaround/response time or total **sum** of those or **maximum/minimum** waiting time for these set of processes.

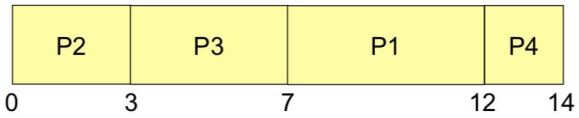
Scheduling Algorithms

- First Come, First Serve (FCFS)
- Shortest Job First (SJF)
- Priority Scheduling
- Round Robin (RR)

First Come, First Serve

Whoever arrives first, runs first.

Process	Arrival	Duration
P1	4	5
P2	0	3
P3	2	4
P4	12	2



Average Waiting Time = (0 + (3 – 2) + (7 – 4) + (12 – 12)) / 4 = 1

Average Turnaround Time = (3 + (7 – 2) + (12 – 4) + (14 – 12)) / 4 = 4.5

Throughput = 4 / 14 = 0.29

Waiting Time = time started - time arrived + extra idle time
Turnaround Time = time completed execution - time it arrived

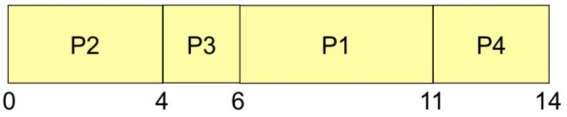
\+ Simple algorithm.
\+ Only one context switch per process.
\- Average waiting time is typically poor.

- Average waiting time is highly variable (depends on order in which processes arrive). If the longest process arrives first, it will make the rest wait for a long time.
- CPU bound processes can hog the CPU. Imagine a system with one CPU bound process and many I/O bound processes. A long CPU burst will run for long until it's finished and the I/O bound processes will be waiting.

Shortest Job First

If the shorter processes went first, then we would have a shorter waiting time (as seen before). **Note: jobs can only be scheduled when they arrive, not before.**

Process	Arrival	Duration
P1	0	5
P2	0	4
P3	1	2
P4	10	3



Average Waiting Time = (0 + (4 – 1) + (6 – 0) + (11 – 10)) / 4 = 2.5

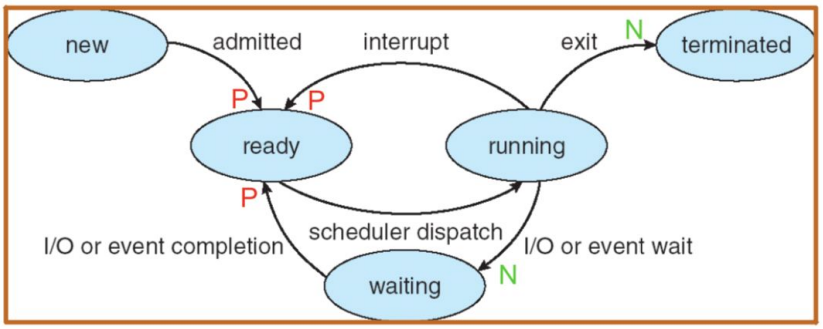
Average Turnaround Time = (4 + (6 – 1) + (11 – 0) + (14 – 10)) / 4 = 6

Throughput = 4 / 14 = 0.29

Preemptive Scheduling

Non-preemptive scheduling: the process that is being executing on the CPU continues until it's finished.
Preemptive scheduling: if another process arrives, the process that is running can be stopped and a new one started.

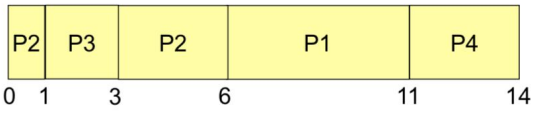
Scheduling decisions occur when a new process is added to the ready queue (P below can cause new scheduling decisions)



Shortest Remaining Time First (SRTF)

Also called preemptive shortest job first.
(Preemptive) If a new process, shorter than the remaining time of the current process arrives, we will do that now.

Process	Arrival	Duration
P1	0	5
P2	0	4
P3	1	2
P4	10	3



Average Waiting Time = (6 + 2 + 0 + 1) / 4 = 2.25 *(vs 3.5 for FCFS)*

Average Turnaround Time = ((11 – 0) + (6 – 0) + (3 – 1) + (14 – 10) / 4 = 5.75

Turnaround time = waiting time + execution time

+ Short processes do not have to wait for long processes to complete before executing. CPU bound processes don't hog the CPU.
+ Maximises throughput.
+ Average waiting time is smaller (for the same set of processes) since no process waits for the longest to complete.
- Risk of starvation* of long processes or long waiting times.
- Multiple context content switching (overhead).
- We cannot reliably estimate process execution time before it executes it.
- Inserting processes in a sorted queue creates overheads.

***starvation:** when a process is denied necessary resources to process its work.

Priority Scheduling

Associate a priority with each process. We assign a priority integer to each process. The lower the number means a higher priority (in windows it's the opposite).
We will schedule the process with the highest priority first (that is the lowest number)

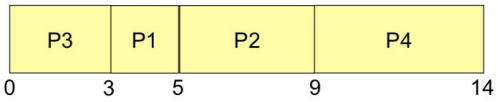
- We can do non-preemptive or preemptive.

SRTF is a type of priority scheduling, the priority level is determined by the remaining executing time.

Non-preemptive Priority Scheduling

Add jobs to at the appropriate position in the ready queue according to their priority.

Process	Arrival	Duration	Priority
P1	2	2	3
P2	0	4	4
P3	0	3	0
P4	6	5	2



Average Waiting Time = (1 + 5 + 0 + 3) / 4 = 2.25

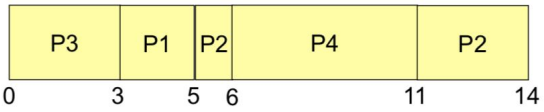
Average Turnaround Time = ((5 – 2) + (9 – 0) + (3 – 0) + (14 – 6) / 4 = 5.75

QUESTION: What occurs if two processes have the same priority and arrive at the same time.

Preemptive Priority Scheduling

If a process arrives with higher priority than the one currently executing, execute it, otherwise insert it in the appropriate position in the ready queue according to its priority.

Process	Arrival	Duration	Priority
P1	2	2	7
P2	0	4	14
P3	0	3	0
P4	6	5	3



Average Waiting Time = (1 + 10 + 0 + 0) / 4 = 2.75
(notice individual waiting time vs priority)

Average Turnaround Time = ((5 – 2) + (14 – 0) + (3 – 0) + (11 – 6) / 4 = 6.25

+ Short waiting times for high priority processes (e.g. interactive user programs).
+ Users (or admin) have some control over the scheduler.
+ Deadlines can be met by giving processes high priority.
+ We can use ageing, which gradually increases priority (reduces number) as time passes.
- Risk of starvation for low priority processes or long waiting times.
- Multiple context switches for each process if pre-emptive.
- Overheads in inserting processes in a sorted queue.

Round Robin Scheduling (RR)

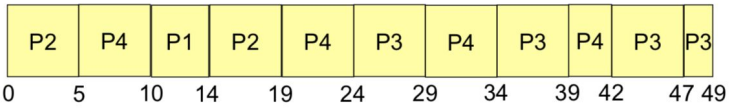
- Features a **time quantum**: the length of time each process is allowed to run for (usually 10-100ms)
- After this time it is preemptived and another process from the ready queue can be scheduled.
- Performance varies according to q
    - If q is large round robin tends to be FCFS.
    - if q is small, context switching overheads become significant.
- Heuristic for efficiency: a q that is longer than 80% of the CPU bursts should be chosen.

*The first job from the queue is taken and made run for q units, then we place it at the **back** of the queue, new processes join the back of the queue **on arrival.***

Round Robin (q = 5)

*Take the first job from the queue (FCFS) and schedule it to run for q units. Then place it on the **back** of the queue. New processes join the **back** of the queue **on arrival**.*

Process	Arrival	Duration
P1	4	4
P2	0	10
P3	17	17
P4	2	18



Average Waiting Time = (6 + 9 + 15 + 22) / 4 = 13

Average Turnaround Time = ((14 – 4) + (19 – 0) + (49 – 17) + (42 – 2) / 4 = 25.25

- **Q0: P2(10),**
- **Q5: P4(18), P1(4), P2(5)**
- **Q10: P1(4), P2(5), P4(13)**
- **Q14: P2(5), P4(14)**
- **Q19: P4(13), P3(17)**
- **Q24: P3(17), P4(8)s**
- **Then they go each one by intervals of 5 until one is done, then the other finishes.**

\+ No Starvation
\+ Each process must wait no more than (n-1)* q time units before beginning execution
\+ Fair sharing of CPU between processes
\- Poor avg response time
\- Waiting time depends on nº of processes rather than priority
\- Particularly large nº of context switches for each process so large overhead
\- Hard to meet deadlines because waiting times are generally high.

There is no monotonic relationship between quantum and average turnaround time. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

Multilevel Queue Scheduling

If processes can be divided into groups we can use multilevel queue scheduling:

- E.g. 1 finite number of priorities, one queue for each priority level.
- E.g. 2 a queue for foreground processes and another for background processes.

Idea: Partition the ready queue into several queues

- Each process is assigned to a specific queue.
- Each queue has a different scheduling algorithm.
- Note that we will also need a scheduling algorithm to choose between queues.
    - Typically fixed priority preemptive
    - Sometimes foreground may have absolute priority: background processes only run when the foreground queue is empty.

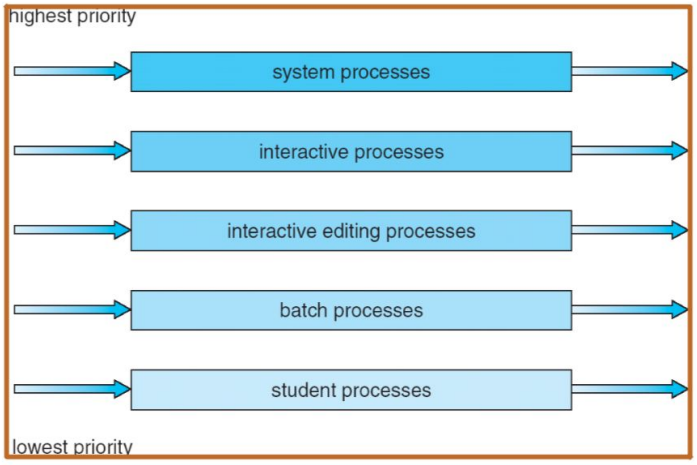
Example

Scenario 1:

- Each queue has absolute priority over lower priority queues
- e.g. no process in the batch queue can run unless the queues above it are empty
- This can result in starvation for the processes in the lower priority queues

Scenario 2:

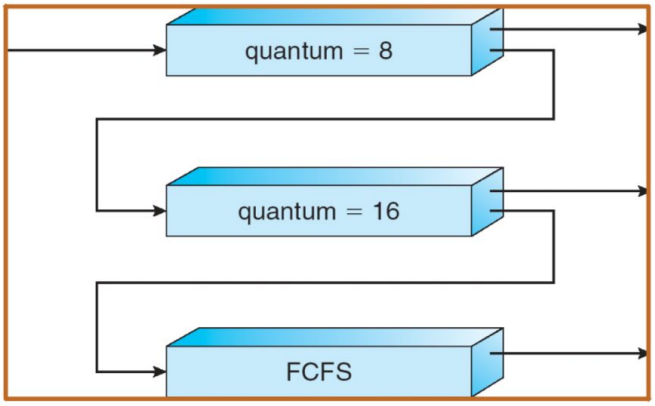
- Time slice amongst queues;
- Each gets a fixed percentage of CPU time.
- e.g.
  - 40% System
  - 30% Interactive
  - 20% Interactive Editing
  - 6% Batch
  - 4% Student
- Again each queue can use its own scheduling algorithm.



Multilevel Feedback Queue Scheduling

- Like Multi-Level Queue but processes can move between queues
- Can use this to implement ageing
- In addition to the queues and their scheduling algorithms we need:
    - Mechanisms to decide when to promote processes.
    - Mechanisms to decide when to demote a process.
    - Mechanism to decide which queue a process will join initially.

Example



- All new jobs enter the top queue (RR q=8);
- If not complete after 8 time units of execution they move to the next (RR q = 16);
- If still not complete after 16 time units of execution the move to the final queue which operates FCFS.

Priority Scheduling via MLFQ

Create n queues, one for each priority level. Higher properties get allocated more CPU time.

- Each process enters in the queue corresponding to its priority.
- After a process has been in the system for some time, its priority is increased so it moves to a higher queue (aging).

Practise May 2016 4b

Multiprocessor Scheduling

- Multiple CPUs can be used to share load
  - Problem becomes more complex with CPUs with different capabilities
  - We focus on CPUs with the same capabilities
- Two variants:
  - Asymmetric multiprocessing (one processor controls all scheduling).
  - Symmetric multi-processing (each processor schedules itself).

Asymmetric vs Symmetric

Asymmetric multiprocessing (ASMP)

- One processor makes all the scheduling decisions, and handles I/O processing and other system activities.
- The other processors execute user code only.
- The need for data sharing is reduced because only one processor handles all system processes.

Symmetric Multiprocessing (SMP)

- Each processor does its own scheduling
- There may be a single ready queue or one for each processor
- Whichever is used each processor selects a process from the ready queue

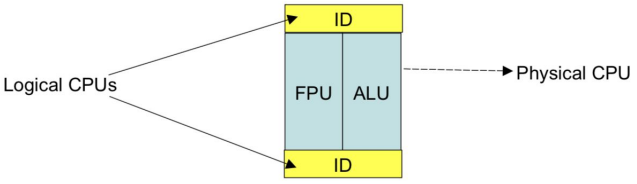
- Efficient CPU use requires load regulation so that we don't have a very busy processor and one that is idle, two approaches:
    - **Push** migration: specific process regularly checks the load on each and redistributes
    - **Pull** migration: an IDLE processor pulls a waiting job from the queue of a busy processor

All major processors use symmetric multithreading

Hyperthreading

Multiprocessing uses multiple (physical) CPUs to run processes in parallel.
Hyperthreading (aka Symmetric Multithreading): allows the same thing but using multiple logical processors. Basically creating virtual processors in one processor.

- Logical Processors:
    - Share the Hardware of the CPU (cache, busses ...)
    - Are responsible for their own interrupt handling)
- Appears as 2 CPUs instead of one. One can do floating point computations, another is doing Arithmetical/Logical computations.



Scheduling in Windows (XP)

- Pre-emptive priority scheduling, with time quanta. Priority levels are integers from 0 to 31 (unusually: high number == higher priority)
- When a process is selected to run (highest priority ready thread) it executes until either:
    - It's time quantum is used.
    - Another higher priority process arrives.
    - It terminates.
    - It makes a system call e.g. I/O request.
- A queue is maintained for each priority level and a process is selected from the queue with the highest priority. If no thread is available for execution a special "system idle process" is executed. This does nothing and it saves us from having to program for special cases in the scheduler, if you assume that there is always a process running, then you can program easier.
- Pre-emption by higher priority threads ensures that real-time threads can access the CPU when required.

Priorities in Windows (XP)

- Processes start at the base priority of their class (inherited from parent)
- Priorities (excluding Real time) are variables within their class.

- When a process completes its time quantum its priority is lowered (never below the base priority for its class). This prevents CPU hogging.
- When a process is done waiting for some I/O bound process, then its priority is boosted so that it's more likely to get to run.

- Gives good I/O device utilisation whilst allowing CPU bound processes to use spare CPU cycles.
- Current active window also given a boost for responsiveness.
- Interactive processes being run by the user typically all boosted are generally given a quantum 3 times as long.

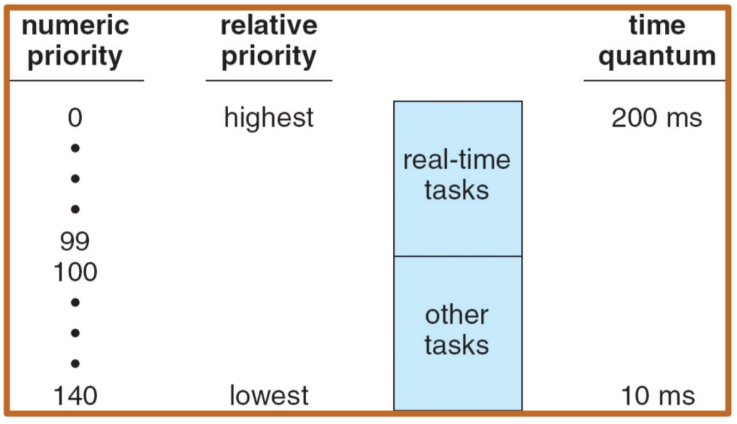
Win32 API Priority Class

Priority Within Class

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Scheduling in Linux (2.5)

- Linux uses pre-emptive priority based algorithm with two separate priority ranges:
  - A real-time range from 0 to 99 (critical)
  - A nice value ranging from 100 to 140
- These two ranges map into a global priority scheme: lower values => higher priority.
- Higher priority tasks receive longer time quanta than lower priority tasks.



Linux (2.5) Scheduling Algorithm

- A runnable task is eligible for execution if it has time remaining in its quantum.
- When a task has exhausted its quantum it is expired and not eligible for execution again until the rest have exhausted their time quanta.
- Because of its support for SMP each processor has its own run queue and schedules itself independently.
- The scheduler selects the eligible task with the highest priority for execution.

- When all tasks have exhausted their time slices all tasks become eligible for execution.

Priorities in Linux (2.5)

- Real time processes have static priorities
- All other tasks have dynamic priorities, based on their nice value and the number 5.
  - Interactivity of a task is determined by the time spent sleeping waiting for I/O;
  - I/O bound processes will sleep for more time than CPU bound processes.
  - For highly interactive processes (lots of I/O) 5 will be subtracted from their priority, thus it is increased.
  - For processes with very little interaction (CPU bound) 5 is added to their priority and thus it is lowered.
- Priority is recomputed on expiration of the time quantum

Scheduling Algorithm Evaluation

Deterministic modelling: take a set of predetermined processes and run through the algorithm.
Queueing model:

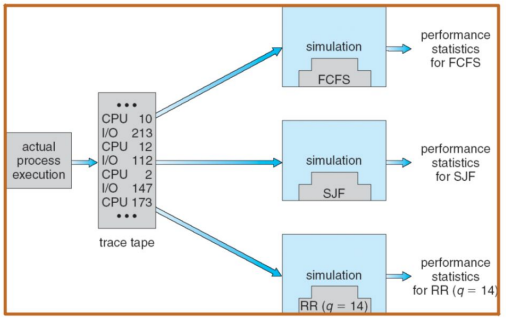
- Use queueing theory to predict performance.
- Knowing the **arrival rates** and services, we can compute utilization, average queue length, etc…
  Little's formula **($n = \lambda \times W$):**
  - Queue length (n), arrival rate (λ), waiting time (W). e.g.
  - 7 processes arrive per second;
  - Queue length normally 14 processes;
  - Therefore average waiting time is 2 seconds.
-
- **DO NOT USE THIS FORMULA FOR CALCULATING AVERAGE WAITING TIMES UNLESS GIVEN AN ARRIVAL RATE, IF GIVEN A SET OF PROCESSES USE DETERMINISTIC MODEL**

Problems with these approaches: unrealistic, hard to model complex algorithms using queueing theory. Arrival rate and time are hard to represent accurately.
We can also evaluate these algorithms using "Evaluation via implementation" that involves actually coding the example and testing it.

Evaluation via Implementation

The only way to accurately evaluate an algorithm is to implement it in the OS

- Lots of work
- User suffer inconsistent performance
- Users may adapt to 'cheat' scheduler
- The ideal is to have a scheduling algorithm that can be configured by the System Admin



- Allows tailoring to the specific system requirements or an API that allows users to change thread priorities

Child Process Creation (Java Implementation)

A process can create child processes (becoming their parent process) which, can themselves create create other processes, forming a process tree.

- **Resource sharing options:** parent and children can share all, some or no resources.
- **Execution options:** parent and children can execute concurrently or parent can wait until some or all of its children have terminated.
- **Address space options:** child process can be a duplicate of the parent (have the same program data) or the child may have a new program loaded into it.

We use an ArrayList and add things to it. We use the class **ProcessBuildier**.

```
public void execute(ArrayList<String> commmand) {
  try{
    ProcessBuilder builder = new ProcessBuilder(command);
    Map<String, String> environ = builder.environment();
    Process p = builder.start();
  } catch (Exception e){
    e.printStackTrace();
  }
}
```

```
ArrayList<String> commands = new ArrayList();
command.add("explorer.exe");
command.add("myHelpFile.pdf");
execute(commands);
```

- Map gives you environment variables that you may need from the computer for the process. E.g. if you need the current working directory.
- Everything you do in java that requires system calls is put into try catch since it can throw an exception.

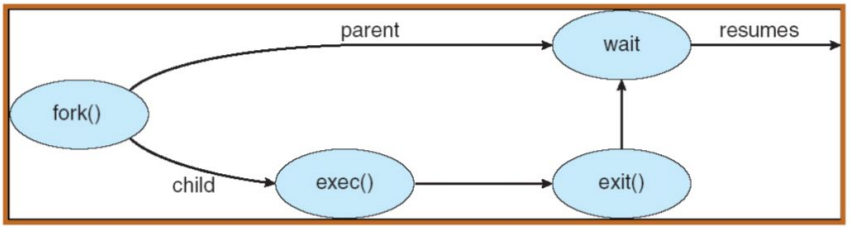
```
public void executeAndPrint(ArrayList<String> commmand) {
  try{
    ProcessBuilder builder = new ProcessBuilder(command);
    Map<String, String> environ = builder.environment();
    Process p = builder.start();
    p.waitFor();   //or not…
    BufferedReader br = new BufferedReader(new InputStreamReader(
                        p.getInputStream()));
    String line;
    while ((line = br.readLine()) != null){
    System.out.println(line);
    }
  } catch (Exception e){
    e.printStackTrace();
  }
  System.out.println("Program terminated!");
}
```

```
ArrayList<String> commands = new ArrayList();
command.add("cat");
command.add("PawsTooSlippy");
executeAndPrint(commands);
```

- p.waitfor() is where you decide as a parent if you wait for your child to execute or not.
- Java relies on process creation in unix.

Process Creation in Unix

- **fork()** system call creates a new process (child process).
- **exec()** system call used after fork to replace the memory space of the process with a new program
- Can use the commands in C or implicitly using the above Java



Process Termination

When a process executes its last statement it asks operating system to clear memory and terminates **(exit)**. (Each process normally returns a value when it terminates e.g 0)

- Exit status is returned from child is received by parent (via *wait()*) it indicates success or failure (abnormal exit).

Parent may terminate execution of child processes with ***kill()***

- If a child has exceeded allocated resources
- If the task assigned to child is no longer required
- At will

If parent is exiting

- Some operating systems do not allow child to continue if its parent terminates.
- All children are terminated - cascading termination.

Lecture 2 Threads

Thread: a lightweight process, basic unit of CPU usage.

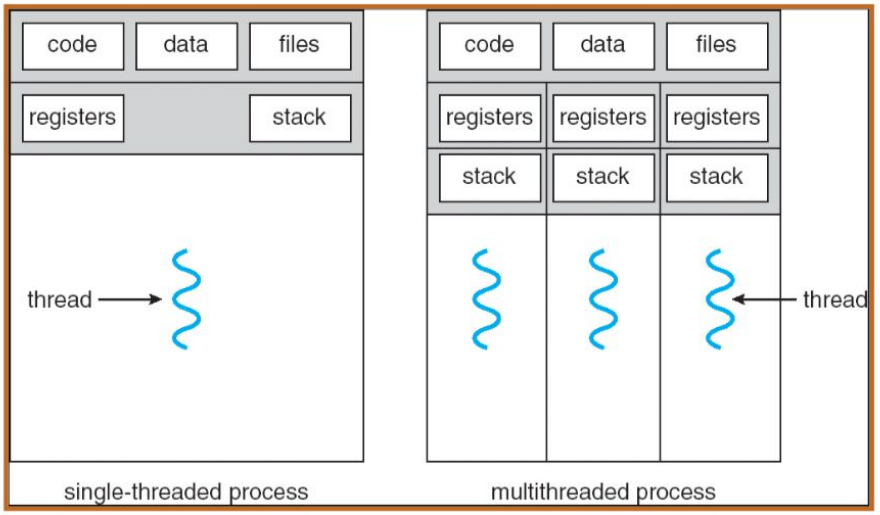
- Has a thread id
- Program counter
- Register Set
- Stack
- Each thread must maintain a register and a stack since they be at different points of execution.

It shares **its code section and data section and other operating-system resources (files)** with other threads belonging to the same process.

- This implies that when a thread is created its possible that you don't have to allocate it any memory, because it just uses the memory already allocated for use in another thread. The only thing it needs to store individually is its own stack and program counter (which does need to be stored somewhere in memory) but it doesn't need it's

own *block of memory* dedicated for other use necessarily, as it can share with other threads if appropriate.

A traditional process has a single thread of control, if it has multiple threads of control it can do more than one task at a time.



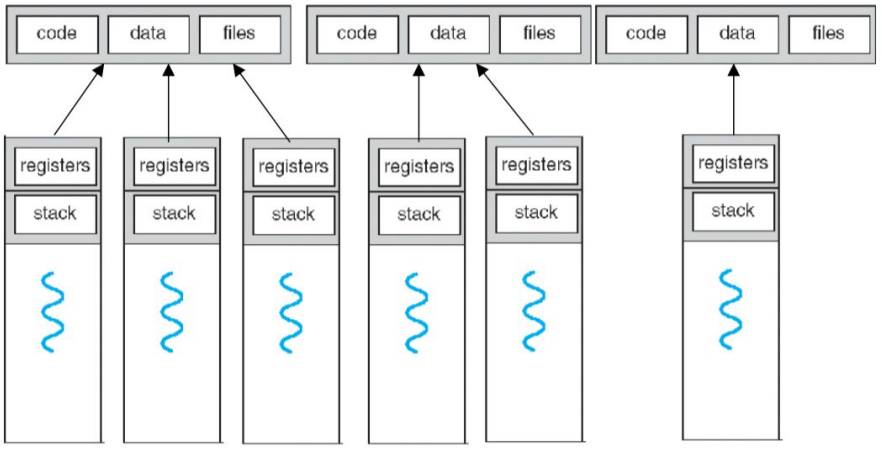
Types of Processes

Process

- Isolated with its own virtual address space.
- Contains process data like file handles
- Lots of overhead
- Every process has *at least* one kernel thread.

Kernel Thread

- Shared virtual address space
- Contains running state data
- Less overhead (because it can share code, data or files)
- From the OS's POV this is what is scheduled to run on a CPU.



Linux and Windows do not distinguish between processes and threads, multi-threaded processes make many threads that share resources.

User Thread

- DIY thread. Creating threads yourself and scheduling them yourself.
- Kernel is unaware, thus it appears as a single threaded process.
- Less overhead at an operating system level (not necessarily less at another since we are programming it).

Kernel Mode: is the mode in which important functions of the operating system are run. As opposed to user mode, which is where user code is run.

Multi-Threading

A CPU may only run one thread at a time but it can be running in the background (it can add my typing to a buffer whilst I do something else, since typing is I/O.)
Multi-threading is used by most of the software we run

E.x. Web browser: might have a thread display images or text while another thread retrieves data from the network.

This is good because if one thread is blocked waiting for I/O (e.g. loading an image) the remainder of the threads can still execute and remain responsive (Java programs with GUIs are multi-threaded automatically).

Other uses of multithreading:

- Web server serving many connected clients at once.
- Expensive computation can be separated into multiple machines

+ Responsiveness: if one thread is blocked (waiting for I/O) the other threads in the process can still run, meaning the user can still interact with the program.
+ Resource sharing: avoids duplications of code, data and files and allows threads to interact with each other easily.
+ Efficiency: threads are cheaper to create (no block memory assignment {see definition of thread}) and to context switch
+ Multi-processor Utilisation: a single threaded process can ever run on one CPU but with multi-threading we can distribute these amongst CPUs: more concurrency = faster execution time.

Process vs Thread

Although a process can do what a thread does it creates more overhead (the creation of a thread is much cheaper since we don't need memory space). Additionally, since the same data, variables and code will be required to service each client there's no point in duplicating that.

Creating Threads Myself

I can write some code that appears to be a single kernel thread but internally it runs some thread for some time and then another etc (these are user threads). In this case this is one kernel thread and when it is scheduled by the OS, that is when it will run. Instead, I can make a Kernel Thread for each User Thread I'm doing and the OS and it will schedule it (this

is what Windows and Linux do, this makes sense since we don't want everyone that writes a thread to write a scheduler as well).

Threading in Java

You can **extend the Thread class** or if you have to inherit something you can **implement Runnable**. They require you to write the method **public void run()** which will state what you want your thread to do.

```
public class PrintP extends Thread {


  public void run(){
      for(int i = 0; i < 100; ++i){
          System.out.println("P");
      }
  }
}
```

```
public class PrintQ implements runnable
    extends SomethingElse {

  public void run(){
      for(int i = 0; i < 30; ++i){
          System.out.println("Q");
      }
  }
}
```

ERROR extends Runnable with capital R

```
public class myMain {
  public static void main(String[] args){
    Thread p = new PrintP();
    Thread q = new Thread(new PrintQ());
    p.start();
    q.start();
    for(int i = 0; i < 200; ++i){
      System.out.println("R");
    }
  }
}
```

(Extended Version)

```
Runnable qRun = new PrintQ();
Thread q = new Thread(qRun);
```

To start a Thread (1st way) you must call **start()** (if you call run() it skips the creating a new thread part) **YOU MUST USE START NOT RUN.**

Our intuition says that there are two threads running, p and q but there are actually 3. When we start p and q they are going to run in different new threads. But note that the main method is also running during the execution of the threads (and is a thread). The result of the program is **unpredictable** because its at the mercy of the scheduler.

Thread.sleep()

```
public class PrintP extends Thread {
  public void run(){
      for(int i = 0; i < 100; ++i){
          System.out.println("P");
          try{
            Thread.sleep(30);
          } catch (InterruptedException e){
            e.printStackTrace();
          }
      }
  }
}
```

```
public class PrintQ implements runnable
    extends SomethingElse {
  public void run(){
    for(int i = 0; i < 30; ++i){
      System.out.println("Q");
    }
    try{
     Thread.sleep(50);
   } catch (InterruptedException e){
          e.printStackTrace();
    }
  }
}
```

Sleep stops the execution and allows other threads to run, when threads sleep they enter a blocked state so context switches happen.

Thread.join()

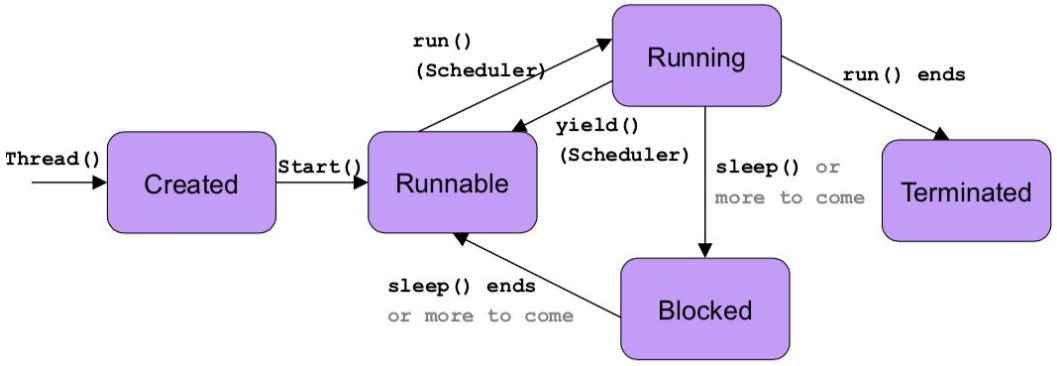
- **p.join()** - will make another thread wait until the one that is called upon is finished. p.join(), P has to finish before other things can run.

Interrupted Exception

"Life as a thread in java is quite risky since parent processes can kill their threads"
There are two methods for killing threads:

- **Asynchronous Cancellation:** the thread is killed immediately.
- **Deferred Cancellation:** the thread checks periodically to see whether it is to terminate

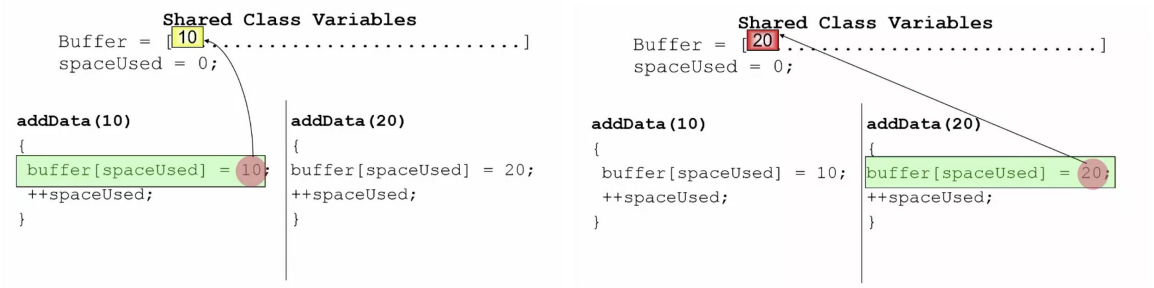
In Java, a thread can be killed whilst its asleep. If so, it will throw a InterruptedException which can then be handled.
For now:



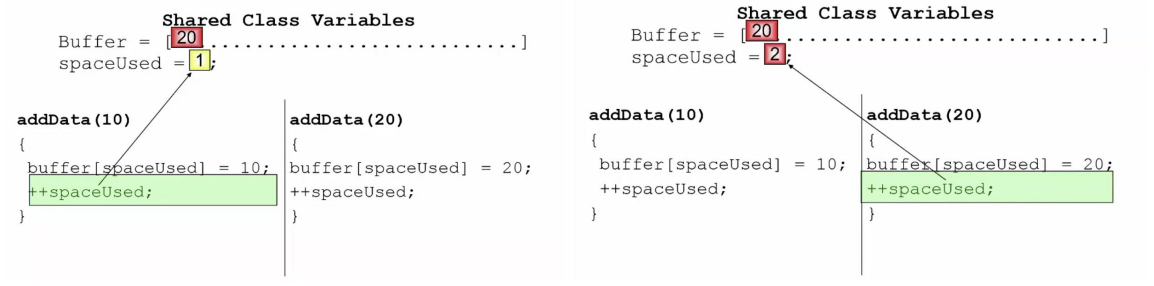
Problems with Running Threads in Java

If we create two threads that use the same buffer (the same array in memory). They **use the same array in memory, not a copy** this can lead to some problems**.**

1st



3rd



Note each new image represents a context switch

We don't know in what order the threads will run. This can lead to data loss (as scene in the images above) if context switching occurs before we increment spaceUsed. Sometimes it will crash, sometimes in won't. This implies that bug fixing is harder since it may crash sometimes with the same inputs.

It can get even more complicated since each line of code can be non-atomic (that is that it creates multiple machine instructions. ++spaceUsed results to:

```
mov R1, .spaceUsed   //put address of spaceUsed into Register R1
ldm R0, [R1]         //load to R0 from memory address in R1
add R0, R0, 1        //R0 = R0 + 1
stm R0, [R1]         //store R0 in memory address in R1
```

Therefore, a *line in Java code* may not always be atomic.

Interleaving

Each line leads to multiple machine code instructions (CPU runs machine code not Java).

```
mov R1, .spaceUsed   //put address of spaceUsed into Register R1
ldm R0, [R1]         //load to R0 from memory address in R1
add R0, R0, 1        //R0 = R0 + 1
stm R0, [R1]         //store R0 in memory address in R1
```

When we do context switching it results in something like:

```
mov R1, .spaceUsed    //put address of spaceUsed into Register R1
ldm R0, [R1]          //load to R0 from memory address in R1
```

R1 .SpaceUsed R0 0 => Context switch store R0,R1 in memory

```
mov R1, .spaceUsed    //put address of spaceUsed into Register R1
ldm R0, [R1]          //load to R0 from memory address in R1
```

R1 .SpaceUsed R0 0 => Context switch store R0,R1 in memory load R0,R1.

```
add R0, R0, 1         //R0 = R0 + 1
stm R0, [R1]          //store R0 (1) in memory address in R1
```

R1 .SpaceUsed R0 1 => Context switch store R0,R1 in memory load R0,R1.

```
add R0, R0, 1         //R0 = R0 + 1
stm R0, [R1]          //store R0 (1) in memory address in R1
```

R1 .SpaceUsed R0 1 => Now spaceUsed = 1

Interleaving: mixing, alternating between.

Atomicity

Atomic Statement: a single statement that cannot be interrupted (we cannot split the atom in CompSci).
Concurrency: is the interleaving of atomic statements (for now we are going to assume that assignment statements are atomic).

Although we treat assignments as atomic, in our array example we have other statements which aren't atomic, to solve this we have something called the **critical section**.

Critical Sections in Java

A critical section: a part of the program which needs to be executed atomically (by themselves). *If multiple threads enter the critical section at the same time we might get errors, as seen in the array example (green boxes). NB the use of the word might since it might not happen, it all depends on how the program interleaves.*

```
addData(10)
{
    buffer[spaceUsed] = 10;
    ++spaceUsed;
}
```

```
addData(20)
{
    buffer[spaceUsed] = 20;
    ++spaceUsed;
}
```

Java provides the keyword **synchronized** which can be used to mark a method as a critical section

- If used only one instance of the given object can be run at once

- We can mark multiple methods as synchronized but none of these methods can run at the same time.

Synchronisation example (it excludes error handling: try, catch)

```
public class Buffer {
    int[] buffer;
    int spaceUsed;

    public Buffer(int size){
        buffer = new int[size];
        spaceUsed = 0;
    }

    public synchronized void add(int toAdd){
      buffer[spaceUsed] = toAdd;
      ++spaceUsed;
    }

    public synchronized void printBuffer(){
        System.out.print(buffer[0]);
        for(int i = 1; i <= spaceUsed; ++i){
            System.out.print(", " + buffer[i]);
        }
        System.out.println();
   }
}
```

We are assuming that the buffer is sufficiently big that it won't get full.

No two threads may be allowed to call add at once;

Also no thread can be allowed to call print whilst something else is adding.

We assume print is not called on an empty buffer

Two threads are not allowed to join the same time.

Exam

Answering questions about thread interaction.

- Say that addition / subtraction (or relevant action) is not atomic therefore interleaving of statements can lead to a loss of information.

Solutions to Critical Section Problem

Critical section problem says that there can only be one thread accessing a resource at a once.

Synchronising Critical Sections

Global Variables	
p	q

`local variables`
`loop_forever`
`  non-critical section`
`  preprotocol`
`  critical section`
`  postprotocol`
	`local variables`
	`loop_forever`
	`  non-critical section`
	`  preprotocol`
	`  critical section`
	`  postprotocol`

It only loops forever if it says it loops forever.

- `preprotocol`: check that it is okay to enter critical section;
- `postprotocol`: signal that critical section is complete.
- Together called a ***synchronisation mechanism***.

Properties of a Solution

- **Mutual Exclusion:** only one process can enter the critical section at once
- **Freedom from Deadlock**: if some processes are waiting to enter their critical section eventually one must succeed (if all are waiting then we have a deadlock).
- **Freedom from Starvation (Success in the Absence of Contention):** If a process is waiting to enter its critical section it must do so eventually (regardless of whether others have finished executing).
    - Note that this does not guarantee 'fairness' if one process gets 10000 entries into its critical section before another enters, the other process is not starved since it ***eventually*** runs
- **Starvation vs Deadlock:** In deadlock processes must be waiting for a variable assignment that will never happen (there is no chance of escape); in starvation it can be that an assignment might not happen due to interleaving but could happen on a different one.

For a solution to be a solution, there must be no interleaving that breaks these properties.

Standard testing doesn't work because each time we only see one interleaving and we need to show correctness for all, testing can find some bugs, but it's not guaranteed to find all bugs (unless all possible interleavings are exhaustively tested). We therefore need to prove correctness of programs considering all possible cases.

Explaining these properties

integer turn ← 1	
p	q
`loop_forever`	`loop_forever`
`p1: non-critical section`	`q1: non-critical section`
`p2: await turn = 1`	`q2: await turn = 2`
`p3: critical section`	`q3: critical section`
`p4: turn ← 2`	`q4: turn ← 1`

Mutual Exclusion - We are never in the state (p3,q3,_,_) - therefore mutual exclusion holds.
Deadlock - in order to deadlock every process must have stopped execution, for that to occur turn = 1 & turn = 2 which is a contradiction.

Assumptions

- A process will **complete** its **critical section**
- A process may **terminate** during its **non-critical section**
- The CPU scheduler will not starve the process
- An atomic assignment operator ←

Execution Example (First Attempt)

integer turn ← 1	
p	q

`loop_forever`
`p1: non-critical section`
`p2: await turn = 1`
`p3: critical section`
`p4: turn ← 2`
	`loop_forever`
	`q1: non-critical section`
	`q2: await turn = 2`
	`q3: critical section`
	`q4: turn ← 1`

NB that **await** is the same as **while (inverse condition)** for q2 **await turn = 2** is the same as **while (turn != 2) {// we might want to sleep!};**
We can visualize processes with state diagrams.

State Diagrams for Processes

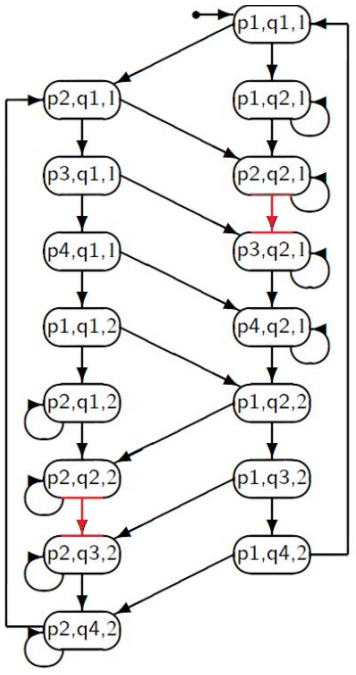
We can show inteleavings with state diagrams.

Shows all execution traces.

Never are we in p3,q3 therefore ***mutual exclusion*** holds!

Deadlock? Well, the only place we can gets stuck is at await: p2 → p3 or q2 → q3 (red arrows).
Deadlock will only happen in (p2,q2,_).

- But if we're in those states then either turn = 1 or turn = 2.
  - turn = 1 Eventually p will be scheduled and progress.
  - turn = 2: eventually q will be scheduled and progress.



Freedom from Starvation (Success in the Absence of Contention)

integer turn ← 1	
p	q
`loop_forever`	`loop_forever`
`p1: non-critical section`	`q1: non-critical section`
`p2: await turn = 1`	`q2: await turn = 2`
`p3: critical section`	`q3: critical section`
`p4: turn ← 2`	`q4: turn ← 1`

- Suppose we have:
  - (p1,q1,1), (p2,q1,1), (p3,q1,1), (p4,q1,1), (p1,q1,**2**) then q exits (or has an infinite loop) during its non-critical section (which is permitted).
  - p1 cannot continue to execute.
- First attempt does not have this property!

If q quits after p finishes, then p will never execute past p2, that is turn will always be 2 and p will be stuck (starvation will occur). This failed because both rely on a single global variable that says when to enter the critical sections for each Thread.

Mutual Exclusion: Execution Trace

boolean wantp ← false, wantq ← false	
p	q
`loop_forever`	`loop_forever`
`non-critical section`	`non-critical section`
`p1: await wantq = false`	`q1: await wantp = false`
`p2: wantp = true`	`q2: wantq = true`
`critical section`	`critical section`
`p3: wantp ← false`	`q3: wantq ← false`

Process p	Process q	wantp	wantq
`p1: await wantq = false`		false	false
	`q1: await wantp = false`	false	false
`p2: wantp ← true`		true	false
	`q2: wantq ← true`	true	true
`critical section`		true	true
	`critical section`	true	true

This is violation of mutual exclusion since we have found an execution trace that have entered a critical section at the same time. To prove that this is incorrect we have to find one

counter example (to show that it's correct, we have to prove that all of the inverleavings are correct).

Informal Deduction (i.e. Mutual Exclusion)

boolean wantp ← false, wantq ← false	
p	q
loop_forever p1: non-critical section p2: wantp ← true p3: await wantq = false p4: critical section p5: wantp ← false	loop_forever q1: non-critical section q2: wantq ← true q3: await wantp = false q4: critical section q5: wantq ← false

Proving without a state diagram.

- For mutual exclusion to hold we must show that (p4,q4,_,_) is not possible
- To get there we would need to go through (p3,q4,_,_) v (p4,q3,_,_)
- If (p4,q3,_,_) -> wantp == true so q cant go past q3
- Therefore mutual exclusion holds for this example since we **cannot enter the critical section in q, once we entered it in p.**

EXAM
In a deadlock question. If it asks for a condition to hold for deadlock to occur. Remember that our pre protocol must NOT hold to go to the next section, thus we have to negate our conditions.

Peterson's Algorithm (satisfies all required properties)

boolean wantp ← false, wantq ← false, integer last ← 1	
p	q
loop_forever p1: non-critical section p2: wantp ← true P3: last ← 1 p4: await wantq = false or last = 2 P5: critical section p6: wantp ← false	loop_forever q1: non-critical section q2: wantq ← true q3: last ← 2 q4: await wantp = false or last = 1 q5: critical section q6: wantq ← false

Mutual Exclusion:

- For mutual exclusion to hold we must show that (p5, q5, _,_,_) cannot be true.
- To get to that state one of the two must already be @5 (and stay there) the other must move from 4 to 5.
  - Processes are symmetric so it doesn't matter which one.
- So let's assume p is at p5: What is the value of *wantp*?
  - Only p can change *wantp*: at lines p2 and p6.
  - Since we're at p5 the last of these two to execute must be p2, therefore *wantp* = true.
- Also note that last = 1 or last = 2 (by inspection of the program: these are the only values ever assigned).

p4: await wantq = false or last = 2	q4: await wantp = false or last = 1

- We know: (p5,q4,true,_,1 or 2);
- We want to show that q cannot transition from q4 to q5 :
  - *wantp* = true (from previous slide) so q can only proceed if last = 1.
  - The only line of code that can set last = 1 is p3.
- When p executed p4 and transitioned to p5 either:
  - *wantq = false:*
    - That means q was at q1 or q2 (inspection of the program);
    - Therefore q must execute q3 setting last =2 before q4;
    - Since p remains at p5 it cannot execute last =1;
    - Therefore last = 2 and q stops at q4;
  - last = 2
    - If last = 2 at p4 and p remains in p5, p3 has not been executed to set last = 1
    - therefore last = 2 and q stops at q4.

Deadlock Checking

- They must be both at line 4
- Both wantp and wantq are true and not changed
- Would require last == 2 and last ==1
- Last can't have two values, therefore
- No deadlock

Starvation Checking

- If p4 it will eventually execute p5
- Unless wantq = true and last = 1
- Where could q be?
  - Q3
    - It will execute last = 2
    - It will then remain at q4 until p executes
  - Q4
    - It will continue to q5 since last = 1

        - Proceed to q6 (non-termination in cs)
        - Execute q6 (set wantq to false)
        - Execute q1 and either
            - terminate, p can proceed as wantq is false
            - Execute q2 setting wantq to true
            - Execute q3 setting last = 2
            - It will then remain at q4 until p executes
    - Q5
        - Follow from step 2 for q4

Temporal Logic

Box - means always the case
Diamond - at some point in the future

- Mutual Exclusion:
  – □ ¬(p5 ^ q5)

- Freedom from Deadlock:
  – □ ((p4 ^ q4) → ◊ (p5 v q5))

- Freedom from Starvation:
  – □ (p4 → ◊ p5)

- It is always the case that p5 and q5 cannot be at the same state
- (For Freedom from Deadlock) Always the case that if you are at p4 and q4 then at some point in the future you will be at p5 or q5
- (For Freedom from Starvation): it is always the case that if you are at p4 you will be at p5 in the future).

Model Checking

- Given a model (formula to verify) and a Program
- Show that all states satisfy the model
- Builds state diagram from the initial state and use search to show that no violating states exist
    - Much quicker for a computer to do state diagrams
    - State space for complex programs can be too big

Lecture 3 Advanced Solutions to the Critical Section Problem

Bakery Algorithm

- We take a ticket, hold to it and then you wait for your number to be called and then you get your service.

Example with two threads:

integer np ← 0, integer nq ← 0	
p	**q**

`loop_forever`
`p1: non-critical section`
`p2: np ← nq + 1`
`p3: await nq = 0 or np <= nq`
`p4: critical section`
`p5: np ← 0`
	`loop_forever`
	`p1: non-critical section`
	`p2: nq ← np + 1`
	`p3: await np = 0 or nq < np`
	`p4: critical section`
	`p5: nq ← 0`

Np and nq are ticket numbers.

- You select a ticket number that is one greater than the previous ticket number. Then you are going to wait for another thread to have a bigger ticket number.
- Zero indicates that a thread doesn't want to enter critical section.
- A positive number means the process is in the queue. The queue is sorted lowest number first.
- **Tie Break:** for equal ticket numbers we will arbitrarily favour np.

Dealing with N number of threads

```
integer array[1…N] number ← [0,…,0]
loop_forever
p1: non-critical section
p2: number[i] ← 1 + max(number)
P3: for all other processes j
p3:  await number[j] = 0 or number[i] << number[j]
p4: critical section
p5: number[i] ← 0
```

- Each thread has a unique id, **i**, in the range [0..N].
- **number[i] << number[j]** means either:
  - **number[i] < number[j]** or:
  - **number[i] = number[j] and i < j.**
- For equal ticket numbers we will arbitrarily favour lower id thread.

- Pick a ticket number that is larger than any currently held ticket (else the program could lead to starvation).
- Our process will enter the critical section either when it's the smallest non-zero process in the array (zero means the process doesn't want to enter critical section), else wait.
- After we are done with our critical section set ticket number to zero (stating that I am not waiting to enter critical section) if not we will starve the system since we will have the lowest non-zero ticket and have finished our critical section.

*Assumption: finding the max of an array is an atomic operation (**this is not an atomic operation**) to fix it we can do the following:*

```
integer array[1…N] number ← [0,…,0]
loop_forever
p1: non-critical section
P2: choosing[i] ← true
p3: number[i] ← 1 + max(number)
p4: choosing[i] ← false
p5: for all other processes j
p6:  await choosing[j] ← false
p7:  await number[j] = 0 or number[i] << number[j]
p8: critical section
p9: number[i] ← 0
```

We implement a second boolean array which makes sure that we have chosen a ticket for a certain ticket by giving it a true value and then setting it to false. Then, inside our loop we wait for the boolean to be false to continue, meaning that the other process has chosen a

ticket number. Before making a ticket wait or proceed we make sure that tickets have chosen a number. This algorithm forces all tickets to have chosen a ticket since the rest will wait for that to occur.

+ i only writes on i and the rest just look into it which means that we do not need concurrency for the algorithm.
+ Mutual Exclusion, Freedom from Deadlock and Freedom From Starvation hold (the latter can be broken if a ticket number chosen is not larger than all existing tickets)
- Unbounded (no upper bound) ticket numbers (you can overflow the maximum number you can store in an integer)
- Each process has to query all other processes to see if it can enter its critical section, even if no others want to enter
- **Bottom line: Too inefficient to use in practice**

NB *if a process quits everything is fine since to quit it must be in its non-critical section. Thus, it has set it's ticket equal to zero before (p9) or has done nothing at all.*

Exam*, in a quiz you've made the mistake.* *The bakery algorithm could lead to starvation if processes did not pick a ticket number larger than that of all existing tickets.False.*
The bakery algorithm could allow violation of mutual exclusion if processes did not pick a ticket number larger than that of all existing tickets. False
These are both true.

Synchronisation Hardware

Instead of using an atomic assignment operator we can use a **lock** that a thread must obtain in order to enter its critical section. We can use hardware to provide us with primitives to allow us to do synchronisation much more easily

Single vs Multi-Processor Solutions

In a single processor environment we could solve the CS problem by disabling interrupts whilst shared data is being modified

In a multiprocessor environment however

- Message has to be passed to each processor everytime a critical section is entered (slow)
- Disabling interrupts affects the system clock if the clock is kept up to date by interrupts

Test and Set

boolean lock ← false
p
loop_forever **p1: non-critical section** **p2: await(!testAndSet(lock))** **p3: critical section** **p4: lock = false;**

```
boolean testAndSet(Boolean lock){
   boolean toReturn =
                  lock.getValue();
   lock.setValue(true);
   return toReturn;
}
```

Test and set changes the value of a variable (e.g. lock) to true, and returns what its original value was. So if test and set returns false then we know lock was false (no one was in their critical section) but now it is true so we can go into our critical section, having set it to true to stop others doing the same. Lock must be set to false on exiting the critical section so that other processes are able to progress into their critical sections.

Think of it as me asking if I can go to the bathroom. Just in case I always think that its true that there is someone in the bathroom. Though the bathroom returns to me what was there before I set it to true. If this returns false then I can go in (and since I already set it to true no one will come). When I exit, I will make it false again.

Single processor instruction that will both test the value of a variable and set it to a new value.

- Stores the value of the lock, sets it to true and return the old value of the lock.
- Test and set will return the value of the lock (false means no one is in CS) and set it to true.
- If two processors attempt to execute this instruction simultaneously the hardware will ensure the instructions will be executed sequentially (in arbitrary order)
- Works for n processes (atomic assignment no longer required because the test and set is atomic so the value of lock cannot be updated part way through the execution of this statement.)

Swap

boolean lock ← false
p boolean key ← true

loop_forever
p1: non-critical section
P2: key = true
p3: while(key = true)
swap(lock,key);
p4: critical section
p5: lock = false;

```
void swap(Boolean lock,
              Boolean key){
   boolean tmp = lock.getValue();
   lock.setValue(key.getValue());
   key.setValue(tmp);
}
```

Swap exchanges the values of two boolean variables (e.g. lock and key). If we want to know that lock was false in order to enter the critical section then we can set key to true and call swap(lock,key). Key will remain true if lock is true, so we know that only when key becomes false we can proceed into our critical section. Further, since we know that key was true, and we just swapped its value with that of lock (which was false) then we know that lock is now true, so no other processes will enter their critical section.

Think of key as a consumable. We have it so it's true! Then we use it, if the lock is being used, we cannot open it (true), then I will get true which means that my true key cannot be used on a lock that's being used. When I get false, then that means I have consumed my key and I have the lock, hence setting the lock to true! Because I'm using it!

- Lock is a global variable shared by all processes whilst p is a local variable stored by each process.
- Single processor instruction that will swap the value of two booleans.
- Hardware enforces the instruction is non simultaneously executed.
- Works for n assignments (no atomic assignment required since the value of the lock is only ever set to false, hence any interleavings will always set it to false)
- Starvation is possible but unlikely.

Semaphores

Busy-Wait

Software mechanisms that underlyingly makes use of hardware instructions. These are less prone to error as more abstract and easier to work with.
Semaphore: example of a commonly used synchronisation mechanism provided by most operating systems.

- Integer variable, v - the value of the semaphore.
- (Optionally) a set of processes that is blocked (initially empty)
- Two methods:

```
Naïve Version (busy wait):
Wait(S){
  while(S.v <= 0){
    //do nothing
   }
   S.v = S.v - 1;
}
```

```
Naïve Version (busy wait):
Signal(S){
  S.v = S.v + 1;
}
```

`while(S.v <= 0){}` Also called a *spin lock.*

- While the value of the semaphore is equal to zero, it does nothing.
- The wait method checks if there is any other process in the critical section (in this case S.v will be 0 or smaller). If so, it will block (not allow a process to proceed into the critical section).
- Note that the wait method must be called with atomic subtraction
- The signal method will be called in the critical section of a process and will add one to the value of the semaphore indicating that it's terminating the critical section and some other process can enter.

This is not very efficient in terms of CPU Utilisation because to check the value of a variable we must use the CPU. So we are using CPU cycles to constantly check the value of a variable while the loop is running **(1)**, that means that we cannot use those CPU values for the thread that is actually executing code and will then signal the semaphore **(1)**.
QUESTION: Can this semaphore be applied to more than two threads?

Blocked-Set Semaphore

Has

- Integer variable, v
- A set of blocked processes (initially empty)

```
Wait(S){
 S.v = S.v - 1;
 if(S.v < 0) {
    S.blocked = S.blocked U this;
    this.state = blocked;
   }
}
```

```
Signal(S){
  S.v = S.v + 1;
  select one q from S.blocked:
  q.state = runnable;
}
```

CS Problem for 2 Threads with Semaphores

Semaphore S ← (1, Ø)	
p	q
loop_forever p0: non-critical section p1: wait(S) p2: critical section p3: signal(S)	loop_forever q0: non-critical section q1: wait(S) q2: critical section q3: signal(S)

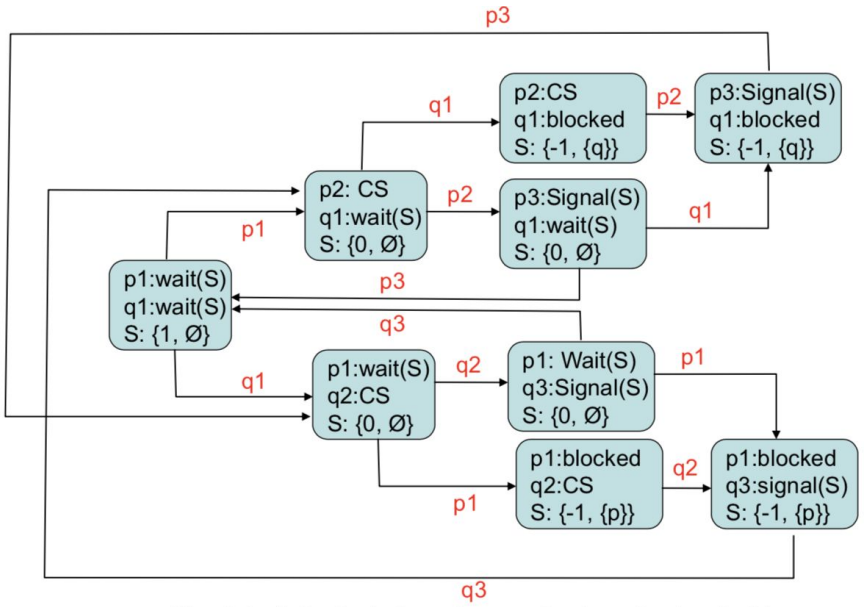
- Initially S.v = 1
- Before a threads enters CS
  - S.v is going to be decremented (zero means some thread in CS)
  - So if the other process calls wait it will be blocked
- When it leaves CS
  - Signal adds 1 so that S.v = 1 and other processes can enter CS
  - If the other process is waiting it is unblocked

Exam

When answering exam questions that say “describe” we can simply put the code, when it says explain we have to give some explanation.

- If information about semaphores is asked for we can provide the methods and explain them.
- If it’s asked about how it solves the critical section problem we can write and describe what is said under *“CS Problem for 2 Threads with Semaphores”.*

State Diagram



No state (p2,q2,_), therefore **mutual exclusion** holds.
No state in which both are blocked therefore **no deadlock**.

Starvation

When a process is awoken it leaves the blocked state and goes into the critical section, freedom from starvation depends on signal. If the thread wasn't scheduled before the other thread, then the other thread could be scheduled to run and compile wait(S) first each time.

Properties of Busy-Wait Semaphore

Semaphore S ← (1, Ø)	
p	q

`loop_forever`
`p1: non-critical section`
`p2: wait(S)`
`p3: critical section`
`p4: signal(S)`
	`loop_forever`
	`q1: non-critical section`
	`q2: wait(S)`
	`q3: critical section`
	`q4: signal(S)`

= number of

- S.v > 0 always
- S.v = init + #signal(S) - #wait(S)
- #CS + S.v. = 1
- #CS = #wait(S) - #signal(S)

Thread p	Thread q	Thread r	S
p1: wait(S)	q1: wait(S)	r1: wait(S)	(0, Ø)
p2: critical-section	**q1: wait(S)**	r1: wait(S)	(-1,{q})
p2: critical-section	q1: blocked	**r1: wait(S)**	(-2,{q,r})
p2: critical-section	q1: blocked	r1: blocked	(-2,{q,r})
p3: signal(S)	q1: blocked	r1: blocked	(-2,{q,r})
p1: wait(S)	q1: blocked	**r2: critical-section**	(-1,{q})
p1: wait(s)	q1: blocked	r3: signal(S)	(-2,{p,q})
p1: blocked	q1: blocked	**r3: signal(S)**	(-2,{p,q})
p2: critical-section	q1: blocked	r1: wait(S)	(-1,{q})
p3: signal(S)	q1: blocked	**r1: wait(S)**	(-2,{q,r})
p3: signal(S)	q1: blocked	r1: blocked	(-2,{q,r})

If we were to continue this interleaving, it would be the case that Q is starved since it's left blocked.

- Starvation can occur but it requires a specific interleaving so it's improbable but possible.
    - We can fix this by using a queue (FIFO) in the semaphore instead of a set (blocked-queue semaphore)

We can use semaphores to make threads wait for each other (similar to join) by initialising the semaphore value to 0. This means nobody that's waiting can start before something is signaled.

- E.g. merge-sort: split array in two and use one thread to sort one side and the other to sort the other side and a third thread does the merging after completion (this has to occur after).

The Producer Consumer Problem

- **Producer:** a process that creates data
- **Consumer:** a process that takes the data and uses it.
- For **asynchronous** communication we need a buffer
- Two issues can occur:
    - Buffer is empty, consumer cannot take data
    - Buffer is full, producer cannot add data

Semaphore notEmpty ← (0, Ø), infinite queue<d> buffer ← empty	
producer	**consumer**
loop_forever **p1: d ← produce** **p2: append(d,buffer)** **p3: signal(notEmpty)**	**loop_forever** **q1: wait(notEmpty)** **q2: d ← take(buffer)** **q3: consume(d)**

For the purpose of example we assume that we have an infinite Buffer.

- Only need to synchronise removal
- Semaphore initialised to 0 not 1 because we don't want a consumer to be able to consume until something has been added to the buffer
- The semaphore value is 0 if the buffer is empty and we are waiting for something to be signaled aka be produced

Semaphore notEmpty ← (0, Ø), Semaphore notFull ← (N, Ø), size N queue<d> buffer ← empty	
producer	**consumer**

`loop_forever`
`p1: d ← produce`
`P2: wait(notFull)`
`p3: append(d,buffer)`
`p4: signal(notEmpty)`
	`loop_forever`
	`q1: wait(notEmpty)`
	`q2: d ← take(buffer)`
	`q3: signal(notFull)`
	`q4: consume(d)`

Finite Buffer producer/consumer

- Producer makes non-empty buffers for the consumer, to make sure that there is always something to consume
- Consumer makes non-full buffer for the producer
- notFull semaphore is initialised to N, so it can be decremented N times before it reaches 0
- If you wait for notFull then you go in the queue for notFull. If you wait on notEmpty then you go in the queue for notEmpty.
- This technique is called **split semaphores** (we also use it to wait for other threads)

Convention when you name a semaphore is that you name it after the thing you are going to wait for.

Initialising Semaphores to Values other than 1 or 0

Semaphore S ← (N, Ø)
p

`loop_forever`
`p1: non-critical section`
`p2: wait(S)`
`p3: critical section`
`p4: signal(S)`

N threads can proceed through wait, before one is blocked to call signal e.g. N = 3, 3 threads can enter at once

This does not enforce mutual exclusion but it does allow a limit on the number of threads in CS (e.g. only 3 processes can open files at one time in the system)

Types of Semaphore

- **Busy Wait Semaphore:** the first type we saw where a process continually loops checking s.

- **Blocked-Set Semaphore (weak semaphore):** the type of semaphore we have been using, stores blocked processes in a set and wakes one arbitrarily;
- **Blocked-Queue Semaphore (strong semaphore):** uses a FIFO (FCFS) queue instead of a set. It also guarantees **no starvation**.

The Dining Philosophers Problem

5 philosophers are sitting and they take turns eating and thinking, each philosopher needs 2 forks (chopsticks or instruments) to eat, but there are only 5 on the table.

- Think is the non-critical section
- No 2 philosophers can hold the same fork simultaneously
- Each philosopher can pick up the fork on his right or left but only one at a time (mutual exclusion)
- Freedom from starvation
- Freedom from deadlock

Semaphore array fork ← [1,1,1,1,1]
Philosopher 4

loop_forever
p1: think
p2: wait(fork[0]) //right fork
p3: wait(fork[4]) //left fork
p4: eat
p5: signal(fork[0])
P6: signal(fork[4])

Semaphores in Java

- API Documentation:
  - http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html
  - import java.util.concurrent.Semaphore;
- Constructor:
  - Semaphore(int permits) Creates a Semaphore with the given number of permits and nonfair fairness setting.
  - Semaphore(int permits, boolean fair) Creates a Semaphore with the given number of permits and the given fairness setting.
- Important Methods:
  - void acquire() Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted. Throws InterruptedException
  - void release() Releases a permit, returning it to the semaphore.
  - N.B. acquire() = wait, release() = signal.
  - boolean isFair() Returns true if this semaphore has fairness set true.

- NB the fair setting indicates if the semaphore is a Blocked-Queue or a Blocked-Set. By default it's not fair (blocked-set).

Producer/Consumer in Java

We make a buffer:

```
public class Buffer {

  LinkedList<Integer> buffer;
  Semaphore access;

  public Buffer(){
    buffer = new LinkedList();
    access = new Semaphore(1);
  }

  public Integer removeItem(){
        …
  }

  public void addItem(Integer i){
        …
 }
}
```

- Accesses to the buffer (add/remove) are the critical sections.
- We need a semaphore to make sure that access to the buffer is atomic.
- Initialising the semaphore to 1 makes sure only one thread can modify the buffer at once.

```
public Integer removeItem(){
   try{
     access.acquire();
   } catch (Exception e){
      e.printStackTrace();
   }
   Integer toReturn =
   buffer.removeFirst();
   access.release();
   System.out.println("Buffer size
   " + buffer.size());
   return toReturn;
}
```

```
public void addItem(Integer i){
  try{
    access.acquire();
  } catch (Exception e){
    e.printStackTrace();
  }
  buffer.addLast(i);
  access.release();

  System.out.println("Buffer size
   " + buffer.size());
}
```

removeItem and addItem have its CS protected by a semaphore with access.acquire(), the thread can be interrupted while waiting, hence the catch expression.

Producer:

```
public class Producer extends Thread {
  Semaphore notFull;
  Semaphore notEmpty;
  Buffer buffer;

  public Producer(Semaphore isNotFull, Semaphore isNotEmpty, Buffer
   toUse){
        notFull = isNotFull; notEmpty = isNotEmpty; buffer = toUse;
    }

    public void run(){
      for(int i = 0; i < 10; ++i){
        try{
            notFull.acquire();
        } catch (Exception e){
            e.printStackTrace();
        }
        buffer.addItem(i);
        notEmpty.release();
          //waste some time
      }
}}}
```

Notice that the semaphore that is released (signalled) is not the same semaphore that is acquired (waited for).

Consumer:

```
public class Consumer extends Thread {
  Semaphore notFull;
  Semaphore notEmpty;
  Buffer buffer;

  public Consumer(Semaphore isNotFull, Semaphore isNotEmpty,
                              Buffer toUse){
    notFull = isNotFull; notEmpty = isNotEmpty; buffer = toUse;
  }

  public void run(){
    for(int i = 0; i < 10; ++i){
      try{
        notEmpty.acquire();
      } catch (Exception e){
          e.printStackTrace();
      }
      Integer item = buffer.removeItem();
      notFull.release();
      //waste some time (a bit more than producer)
      System.out.println("Got " + item);
        }}}
```

Notice that the semaphores are now used in the opposite order.

Main Method:

```
public class ProducerConsumerSemaphore {

    public static void main(String[] args) {
          Semaphore notEmpty = new Semaphore(0);
        //determines the size of the buffer
        Semaphore notFull = new Semaphore(5);

        Buffer buffer = new Buffer();
        Producer p = new Producer(notFull,notEmpty,buffer);
        Consumer c = new Consumer(notFull,notEmpty,buffer);

        c.start();
        p.start();
  }
}
```

- Notice notEmpty = 0, zero threads are allowed to consume
- NotFull = 5, 5 threads can produce before any need to be blocked

Semaphore Debugging

- Don’t forget to release semaphores that you acquire
- Avoid deadlock:
    - One thread doing acquire A, acquire B and another doing acquire B, acquire A is likely to lead to this.
- Only keep the semaphore as long as you need it.
- Beware: you can call release() if you didn’t call acquire() and that can mean more things than intended enter critical sections.

NB, when implementing code to do with semaphores we have to make sure that every party is using the SAME instance of the semaphore.

4 Monitors and Concurrency in Java

Issues with Semaphores

Semaphores are still prone to errors, they must correctly implement wait(S) before they access the shared resource { else end up in mutual exclusion }, critical section and signal(S) {we can end up with deadlock}. Someone can switch them around.

Using Monitors to solve this issue

Invented by Hoare and Hansen
Your shared resource is represented by a monitor which has methods through which other threads/processes can access the resource (this is the only way)

- Think it's like a private instance variable with public accessor methods

We then stipulate that none of these methods can execute concurrently thus only one method can modify or use the resource at once.

+ Responsibility for mutual exclusion relies on the creator of the resource (more difficult for individuals accessing the resource to create sync bugs).
+ The management of resources is distributed. Each resource can be handled by a separate monitor.
    - If processes are requiring to run methods on the same monitor, mutual exclusion can be enforced.
    - If processes are requesting to run processes of two separate monitors can be executed concurrently.
+ Fits with OO design

Example:

```
monitor Integer
  Integer n = 0;
  method increment()
    n = n + 1;
  method decrement()
    n = n - 1;
```

p	q
p1: Integer.increment()	q1: Integer.increment()
p2: Integer.decrement()	q2: Integer.decrement()

- All methods in the monitor are executed in mutual exclusion;
  - A process must acquire the *lock* for the monitor to execute a method.
- increment (decrement) cannot be executed by both processes at the same time;
- P cannot execute decrement whilst q executes increment and vice versa.

Implementing a Monitor using semaphores

```
class Monitor
  semaphore s = 1;
  //any required variables
  method method1()
    s.wait()
    //actual method implementation
    s.signal()
  method method 2()
    s.wait()
    //actual method implementation
    s.signal()

etc.
```

Implementing Semaphores using Monitors

```
monitor Semaphore
  Integer s = k

  method semaphoreWait()
    while(s = 0)
      wait()
    s = s - 1

  method semaphoreSignal()
    s = s + 1
    notifyAll()
```

P	q
loop forever: p0: non-critical section p1: Semaphore.semaphoreWait() p2: critical-section p3: Semaphore.semaphoreSignal()	loop forever: q0: non-critical section q1: Semaphore.semaphoreWait() q2: critical-section q3: Semaphore.semaphoreSignal()

Monitor Wait() and NotifyAll()

A monitor has a set of *blocked threads that are waiting*. You must hold the **monitor lock** to execute any code.

wait()

- Adds the current thread to the set blocked
- Sets its status to blocked
- Releases the monitor lock.

notifyAll()

- Removes all process from the set blocked (if blocked is non empty);
- Sets their states to ready (whichever thread the scheduler selects to run first will get the monitor lock, the other processes will be blocked on entry).

- Avoids deadlock because it awakes multiple threads, if one of them calls wait then another will execute, more context switching. (**Recommended)**

notify()

- Selects and removes a process from the set blocked (if blocked is not empty)
- Sets the state of that thread to ready (it's next to execute)
- Note that it wakes up only one arbitrary thread, thus deadlock can occur. (E.g. if a producer awakes another producer for example, it will call wait at some point thus deadlocking the system since no one will consume the product.)
- You should only use notify when you are absolutely sure that waking up any one thread will make your program work. Using notifyAll will create more overhead but allow to mitigate future issues (people extending your code).

Threads can only call wait/notifyAll/notify if they hold the monitor lock. This is because when the methods are called, they have to release the monitor lock, to do so they must have it.

```
wait(){
 blocked = blocked U this;
 this.status = blocked
 lock.release()
}
```

```
notifyAll(){
  while(blocked not empty){
    p = blocked.remove();
    p.status = ready;
  }}
```

```
notify(){
  if(blocked not empty){
    p = blocked.remove();
    p.status = ready;
  }}
```

notifyAll(condition) vs notify(condition)

- Not permitted in Java
- Only processes that are actually waiting for the condition that is true get awaken

```
monitor Buffer
  int[5] Buffer
  int spaceUsed;
  condition notEmpty
  condition notFull

method addItem(int i)          method removeItem()
    while(spaceUsed = 5)           while(spaceUsed == 0)
     wait(notFull)                   wait(notEmpty)
    buffer[spaceUsed] = i          i = buffer[spaceUsed]
    ++spaceUsed                    --spaceUsed
    notifyAll(notEmpty)            notifyAll(notFull)
                                   return i
```

p	q
loop forever: p1: i <- produce p2: Buffer.addItem(i)	loop forever: q1: i <-Buffer.removeItem() q2: consume(i)

Now maintain a separate queue for each condition.

```
notify(cond){
  if(cond not empty){
    p = cond.remove();
    p.status = ready;
  }}
```

```
notifyAll(cond){
  while(cond not empty){
    p = cond.remove();
    p.status = ready;
  }}
```

Exam

Why calling notify() rather than notifyAll() in a monitor can lead to deadlock.

- It's possible that the single thread that is woken up may not be able to run
- Because it immediately needs to wait for another condition
    - It's possible that the thread is a producer calling notify and another producer is woken up even though the buffer is full.

Since the woken up thread cannot run, it will return to the blocked state, but since all other threads are also in the blocked state, thus nothing can run. If notifyAll is called, then all the threads have had the blocked state removed thus something will be able to run.

Can we use a Queue instead of a Set?

- It's only worth storing a queue if we are going to call notify() and only awake the head of the queue, if we wake up every thread then the scheduler will choose what to run.

```
monitor Example
  //some variables
  condition cond1 cond2
method method1()                  method method1()
    while(!cond1)                     while(!cond2)
      wait()                            wait()
   //optionally do something      //optionally do something
    while(!cond2)                     while(!cond1)
      wait()                            wait()
    //something using both            //something using both
    notify(cond1)                     notify(cond2)
    notify(cond2)                     notify(cond1)
```

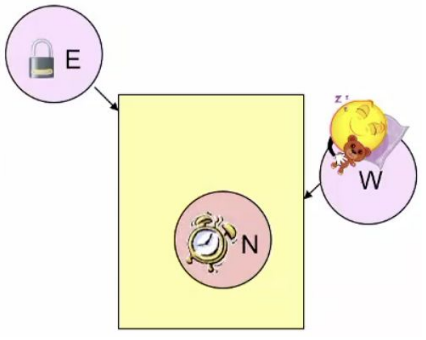
Monitors can deadlock. It's difficult since everything is in the same class. Deadlock can occur if both processes are waiting on the same condition which will never occur.

E.g. If method1 executes

Notify before Returning

We will normally place notifyAll() at the end of our method to ensure that everything has executed before we wake up other threads. Nevertheless, imagine that you have to return something in Java. This means that we will have to put a return statement after that. Potentially context switching could occur and disrupt normal functionality of our program. We look at priorities in to fix this.

Threads States w.r.t Monitors



Threads:

- Waiting to enter the monitor (E) - *it's waiting to start executing a method in the monitor (not waiting for a condition) but it has not called wait because something else was executing before.*
- Waiting to be notified (W) - *it's called wait and it's in the blocked set*
- Executing notify (N) - *thread that is executing the monitor and has just called notify. It holds the lock.*
- In Java priority is given to thread N with threads E and W having equal priority:
  - E = W < N
- In the original definition of monitors waiting processes have priority:
  - E < N < W

Implications of Relative Priority

Classical case (Entering < Notifying < Waiting)

- **Immediate resumption requirement:** if a single process is notified it doesn't have to recheck the condition since it has just been signaled and nothing has been allowed to run.

Java Case (Entering = Waiting < Notifying)

Since the notifying threads have a larger priority than the waiting and entering ones, we will assure that our methods finishes executing before context switching occurs.

- In java there is no guarantee that the signaling process hasn't changed the condition in the code it continued to execute since E and W threads are equivalent (an E thread can start to execute and change the condition). Therefore we will need to recheck the condition before it can continue executing. To do so, in our code we will **use While(!(condition)) instead of if(!(condition))** since a while loop won't exit if the condition is not met.
  - We must check all conditions at once.

Readers and Writers Problem

There are two types of thread: reader and writer.

- Many readers can access the database at once.
- Only one writer can access the database at once.
- No reader can access the database at the same time as a writer.

```
monitor Buffer
  int readerCount = 0
  boolean writing = false

method startRead()
    while(writing)
     wait()
    ++readerCount;

method endRead()
    --readerCount;
    notifyAll()

method startWrite()
  while(writing ||
          readerCount != 0)
    wait()
  writing = true;

method endWrite()
    writing = false;
    notifyAll()
```

P	q
loop forever: p1: startRead() p2: read from database p3: endRead()	loop forever: q1: startWrite() q2: write to database q3: endWrite()

Breaking up Conditions

```
method startWrite()
  while(writing ||
         readerCount != 0)
    wait()
  writing = true;
```

```
method startWrite()
  while(writing)
    wait()
  while(readerCount != 0)
    wait()
  writing = true;
```

We must check all our conditions at once and using a while loop since they have to all be re-evaluated after wait() has been executed, we cannot break them up.

- On the left both conditions are checked together so we know both hold when we set writing = true
- On the right we could have a violation of our required properties:
    - Wait for writing to be false
    - Continue to wait for reader count = 0
    - Possible that some other writer also waiting for reader count = 0, is scheduled before us when reader count becomes 0
    - It starts writing but has not yet finished
    - This writer is awoken checks reader count is still 0 and continued to write
    - Two writers are now writing simultaneously

Writer and Reader Starvation

If a new reader arrives **before any other reader finishes**, the readers can hog the system. The writer could be left to starve by the reader.

```
monitor Buffer
  int readerCount = 0
  boolean writing
method startRead()          method startWrite()
    while(writing)            while(writing ||
     wait()                           readerCount != 0)
    ++readerCount;              wait()
method endRead()              writing = true;
    --readerCount;            method endWrite()
    notifyAll()                   writing = false;
                                  notifyAll()
```

Solution:

```
monitor Buffer
  int readerCount, waitingReaders = 0
  boolean writing, waitingToWrite = false
method startRead()          method startWrite()
    while(writing ||          while(writing ||
        waitingToWrite)                 readerCount != 0)
     wait()                     waitingToWrite = true
    ++readerCount;              wait()
method endRead()              writing = true;
    --readerCount;            waitingToWrite = false
    notifyAll()               method endWrite()
                                  writing = false;
                                  notifyAll()
```

Now readers can starve if at all times **writers are coming in before another one finishes**.

Exam

NB for starvation to occur we **must** state that this occurs before another writer finishes (if a writer finishes it calls notifyAll() hence a reader can run) - For full marks.

The solution is to make a variable for turns:

```
monitor Buffer
 int readerCount, waitingReaders = 0
  boolean writing, waitingToWrite, readerTurn = false

method startRead()
    while(writing || (waitingToWrite && !readerTurn))
        ++waitingReaders
        wait()
        --waitingReaders
    ++readerCount;
    readerTurn = false

method startWrite()
  while(writing || readerCount != 0 ||
           (waitingReaders > 0 && readerTurn))
      waitingToWrite = true
      wait()
  writing = true
  waitingToWrite = false
  readerTurn = true;
//end read and end write as before
```

Monitor Solution to the Dining Philosophers problem

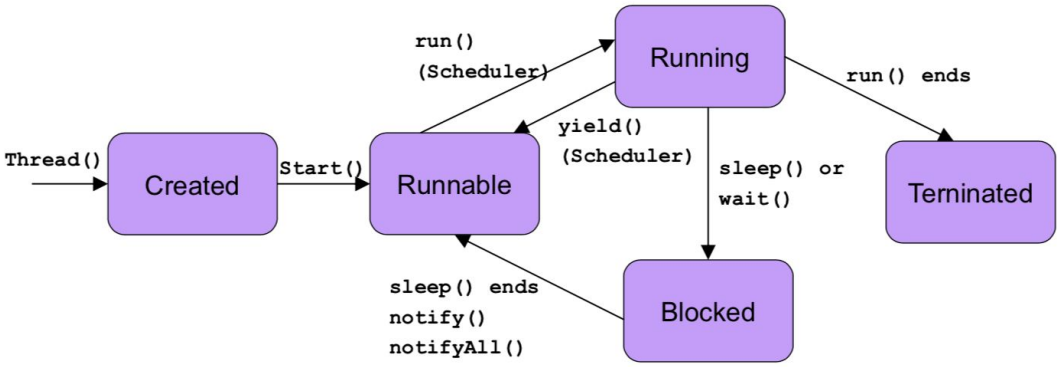
Monitors in Java

- Java's primary means of providing solutions to mutual exclusions is via monitors
- They are not explicitly represented, rather every class can be thought of as a monitor
- Unlike traditional monitors not all methods are required to be mutually exclusive
- Instead, methods that need to be mutually exclusive are marked with the keyword **synchronized.** (Making all methods synchronized would not allow two methods to run at the same time).
- We cannot make everything synchronized because that defies concurrency, hence no threads are running at the same time.

Object: Java API

protected Object **clone**()	Creates and returns a copy of this object.
boolean **equals**(Object obj)	Indicates whether some other object is "equal to" this one.
protected void **finalize**()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?> **getClass**()	Returns the runtime class of this Object.
int **hashCode**()	Returns a hash code value for the object.
void **notify**()	**Wakes up a single thread that is waiting on this object's monitor.**
void **notifyAll**()	**Wakes up all threads that are waiting on this object's monitor.**
String **toString**()	Returns a string representation of the object.
void **wait**()	**Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.**
void **wait**(long timeout)	**Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.**
wait(long timeout, int nanos)	**Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.**

Java Thread Lifecycle



Single-Track Bridge Example

- A bridge has room for only one lane of traffic
- Traffic must not enter the bridge going in opposite directions

```
public class Bridge {
  int carCount = 0;
  boolean westbound = false;

  //This is the code for the first case where one car at a time uses the bridge
  /*public synchronized void cross(Vehicle v){
    System.out.println(v + "entering bridge");
  try{
     Thread.sleep(100);
   } catch(InterruptedException e){
      e.printStackTrace();
   }
   System.out.println(v +  " leaving bridge");
  }*/

  public synchronized void enterBridge(Vehicle v){
      while((v.getWestbound() != westbound && carCount > 0) ||
                                (!v.getWestbound() && carCount > 2) ||
                                   (v.getWestbound() && carCount > 0)) {
         try{
            wait();
          } catch (InterruptedException e){ e.printStackTrace();}
        }
        westbound = v.getWestbound(); //set direction to your direction
        ++carCount; //increment cars using bridge
        System.out.println(v + " entering the bridge");
  }

  public void crossBridge(){
    try{
      Thread.sleep(100);
    } catch (InterruptedException e){
       e.printStackTrace();
    }
  }

  public synchronized void exitBridge(Vehicle v){
    --carCount;
    System.out.println(v + " exiting the bridge");
    notifyAll();
  }
}

-------

public class Vehicle extends Thread{
  Bridge bridge;
  String name;
  boolean westbound;

  public Vehicle(Bridge toCross,boolean west, String n){
    bridge = toCross;
    name = n;
```

```
    westbound = west;
  }

  public void run(){
    //code for first example
    //bridge.cross(this);
      bridge.enterBridge(this);
      bridge.crossBridge();
      bridge.exitBridge(this);
  }

  public String toString(){
    String dir = "west";
    if(!westbound) dir = "east";
    return name + " going " + dir;
  }

  public boolean getWestbound(){
      return westbound;
  }

}

---

public class Main {
    public static void main (String[] args){
        Bridge b = new Bridge(); boolean dir = false;
        for(int i = 0; i < 5; ++i){
            Thread t = new Vehicle(b, dir, "car" + i);
            t.start();
            dir = !dir;
            try{
                Thread.sleep(70);
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }

}
```

This example has the notion that as many cars as we want can enter the bridge so long as they are in the same direction and 1 car can go westbound and 3 cars can go eastbound at a given time but they cannot cross simultaneously in both directions.

To point out:

```
while((v.getWestbound() != westbound && carCount > 0) ||
                            (!v.getWestbound() && carCount > 2) ||
                               (v.getWestbound() && carCount > 0)) {
```

Note that this is the condition for waiting. We can say: I have to wait when...

`(v.getWestbound() != westbound && carCount > 0)`
states that no car can go in the opposite direction if we are going westbound and if there is more than a car (>0). If the last car was going westbound but there are no more cars, then we don't have to wait.
`(!v.getWestbound() && carCount > 2) || (v.getWestbound() && carCount > 0)`
These refer to the upper limits that the cars can go. If cars are going westbound and there are more than 2 we must wait. (Note that we write one less than the boundary, this would be the same as saying that 3 equal or less cars can go at once)

Inbuilt Concurrency in Swing

- When working with swing Java automatically uses thread to allow parallelism to ensure that GUIs are responsive. It makes your java code (main thread and any others that you created) run in a different thread.
- This means you can get concurrency bugs without even having explicitly created a thread
- Swing has a special thread: the **event dispatch thread**
  - All event-handling code is executed in this thread
  - Most code that interacts with the Swing framework must also execute on this thread
- Can also implement Swing worker threads to perform time-consuming tasks like loading background images.

```
@Override
  public synchronized void paintComponent(Graphics g){
    //etc.
  }
```

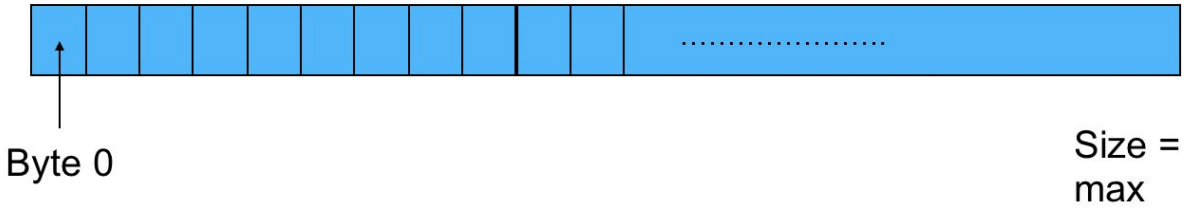
Paint will still run in a separate thread but now must not execute in parallel with removeBall.

```
  public synchronized void removeBall(Ball toRemove){
    balls.remove(toRemove);
  }
Make animate call removeBall(fiveBounces) instead of removing the ball.
```

Don't just synchronize the whole of animate because then paint can't run even when it is sleeping.

5 Memory Management

At a very low level memory is the RAM in your machine.
Memory in our machine is like an array, but bigger. The memory needs of most programs change whilst they are running thus dynamic memory allocation is really important.



Anything a program needs **exactly one copy of** can go in memory at a fixed address

- Java static variables are memory in a fixed address since its shared amongst the program.
- In a user interface a button with the word OK, only has to be stored once.
- Read-only variables are fixed: known as **text**.
- Anything static but not read-only: **data**

Method variables

We can have variables named the same in two different methods, hence we need to think how we separate these. This may also occur if a variable falls out of scope, then we lose access to it. **Variables fall out of scope in the order they were declared (LIFO).**

Scope is a side effect of how you computer manages memory.

The Stack

The stack manages how variables fall in and out of scope but it also stores the next line of code that has to execute when the current called method finishes. It works in the following way:

- A method is called, the next line of code in the current method is pushed onto the stack (so we know where to resume execution).
- Whatever occurs in the method is pushed onto the stack (variable declarations, calling other methods)
- When the method terminates, variables fall out of scope and we look onto the indicator of the next line and resume computation there.

'**Call**' in x86 assembler

- Push the address of the next line of code to the front of the stack
- Jump to the code for the named function

```
4e <main>:
...
5c: sub    $0x4,%esp
5f: movl   $0x64,0x0(%esp)
66: call   1b <steve>
6b: ...
78: call   f0 <printf@plt>
7d: ...
80: call   1b <steve>
85: ...
92: call   f0 <printf@plt>
97: add    $0x4,%esp
```

‘Ret’ in x86 assembler

- Pop a memory address of the front of the stack
- Jump to the code at that address (goes back to where it has to execute next)

```
1b <steve>:
1b: sub    $0x8,%esp
1e: movl   $0x3,0x4(%esp)
26: movl   $0x7,0x0(%esp)
...
4a: add    $0x8,%esp
4d: ret
```

Stack Pointer (SP)

The stack pointer signalises where the stack can be modified. To do so, we will have to subtract from it to create space for a new variable.
Sp - 4 (because there are 4 bytes in an int) will allow us to create space to store an int. Now our stack pointer will be at Sp before - 4 and the memory address of our int will be Sp now + 4.

To give the memory back, we just add 4 bytes ot the Stack Pointer then we know that this memory is “free to use”

```
4e <main>:
...
5c: sub    $0x4,%esp
5f: movl   $0x64,0x0(%esp)
66: call   1b <steve>
6b: ...
78: call   f0 <printf@plt>
7d: ...
80: call   1b <steve>
85: ...
92: call   f0 <printf@plt>
97: add    $0x4,%esp
```

Creates a variable (4 bytes)

0x64 (hexadecimal of 100) in sp
Calls method steve

Prints out to the console

Calls method steve

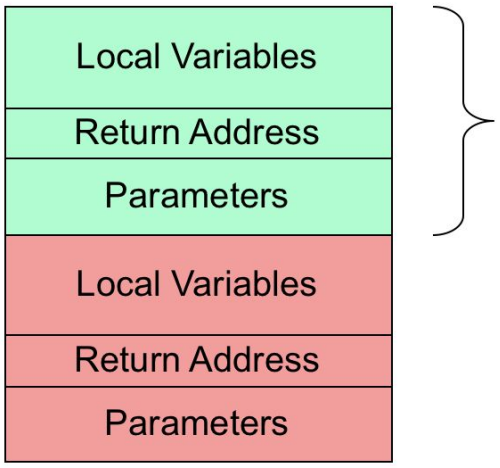
Returns the memory to the SP

```
1b <steve>:
1b: sub    $0x8,%esp
1e: movl   $0x3,0x4(%esp)
26: movl   $0x7,0x0(%esp)
...
4a: add    $0x8,%esp
4d: ret
```

Inside Steve
Creates two variables
0x3 in sp+4
0x7 in sp

Returns memory
Returns to the main method

Stack Frames



One **stack frame**

Everything a method needs to run.

Exact details are a **calling convention)**

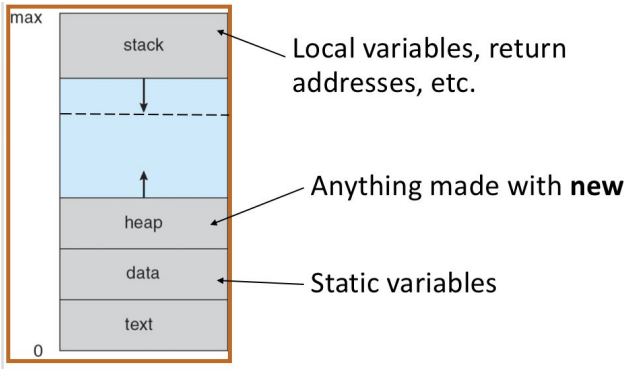
Keyword new in Java

```
public static ArrayList<Integer> li;

p.s. void makeList() {
  ArrayList<Integer> here
        = new ArrayList<>(10);

  li = here;
}

p.s. void main(...) {
  makeList();
  li.add(42);
}
```



Certainly in this example, when ***li*** falls out of scope (that is the method terminates) then we lose access to it. The use of the keyword ***new*** if you want the object to still exist after the end of the method.

new ArrayList<>(10) - make an ArrayList object somewhere in memory and return the memory address of where it is. New is a function

ArrayList<Integer> here = - here goes on the stack as it's a local variable, but here is not the object itself. ***li*** is also a memory address, when we set the value of ***li*** to here (we copy over the memory address) hence we can use li further on.

li = here just copies the memory address of here to li. So it creates a reference to the object which is in memory, being able to manipulate it further.

Anything made with new goes on the heap. We have to use the heap if we want something to still exist after the method has finished since it will not fall out of scope.

There is a dotted line that represents the stack boundary, that is the maximum space the stack can use.

NB, when we call a method, the method parameters are pushed onto the stack, then the address of the next line of code.

The Heap

Heap: a block of memory starting at address x, finishing at address y.

- Some parts are free; other parts are in use.
- `new`: take some of the free memory, mark it as in use.
- Memory marked as free when we no longer need it.
    - Garbage collection in Java
    - Delete in C++

Example Question

```
 0 bool success = false;
 1 void getPIN() {
 2     int PIN;
 3     keyboardInput(PIN);
 4     if (PIN == 1234) { success = true; }
 5 }
 6 void login(int maxTries) {
 7     int trycount = 0;
 9     while (trycount < maxTries) {
10         getPIN();
11         if (success) {
12             // do something
13             return;
14         }
15         ++trycount;
16     }
17 }
18 void main() {
19     int maxTries = 3;
20     login(maxTries);
21 }
```

i. When the execution of code reaches the end of line 2 for the first time, what is on the stack? [4 marks]

ii. Assuming the code is running on a 32-bit machine – so integer variables and return addresses are each 4-bytes in size - how much stack space has been used at this point? [1 mark]

NB, in this case success is a global variable (we know its global because to be it, it has to be either in a class or in a method) Anything that is not in a particular scope must be global.

i) We start at main.
Right being the top of the Stack

maxTries	(method params)	Next command line	tryCount	Next command line	PIN
3	3	21	0	11	null

ii) there are 6 variables on the stack at 4 bytes each. 6 x 4 = 24 bytes

Choosing which holes to use on the Heap

Say we request memory of size **n** and there are a few slots available. We have to find a slot **m** which is larger than n. (m >= n). Any leftover space becomes a new hole (m - n).

- **First-fit:** the first hole of size m <= n.
- **Best-fit:** the smallest hole that m >= n.
- **Worst-fit:** the largest hole available (in which m >= n) and creates the largest remaining hole (m - n) note that adding a MCB will decrease the remaining hole.

We can implement a Memory Control Block in the following manner:
Memory Control Blocks are stored on the Heap and they allow us to manage the memory in the first place.

```
class MemControlBlock {
public:
    bool available;
    int size;

    MemControlBlock* previous;
    MemControlBlock* next;
}
```

It must state:

- If the memory chunk is available
- How big is it (size of the total available space)
- A doubly linked list (pointer to the next and previous memory control block.

*The total amount of storage used in this case is 1 byte for **available** and 4 bytes for **each other variable** (total 13 bytes) but on an a 32 bit machine it assumes that where things are stored is in memory it assumes it's a multiple of 4 bytes, therefore the amount of memory stored is 16 bytes. The compiler adds padding (wasted free space). Note if we declare 4 booleans first then it doesn't add padding.*

Some of the space in the holes is taken by memoryControlBlock which tell us information about the space and wether is available (as well as links to the next and previous block)

NB, memory control blocks are doubly linked.

Leftover Space

```
int spare =  curr->size - size
           - sizeof(MemControlBlock);

if (spare > 0) {
  ...make a new MemControlBlock...
}
```

Whenever we make a new memory allocation and there's spare space (checked with the code on the side) we will create a new memory control block. In the heap memory control blocks and memory allocated has to alternate. But what happens if we allocate 124 bytes of data to a 128 byte whole. We have 4

bytes left which is not enough for a new memory control block. This causes **internal fragmentation.**

Addressing Memory

- A MCB is at address x.
- The block of memory it is controlling is 16 bytes later (immediately after it)
- To get to it we have to add 16 bytes to x.
- In C++ bytes are chars so we can do the following.

```
char * y = reinterpret_cast<char*>(x);
y += 16; // is now 16 bytes after x
```

Placement new

Placement new allows us to specify where in memory we want our object to be stored. We can do so in the following way.

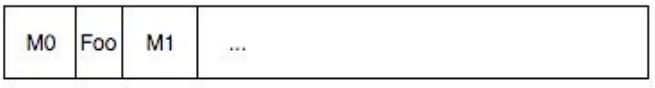
```
MemoryControlBlock * mcb = new(memoryAddress) MemoryControlBlock(...);
```

When we use delete on the a normal new it **calls the destructor** and **marks the memory as being free** but when using placement new we have to manage our own memory:

- Using `placement new` you have to sort out and call your own destructor `(y->~Foo();).`
- Take 16 bytes off of **y** to get the MemControlBlock address and `reinterpret_cast<MemControlBlock*>(...)`
- Then mark it as available and merge it with other free blocks if necessary (if there are any).

NB, new returns a pointer to the memory allocated.

Reminder: to do arithmetic on memory addresses I have to use reinterpret_cast.



If we created Object Foo with placement new, when we call the destructor we will do the following.

Say we created a pointer of type Foo called f.

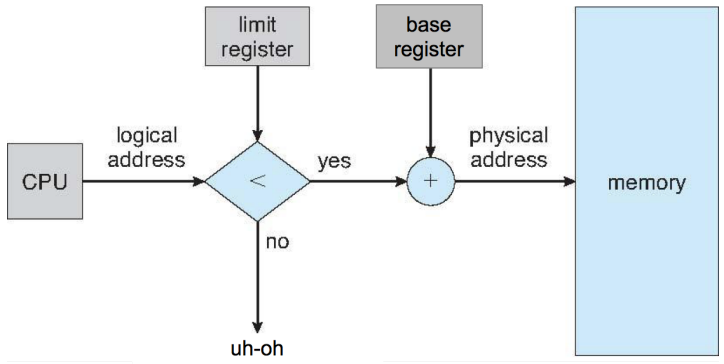
```
// call the destructor for Foo.
f->~Foo();
// change the type to char so we can do memory arithmetic
Auto x = reinterpret_cast<char*>(f);
X -= 16; // we decrease the value by 16, now we are at the start of M0
Auto y = reinterpret_cast<MemControlBlock*>(x); // we can now merge
blocks and other things.
```

NB, that in actual code it's bad practise to hardcode 16. You should really write `sizeof(MemControlBlock);`

Exam

Note that they try to catch you with the LAST chunk or hole of memory in Memory! Note that you ALREADY have a MCB there! Say our last chunk has 60 bytes spare and we want to allocate 60 bytes WE CAN because we already have a memory control block in place. NOte that this just occurs at the end since we created a MCB when allocating the previous chunk of memory. Also when memory partition is given, assume that is what goes after the memory control block (you don't have to subtract 16)

Memory Management for a Process

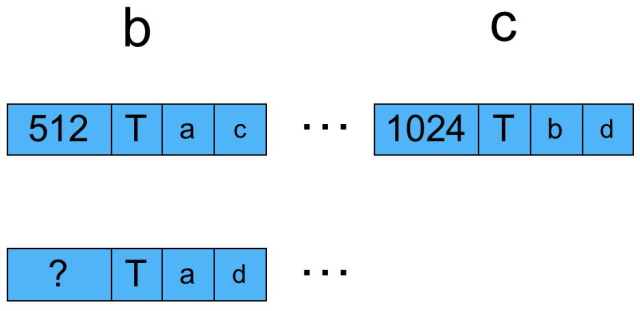


Base register: records where the memory starts (base)
Limit register: records how much memory is available (limit)

Memory binding turns logical memory into physical memory

Memory Blocks are Doubly Linked

Memory blocks have a previous and a next (which point to the previous memory block and the next) hence it's doubly linked.



Having a doubly linked list enables us to merge with available memory which is before our current point.
If we merge the 512 and 1024 memory blocks:

512 + 16 + 1025 = total space available
**** (since the Memory Control Block is 16 bytes and that is now not useful) ****

NB, When updating the MCB of the previous we need to update the available size, and the link to the next MCB (which is the next MCB for the one that we are deleting).

This enables us to ensure that we don't have holes in our memory which causes **external fragmentation** which occurs when your memory is full of holes and you have not been merging them together you can have lots of free small amounts of space which makes the space in our heap useless.
External fragmentation: occurs when there is enough memory to meet the allocation request but memory is not contiguous.

Memory pools

Help us avoid fragmentation.
If we have memory arranged like so:
We have small objects and large objects.

When we delete two large objects:
Then we have a small object fragmenting our memory.

Fragmentation occurs since we cannot use the whole large block as one as the green small memory occupation is using it up. Rather we'd prefer to arrange the information like so:

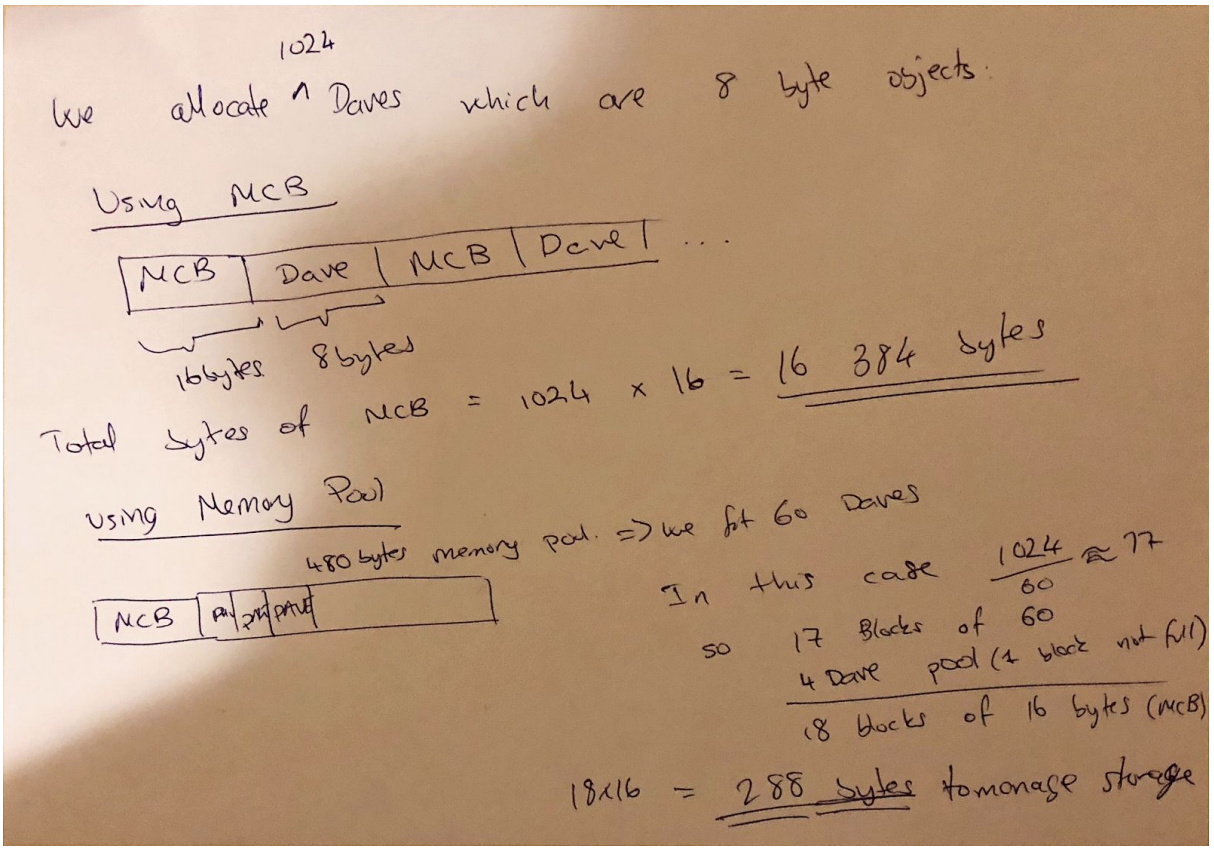
Memory pool: will allocate space for several small objects in **one contiguous block**

- Say our small objects are 16 bytes each and we allocate space for 128 of them at once (2048 bytes). It then splits that into little bits and keeps a singly linked list which are free.
- Allocation request: pop item off the list and fills it up.
    - If none left make another block's worth (2048 bytes of small space)
- Deallocation request: push item onto the list
- The block itself is only deallocated when every object inside its free.

- Downside is that if you have one 16 byte object allocated you have to keep the whole 2048, in practise these work very well.

+ If we have a 8byte object and we have MCBs for each for every Kb of our object we have 2Kb of MCB which is inefficient. Using a memory pool we can use a single MCB

for each memory pool.



6 Deadlock

Starvation vs Deadlock

Starvation: there exists an interleaving in which a process doesn't enter its CS.
Deadlock: from a certain state there is no interleaving that will allow any process to enter the CS. (*Deadlock is a starvation of all processes).*
NB to get full marks in exam explain the process of starvation without deadlock, if it exists as well.

Deadlock in Semaphores

p2:	**wait(S1)**	**q2:**	**wait(S2)**
p2:	**wait(S2)**	**q2:**	**wait(S1)**

Note it should say p3 and q3 in the last line. If we are stuck waiting for another process to signal, but that other process is waiting a signal which has to be made by our original process we can lead to deadlock. Both processes are waiting for a signal from the other but this doesn't occur.

Programming Concurrency

- The risk of deadlock is high when writing concurrent programs
- Deadlock occurs in interaction between threads so the cause is not always visible.
- You may have to run the code for some time if deadlock is improbable.
- The longer the program runs the higher the risk for deadlock.

Necessary Coffman Conditions for Deadlock

All of the following conditions must hold for deadlock to occur:
Mutual Exclusion: only one process at a time can use an instance of a resource.

- NB if a solution doesn't hold for mutual exclusion this doesn't mean it will necessarily deadlock.

Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.
No preemption: a resource can be released only voluntarily by the process holding it, after it has completed the task it was doing.
Circular wait: each process is waiting for a resource that is being used by another. Such that there exists a set { P0, P1, … , Pn } in which:

- P0 is waiting for a resource held by P1
- P1 is waiting for a resource held by P2
- …,
- Pn-1 is waiting for a resource held by Pn, and P0 is waiting for a resource held by P0.

Bottom line: If I am a process, I'm always waiting for a resource held by the guy in front of me and the last guy is waiting for something from the first guy.

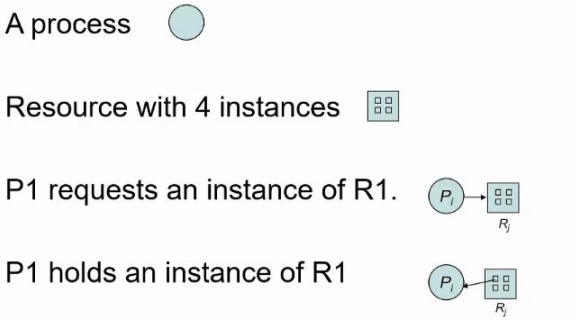
Handling Deadlock

Ostrich Approach

Ostrich approach: operating systems ignore deadlock, they assume that programmers know about it and will write code that doesn't deadlock. This is because **deadlock is rare** and **it's expensive to prevent and detect.**

Resource Allocation Graphs

A graph where every vertex represents a process and one representing each resource.
Request degree: an edge from a process to a resource means that a process has requested an instance of that resource.
Process -> Resource
P for please give me a resource.



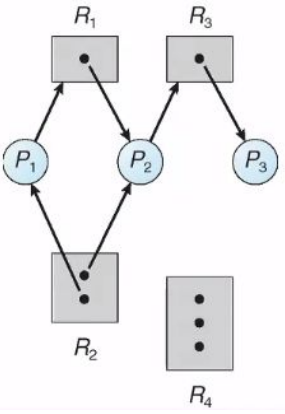
Assignment edge: an edge from a resource to a process means that process currently holds an instance of that resource.
Resource -> Process

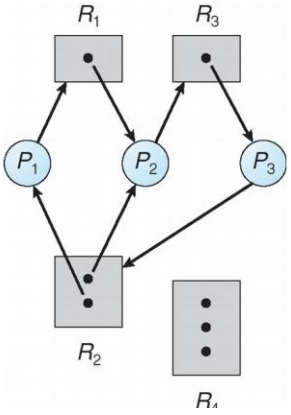
When a process **gets a resource** the direction of the arrow (edge) is flipped. We change from a request to an assignment edge.

Resources are **rectangles** and processes are **circles**

Defined by three sets:
- $P = \{P_1, P_2, P_3\}$,
- $R = \{R_1, R_2, R_3, R_4\}$,
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

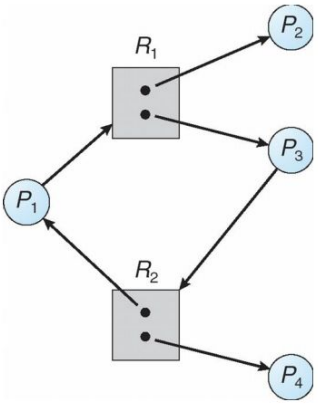
And the number of instances of each resource:
- $R_1:1, R_2:2, R:1, R_4:3$





Deadlock is present in this particular instance since P2 needs R3 which is held P3, P3 need R2 which is held by P2 and P1 (P1 needs R1 which is held by P2). All processes require something that is being held by another process to continue, thus we get deadlock.

We need R2 to be released but this will not occur.



No deadlock occurs here because P2 could run and release R1 when done and so could P4, this would enable P3 to run and P1 to run since now instances of R1 and R2 are available.

- If there are **no cycles** in the RAG then there is no deadlock.
- If there are cycles in the graph:
    - If there is only one instance of each resource then there is deadlock (that means it's a circular wait).
    - If there are multiple instances of resources there is a possibility for deadlock.

Ways to handle Deadlock

To handle deadlock we can do three things:

- **Prevention:** ensure the system cannot enter deadlock.
- **Cure:** allow deadlock to occur and then recover.
- **Ignorance:** ignore it and reboot.

Prevention

We have to break **one** of the conditions required for deadlock.

- Breaking **mutual exclusion.** In general we don't want to break mutual exclusion. A resource may need mutual exclusion thus we have to respect this. What we can do is only enforce mutual exclusion on resources the absolutely require it rather than all of them.
    - E.g. In the readers and writers problem, we might want to only restrict concurrent usage of the data to writers since readers can read simultaneously with no issue.
- Breaking **hold and wait**. Guarantee that whenever a process requests a resource it does not hold any others
    - Process must request and hold all its resources it will ever need before it can begin.
      Or
    - You can start running but you can only request resources when you have none, that means that it will have to stop, release everything and ask for all the resources again at once.
    - *Problems* of low resource utilisation and possible starvation.
- Breaking **pre-emption**. If a process requests a resource it cannot get immediately then it must release all the resources it holds. To enforce this a list of all the resources the process is waiting for must be kept. The process will only restart when all the process that you are waiting for are available.
    - There's risk of starvation since there is the possibility that some other process always takes a resource first.
- Breaking **circular wait. (Most usable in practise)** Impose a total ordering on a resource types and make all processes request resources in increasing order.

Deadlock Avoidance

To avoid deadlock we need extra info about resources:

- Max number of resources of each type a process will need. *This is not realistic since we don't know until we run a program what resource we're going to need.*

- If we ask the programmer to declare which resources its going to need, then we can use the **Resource Allocator State** of a system to enforce that we will never allow *circular wait (we decided that this is the property we are going to break)*.
    - If we have a single instance of each resource, we can check this using a resource allocation graph.
    - Else if we have multiple instances of each resource, we must use the Banker's Algorithm.

Resource allocation state: the number of available and allocated resources and the maximum number of resources each process can need.

Safe State

If a system is in a safe state it cannot deadlock. Else it might deadlock.
Every time a process in the system requests a resource we have to make sure that it leaves the system in a safe state.
Safe state: if there exists an order in which the processes can run in which the first resources need is available, the second process can be satisfied by what was available plus the resources that the first process has now released and so on.
We can number the processes so that one can run, then two, then three. The order is arbitrary and may change but it must be possible to run them in order.

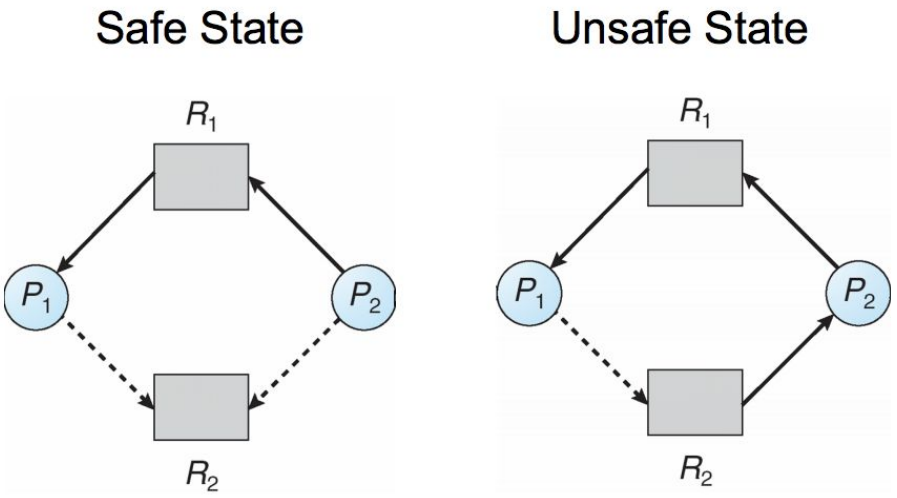
- If something needs a resource which is needed by a lower number process, then it has to wait for that lower number process to complete. **This means circular wait cannot occur since a process can only wait for a process which is lower in number.**
- Note that although we assign an order to processes, these processes will not always run in the same order, rather they will always run in an order in which the next can execute when the previous terminates.

Resource Allocator Graph Method

Claim edge: claim edges Pi -> Rj indicate that a process Pi can request a resource Rj. The system defined apriori that that process may require that resource. (dashed line)

- Claim edge becomes an assignment edge when the resource is allocated.
- Assignment edge reverts to claim when it releases it.
- All resources must be requested apriori (all claim edges are known and put in the graph initially).

If a process requests a resource it can only be granted if it doesn't create a cycle in the RAG.



The Banker's Algorithm (by Dijkstra)

When a process requests a resource we are going to do a check and if we detect that giving that resource will give us a problem, we will then wait to give it its resource. We will assume that each process will terminate the critical section in finite time and it will not quit during its critical section (this ensures it will release its resources).

- Used when we have multiple instances of each resource.
- Very computationally expensive O(mn^2) for n processes and m resources.

For n processes and m resources…

Allocation: *n* x *m* matrix. If Allocation[*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj*

Max: *n x m* matrix. If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*

Available: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type *Rj* available

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

R_0	R_1	R_2
3	3	2

Explanation:

- In allocation, P0 is allocated one copy of R1. P1 is allocated 2 instances of R0.
- In max, P0 will need a maximum of 7 instances of R0 (maybe not all at the same time but hypothetically it could occur).
- In available, tells us how many instances of resource are currently free.

Safety Check

1. Find a process for which there are sufficient available resources now to meet its maximum need.
2. Simulate executing the process i.e. release all its resources (make them to available)
3. Go back to 1.
4. If all processes have completed execution the system is safe.
5. If you cannot find such a process, but there are still processes that haven't executed, then the system is unsafe

Banker's Algorithm Formal Definition

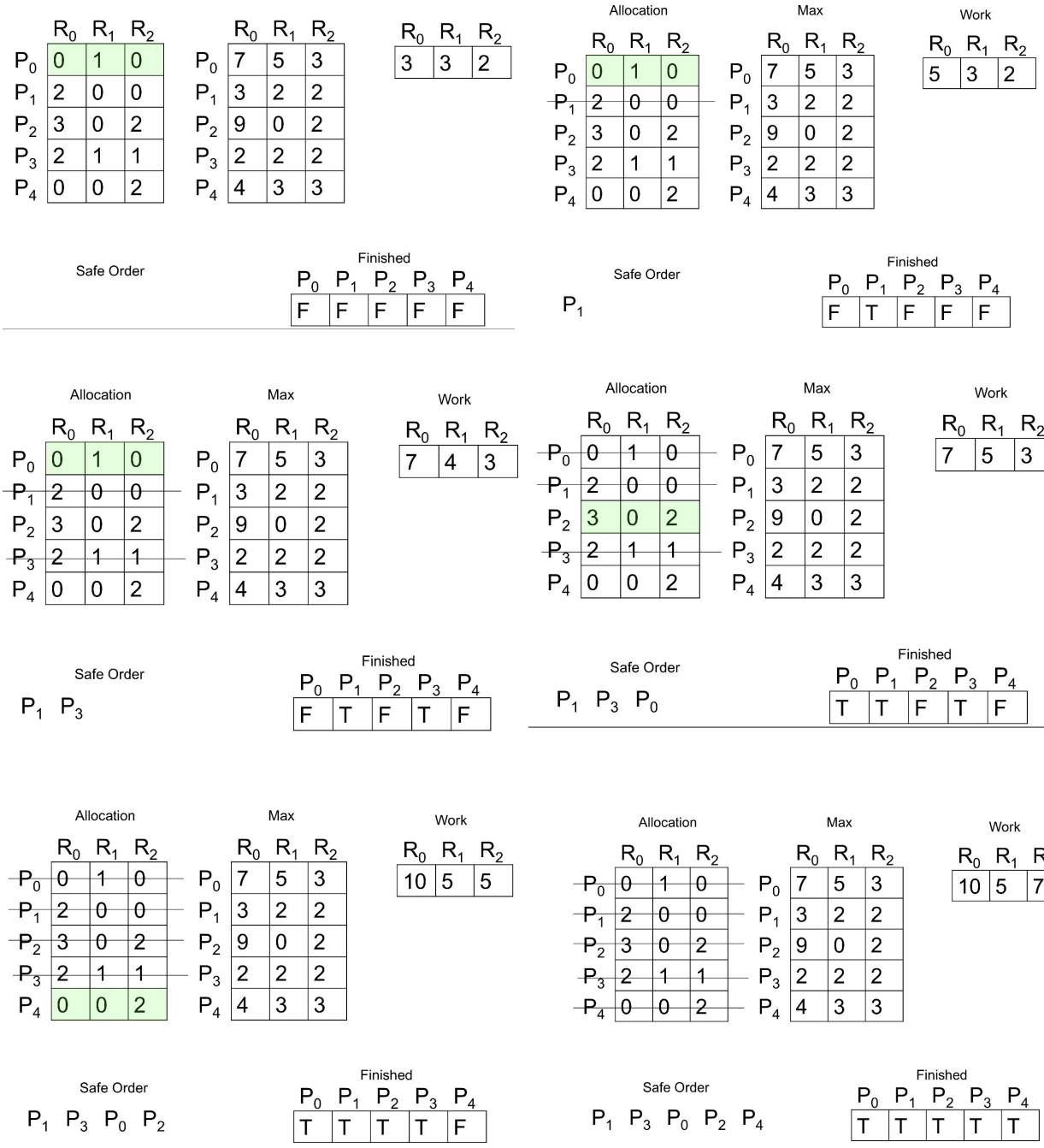
Note that the syntax is not what it normally means.

- Array1 >= Array2 (both same length) means:
    - Array1[i] > Array2[i] for all i.
- Array1 += Array2 means:
    - Array1[i] += Array2[i] for all i;

1. Let Work and Finish be arrays of length m and n, respectively.
   Initialize:
    - Work = Available
    - Finish [i] = false for i = 0, 1, …, n- 1
2. Find and i such that both:
    - Finish [i] = false and Max[i] - Allocation[i] <= Work
    - If no such i exists, go to step 4
3. Work = Work + Allocation[i]
   Finish[i] = true
   go to step 2
4. If Finish [i] == true for all i, then the system is in a safe state otherwise it is unsafe.

NB we will break ties by always picking the one with the lowest id.
Remember, when a resource runs, it releases its resources.

Example



NB, if multiple processes can run always choose the one with the smallest id.

If a process requests a resource:

- We update allocated as if the process had been allocated the resource
- We run the safety check algorithm
- If it's safe we allocate the resources
- Else we undo the update and we don't allocate them and the process must wait.

***Remember**, this process is O(mn^2), therefore every time we run it we are slowing down. To prevent this we will discover other solutions in the following section.*

Deadlock avoidance vs Circular-wait scheme

- A deadlock avoidance scheme tends to increase runtime overheads because we have to keep track of the current resource allocation.
- A deadlock avoidance scheme allows for more concurrent use of resources than schemes that statically prevent the formation of deadlock. A deadlock avoidance scheme could increase the system throughput.

Detecting Deadlock

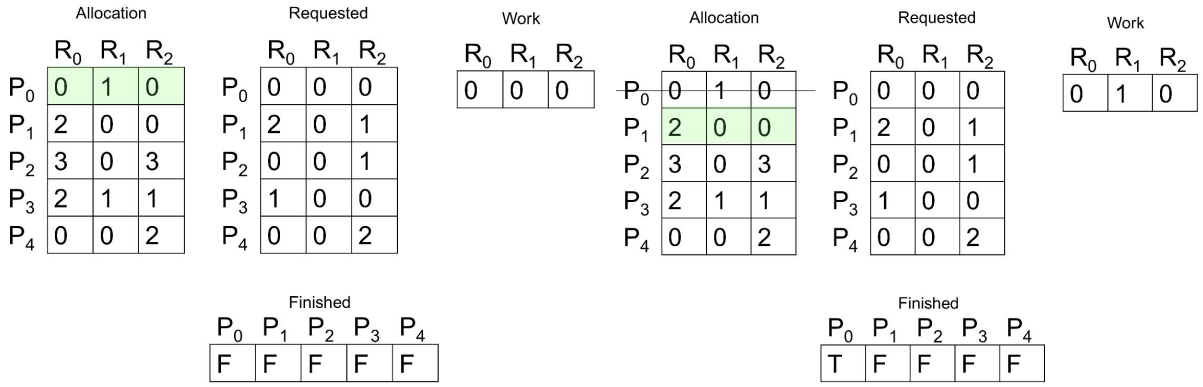
Detect when it has happened and try to recover from it.

- With a single instance of each resource we can look in the RAG for cycles.
- If there are multiple instances of each resource we will sue a variation of the Banker's algorithm:
    - Instead of using the **maximum** number of resources a process might need, use the resources a process is currently requesting.

1. Let Work and Finish be arrays of length m and n, respectively.
   Initialize:
    - Work = Available
    - Finish [i] = false for i = 0, 1, …, n- 1
    - ***If the process has no resources currently allocated Finished[i] = true.*** *(this will not help us executing any other process therefore lets ignore it).*
2. Find and i such that both:
    - Finish [i] = false and ***Request[i]*** - Allocation[i] <= Work
    - If no such i exists, go to step 4
3. Work = Work + Allocation[i]
    - Finish[i] = true
    - go to step 2
4. If Finish [i] == true for all i, then the system is in a safe state *otherwise* ***if Finish[i] == false then the process is deadlocked.***

We assume that if the process has sufficient resources to continue now it will complete. But actually, it may require more resources in the future. That's okay since we'll detect deadlock when those resources are requested.

Example



	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

	R_0	R_1	R_2
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

R_0	R_1	R_2
0	0	0

	R_0	R_1	R_2
P_0	~~0~~	~~1~~	~~0~~
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

	R_0	R_1	R_2
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

R_0	R_1	R_2
0	1	0

P_0	P_1	P_2	P_3	P_4
F	F	F	F	F

P_0	P_1	P_2	P_3	P_4
T	F	F	F	F

We've been through all the process and none of them can run, since now we check for the resources available (work) against the requested **without** including the already allocated.

NB, *requested refers to resources requested on top of the ones allocated.*

Exam*, whenever we have a question regarding the Banker's algorithm, pay attention if the question is "this system in a safe state" to do deadlock prevention or "is this system deadlocked" to do deadlock detection.*

We call the deadlock detection algorithm whenever we want. How often we call it depends on how likely is the system to deadlock and how many processes will need to be rolled back (how many processes have to be killed).

- If we wait too long, it might be difficult to detect what caused the deadlock since now there are multiple deadlocks (multiple cycles).

We assume that there will not be any form of circular-wait in terms of resources allocated and processes making requests for them.The optimistic assumption is that there will not be any form of circular-wait in terms of resources allocated and processes. This assumption could be violated if a circular-wait does indeed in practice.

Recovery from Deadlock

Process Termination

We should abort all deadlocked processes. Abort one process at a time until the deadlock goes away. We should abort each process in the order:

- Priority of the process
- How long the process has computed, how much longer until completion

- Resources used
- Resources needed
- How many processes will need to be terminated
- Is the process interactive or batch (imagine you kill word when someone hasn't saved)

Resource Pre-emption

Instead of killing a process and bring it back from the start we can select a process and take resources from it to break the deadlock.

- Taking resources away and telling it to wait.
- Rollback - return to a safe state and restart the process from that state
- Selecting a victim which minimizes cost (victim has to have lowest value for us)
- Risk of Starvation with a same cost function we could always pick the same process as victim. We will include number of rollbacks in cost function to make sure this doesn't occur.
    - If we select the same victim multiple times, the cost of victimizing it will be higher and higher and it will not occur.

9 Protection and Security

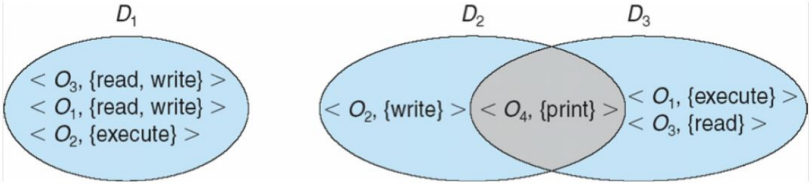
Protection problem: ensure that any active program is accessing resources in a way that is specified in a certain way (policy).
Policy: tells us who should be accessing our system resources and in which way.
Principle of least privilege: tells us that systems and users are only given just as much as they need to complete their tasks, not more. So that if there is a security breach, less damage is made.

- Limits damage if entity has a bug, gets abused.

Protection Domain

A process normally executes within a protection domain.
Protection domain: specifies a number of objects and operations that can be performed on these objects.

Domain = set of <object-name, access rights (or operations that can be performed)>



In order to execute O3 you must be in D1 or D3.

Domain can be a user, process, procedure

The Need to Know Principle

- A process should be allowed to access only those resources for which it has authorisation
- At any time, a process should be able to access only those resources that it currently requires to complete its task.

If the association between process and domain is **fixed**, we should have a way to change the content of a domain so that it reflects the **minimum necessary access rights**.

If the association between processes and domains is **dynamic**:

- A **Domain switching** mechanism is available, enabling the process to switch from one domain to another (may want to allow the content of a domain to be changed)

Access Matrix

- An abstract model of protection
- Rows -> domains
- Columns -> objects
- Entry(i,j) is a set of access rights: set of operations that a process executing them in domain Di can invoke on object O

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Control over **dynamic process associations**

Switch: operation from one domain to another.

- Switching from Di to Dj is allowed ⇔ switch ∈ Entry(i,j)
- A process executing in a particular domain can switch to another domain if and only if it has the switch operation defined in the entry of another domain.
    - E.g. A process executing in Domain D2 can also execute in Domain D1.
    - I.e program with two modes, user mode and kernel mode. A process can switch to kernel, execute some operations and then back to user.

Control over **Content Change**

We need some mechanism to allow us to change the content of the entries. We have 3 operations:
Copy (denoted by *): allows an access right to be copied within the column (that means for that object) for which the right is defined.

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

When a right R* is copied from Entry(i,j) to Entry(k,j) we can have:

- **Copy:** R* remains in Entry(i,j), and R* is created in Entry(k,j).
- **Limited Copy:** R* remains in Entry(i,j), and R is created in Entry(k,j).
    - *Img (a) and (b) -> for F2, read access has been granted within D2. The process in domain D3 will not be able to copy the right.*

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

- **Transfer:** R* is removed from Entry(i,j), and R* is created in Entry(k,j).

Owner operation:

- If owner ∈ Entry(i,j), then a process executing in domain Di can add and remove any right in any entry in column j (i.e object Oj)
- Basically if you are the owner, you can change any rights anywhere within that column (object).
- D1 is the owner of F1 thus it can revoke D3 access to execute.
- D2 is the owner of F2 and F3 and it can also make changes.

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

Control operation:
Will allow us to change the rights within the row.

- Defined only to domain objects
- If control ∈ Entry(i,j) then a process executing in domain Di can remove any right from row j.

- A process in Domain D2 will be able to change any access rights to any entries in the row associated with domain D4.

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

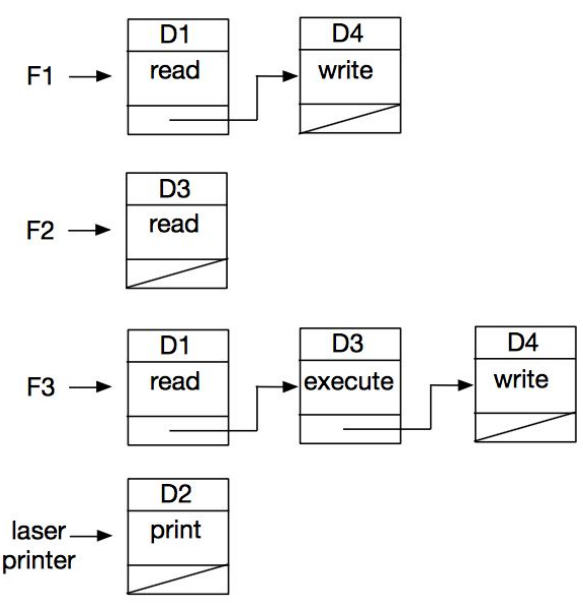
Implementation of the Access Matrix

There are two ways to implement the Access Matrix. Store the matrices by row or by column, By row (**Capability Lists for Domains)** or by column (**Access Lists for Objects**).

Access List for Objects

In this case we are storing the matrix **by column**.
As an object (or process) in what environments can I do what.

Each column is implemented as an access list for an object. It lists every objects in pairs of entries
<domain, access rights>
defining all domains with a nonempty set of access rights for that object.
May contain a default set of access rights for the object.



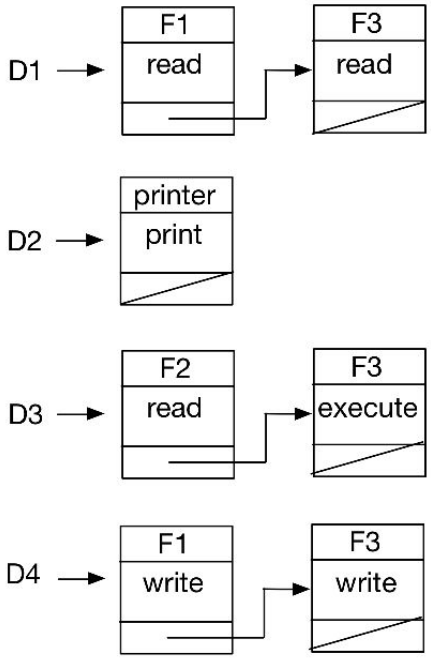
domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Capability Lists for Domains

In this certain environment, these processes/objects can do these things.

Each row is implements a capability list for one domain:

- a list of objects together with the operations allowed on these objects. It must be **unforgeable** (not directly accessible by a process in the domain).
- F1 can read D1. F3 can execute D3.



domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Comparison of implementations

- Access Lists for Objects
  - \+ Corresponds to need of users
  - \+ Revocation or expansion of access rights to an object is easy
  - \- Determining access rights by domain is difficult and expensive. It has to be done every time the object is accessed.
- Capability List for Domains
  - \+ Easy to localise information for a given process
  - \+ Faster access to objects
  - \+ Capabilities can be passed from one domain to another easily.
  - \- Do not correspond directly to need of users
  - \- Revocation of access rights might be inefficient and difficult

Operating systems use a combination for both.

Traditional Unix File Access Control

Unix File Access Control System is a simplified ACL system.

- Every user has an id. The super user or **root** sets file permissions and has an id of 0.

We have 3 classifications of users with each file:

- **Owner:** user which created the file.
- **Group:** a set of users who are sharing the file and need similar access.

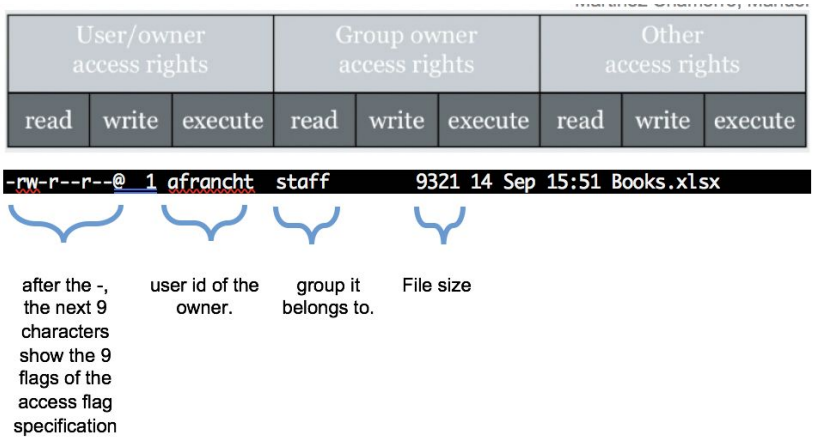
owner	group	other
rw-	r--	---

- **Universe:** all other users in the system

Three modes of access

- **Read(r):** view the contents
- **Write(w):** modify or remove content
- **Execute(x):** run as a program (if its a directory -> you can enter and run any file inside of it)

To see the above permissions use ***$ls -l*** to see the permissions in the directory or $chmod (change mode) to change the permissions.




```
$ls -l    testfile
-rwxrwxr--  1 lina  staff 1024  Feb 12 13:00  testfile
$ls -l    testdirectory
drwxr-----  2 lina  staff 25000  Sep 5 18:30  testdirectory
```

In the second row, the first letter is a ***d*** hence this is a directory.

Chmod

We can use change mode to change the permissions of a file or a directory.

Symbolic mode:

- operator +,
- operator -
- operator =
- add, remove or specify.

```
$ls -l    testfile
-rwxrwxr--  1 lina  staff 1024  Feb 12 13:00  testfile

$chmod  o+wx    testfile
$ls -l    testfile
-rwxrwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  u-x    testfile
$ls -l    testfile
-rw-rwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  g=rx    testfile
$ls -l    testfile
-rw-r-xrwx  1 lina  staff 1024  Feb 12 13:00  testfile
```

Absolute mode (using numbers) (its actually each number in binary):

- 0 -> (---)

- 1 -.> (--x)
- 2 -> (-w-)
- 3 -> (-wx)
- 4 -> (r--)
- 5 -> (r-x)
- 6 -> (rw-)
- 7 -> (rwx)

```
$ls -l    testfile
-rwxrwxr--  1 lina  staff 1024  Feb 12 13:00  testfile

$chmod  777    testfile
$ls -l    testfile
-rwxrwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  677    testfile
$ls -l    testfile
-rw-rwxrwx  1 lina  staff 1024  Feb 12 13:00  testfile


$chmod  657    testfile
$ls -l    testfile
-rw-r-xrwx  1 lina  staff 1024  Feb 12 13:00  testfile
```

There are 12 protection bits in Unix. 9 are to specify read/write/execute permissions for the owner/group/other. The other 3 are to:

- Set User ID **UID** and set group ID **GIP** are additional two bits for privilege escalation.
    - 0: if a user has write permission to the directory, they can rename/remove files.
    - 1: only the file owner, directory owner and root can do so.

Example: /usr/bin/passwd

- The **passwd** command changes the password of a user
- To do this we must be able to write to /etc/shadow
    - Owned by root, permissions rw-------
- The password program has to give the user additional permissions so they can write to /etc/shadow
- This can be done via **set UID** and **set GID**

Privilege Escalation

- A process has a
    - Real User Id (RUID): the user that started the process. Never changes
    - Effective User Id (EUID): the user which the process appears to run, the one used for permission purposes.
    - Real Group Id and Effective Group Id
- **Set UID** bit:
    - If 1, when the process is started EUID = owner of executable
    - If 0, then EUID is the User Id of whoever started the process.
    - When we set UID to 1 on a n executable file, any other user that is running this file can run it as a user of the file.
- **Set GID** bit:
    - If 1, when process is started, the Effective Group Id = the group of the executable

- Example: when /**usr**/**bin**/**passwd** has **set UID** bit=1:
  - Executable file owned by root
  - Hence, **EUID** is then root, so can write to /**etc**/**shadow**

```
$ls -l  /usr/bin/passwd
-rwsr-xr-x  1 root  root 19031  Feb 7 13:47  /usr/bin/passwd
```

***S** stands for UID is equal to 1. When we execute the file it will appear as the owner, which is root. The real user id would be whoever started the process but the EUID would be root.*

The risk of Set Id, Owner = root, **set UID** = 1 means the process runs as root, which permits essentially anything.

Access Control Lists in UNIX

- Main limitation of owner/group/other model:
  - We must try to capture all use in three cases.
  - One group per file
  - Only the owner/root can change permission, not multiple users.

Example: POSIX ACL - modern UNIX-based operating system ACL

- Still have an owner, owning group and other permissions.
- We can grant permissions for additional **named** groups/users
- Mask - allows us to control the permissions that we can assign to named users and named groups.
- Mask provides an upper bound for the permissions of named users and groups.

Entry Type	Text Form
Owner	`user::rwx`
Named user	`user:name:rwx`
Owning group	`group::rwx`
Named group	`group:name:rwx`
Mask	`mask::rwx`
Others	`other::rwx`

```
$ls -l    testfile
-rwxr-----  1 lina  staff 1024  Feb 12 13:00  testfile

$ getfacl  testfile
# file: testfile
# owner: lina
# group: staff
user::rwx
group::r--
other::---
```

***$getfacl* command** will give us the information about the file

Adding a named user using $setfacl -m (m flag is modify).

```
$ setfacl -m user:mike:rwx testfile

$ getfacl  testfile
# file: testfile
# owner: lina
# group: staff
user::rwx
user:mike:rwx
group::r--
```

Think that mask gives us what the ***old group*** can see. Than is named users, named groups and group.

Using ***$ls -l*** shows us something different. It shows us a + telling us it's an extended form and it shows us the mask. We can change the mask using ***$chmod***

```
$ chmod g-w testfile

$ ls -l testfile
-rwxr-x---+ 1 lina  staff 1024  Feb 12 13:00  testfile

$ getfacl testfile
# file: testfile
# owner: lina
# group: staff
user::rwx
user:mike:rwx            #effective:r-x
group::r--
mask::r-x
other::---
```

In this case user:mike will not we able to write, nor the group.

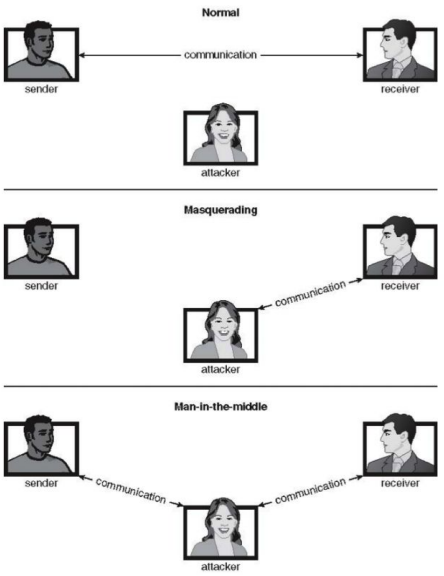
The Security Problem

- System is **secure** if resources are used and accessed as intended under all circumstances (impossible) since this includes external factors.
- **Intruders** (crackers) attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security
- Attack can be accidental (easier to protect) or malicious

Security Violation Categories

- **Breach of confidentiality**: unauthorized reading of data (or theft of information).
- **Breach of integrity:** Unauthorized modification of data
- **Breach of availability:** Unauthorized destruction of data
- **Theft of service**: Unauthorized use of resources
- **Denial of service (DOS):** Preventing legitimate use of the system

Security Violation Methods



- **Masquerading:** pretending to be an authorized user to escalate privileges (breach authentication)
- **Replay attack:** As is or with message modification
- **Man-in-the-middle attack:** Intruder sits in data flow, masquerading as sender to receiver and vice versa
- **Session hijacking:** Intercept an already-established session to bypass authentication

Security Measure Levels

- **Physical:** Data centers, servers, connected terminals (physical security)
- **Human**: avoid social engineering, phishing, dumpster diving
- **Operating System:** Protection mechanisms, debugging
- **Network:** intercepted communications, interruption, DOS

User Authentication

To authorise a user we first need to authenticate him, often using passwords. Passwords can be:

- Guessed
- Accidentally Exposed (shoulder surfing)
- Sniffed
- Brute-forced
- Illegally transferred

To make a password more secure we can:

- Encrypt it
- Use one-time passwords
- Biometrics: finger attribute (fingerprint, hand scan)
- Multi-factor authentication (USB dongle, biometric measure and password)

10 Security

Passwords

Encrypt to avoid having to keep secret

- But keep secret anyway (i.e. Unix uses superuser-only readably file
- /etc/shadow)
- Use algorithm easy to compute but difficult to invert (hash it)
- Only encrypted password stored, never decrypted

- Add salt to avoid the same password being encrypted to same value (prevents dictionary attack)
- One-time passwords
- Use a function based on a seed to compute a password, both user and
- computer
- Hardware device / calculator / key fob to generate the password
- Changes very frequently
- **Biometrics:** some physical attribute (fingerprint, hand scan)
- **Multi-factor authentication:** need two or more factors for authentication, i.e. USB dongle, biometric measure, and password

Program and System Threats

Trojan Horse
A program that secretly performs some malicious action in addition to its visible actions (e.g. login emulator resulting on stealing the username and password). It can be deliberately programmed or a consequence of another programmer.
Trap Door
A designer or a programmer deliberately inserts a security hole that they can use later (could be included in a compiler, difficult to detect since we won't find any evidence of it in the code as it occurs in the compiler).

- An error in rounding in bank transactions may send the extra money to another bank account.

Logic Bomb
Program that initiates a security incident under certain circumstances.
Virus
Fragment of code embedded in an otherwise legitimate program. It can replicate itself (by infecting other programs).
Worm
A program that copies itself onto another system. Consumes system resources, often blocking out other, legitimate processes.
Stack and Buffer Overflow
The attacker do the following:

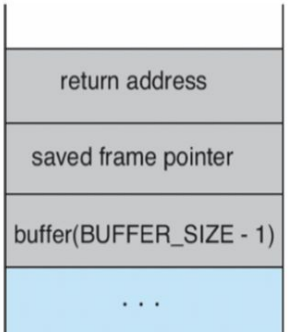
- Exploits a bug in a program (e.g. failure to check bounds on inputs, arguments).
- Change the return address on the stack so that when it returns, it points to a point in which there is malicious code which is executed.

Solution:

- Adopting a better programming methodology by performing bounds checking. Note that sometimes even though there is bounds checking, a user can enter malicious code by passing binary code as parameters.
- Prevent the execution of code that is located in the stack segment of a process's address space via using special hardware support.

The user tries to give for input than the buffer can take to try to rewrite other places in the stack or buffer, the return address.

```
#include <stdio.h>
#define BUFFER SIZE 256
int main(int argc, char *argv[])
{
   char buffer[BUFFER SIZE];
   if (argc < 2)
```



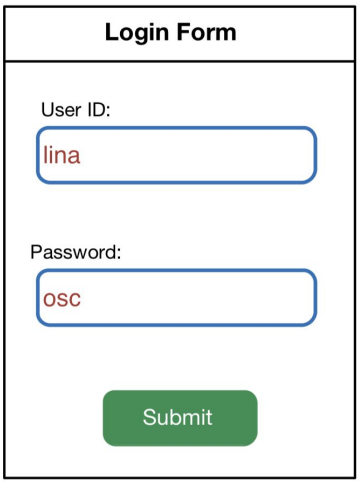
SQL Injection

One of the most dangerous threats for web applications. It allows attackers to have unrestricted access to the databases underlying the applications.

- Any application that uses user input as input into an SQL query is vulnerable to the attack.
- Attack occurs when the user input is treated as SQL code.
- SQL injection attack are caused by insufficient validation of user input.

SQL Injection Mechanisms:

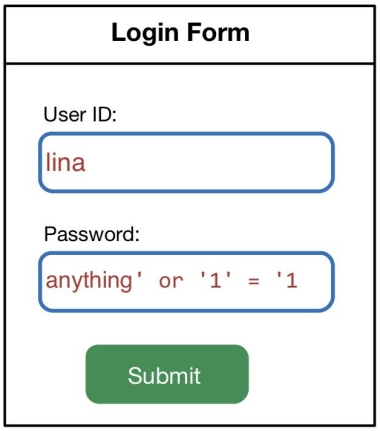
- Through user input
- Cookies (file generated by the web app and stored in the users browser, used to store information, malicious code can be inserted there for later use)
- Server variables
- A login form may create a DB query like this one:




```
SELECT * FROM user_info WHERE
userid = 'lina' and password = 'osc'
```

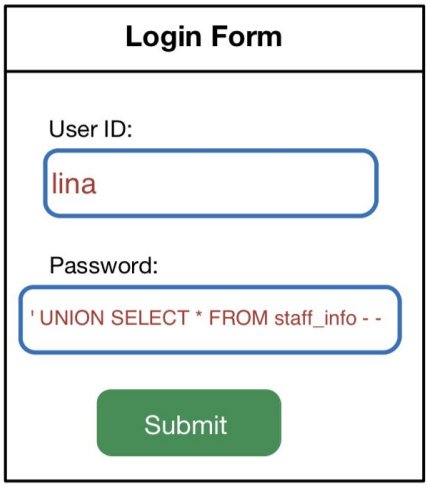
Types of injection

- **Tautologies** - by injecting a parameter that will always be resolved to true. Used to bypass authentication, extract data and identify injectable parameters.
    - Not starting or final quote is added since it's assumed it will be enclosed within string delimiters.
    - '1' = '1' will make the condition always to be true. Not only the user will be allowed access since the query returns true but he will be able to view all the information. (NB in this example we are printing our result to the user)



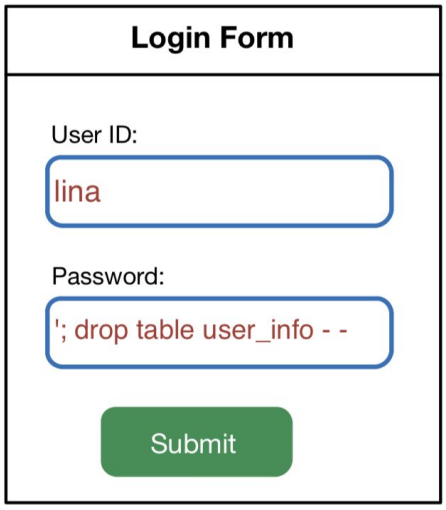

```
SELECT * FROM user_info WHERE
userid = 'lina'
and password = 'anything' or '1' = '1'
```

- **Union query** - bypass authentication, extract data




```
SELECT * FROM user_info WHERE
userid = 'lina' and
password = '' UNION SELECT * FROM staff_info --'
```

  - Some knowledge of the database is required (e.g. we need to know the name of the tables).
  - **--** in SQL is used to comment out. --’ makes it ignore the string delimiter which is put in by default.
- **Piggy-Backed Queries** - extract, add or modify data. Execute remote commands, denial of service
  - For this attack to be allowed, the database requires that two commands can be chained.




```
SELECT * FROM user_info WHERE
userid = 'lina' and
password = ''; drop table user_info --'
```

- **Illegal/Logically Incorrect Queries** - Identify injectable parameters, identify database, extract data
    - In the default error message, table names and other database information is shown

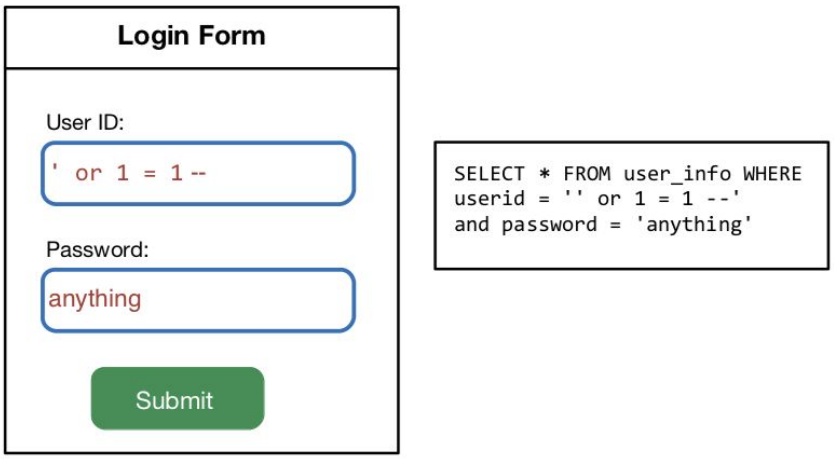
Pin Code

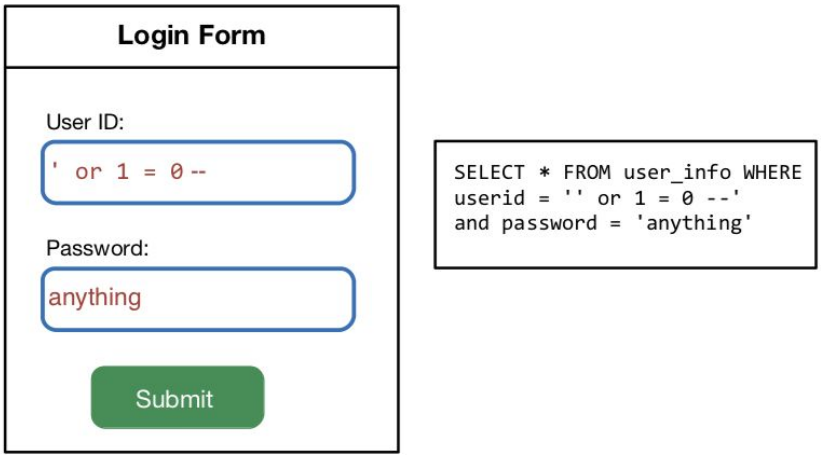
```
convert(int, (select top 1 name from sysobjects where xtype= 'u'))
```

Error Message:

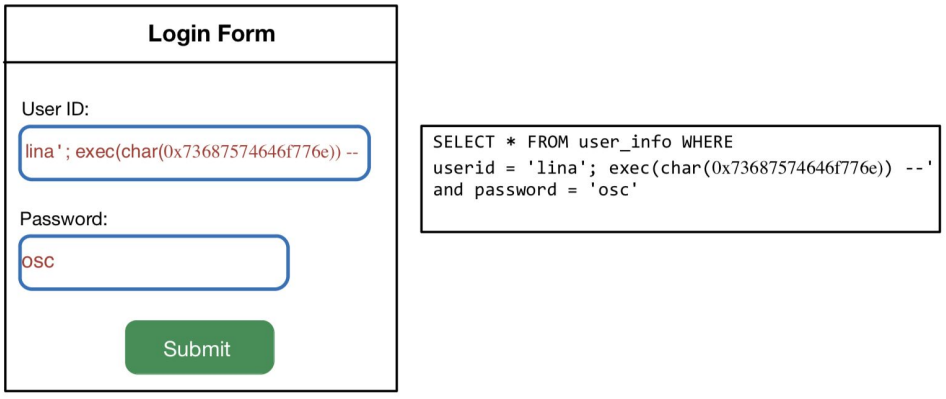
Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int.

- **Inference -** we supply some data and we observe what happens to understand how it works. Used to Identify injectable parameters, identify database and extract data.
    - We make the query and simply observe to see what the response is to find ways to exploit the system.





- **Alternate Encodings -** executing a command using encodings, rather than raw text, used to evade detections.
  - In the example below, $char(0x73687574646f776e)$ translates to SHUTDOWN.



In fear that the shutdown command might be detected by the programmer we use encodings.

Defending against SQL Injection

Using **parameterized queries** not data which is directly implemented:

- Don't use the data the user inputted directly.
- Specify placeholders for parameters
- If we use parameters, Input will be treated as data (as a string) not SQL commands
- We create a prepared statement so it's compiled and then we give the parameters, then we execute the query.
- Anything that we supply now will be treated as data since we have precompiled the code.

Not Parametrised Query

```
String query = "SELECT * FROM user_info WHERE userid= '" + request.getParameter("uid")+ "'"
AND password='" + request.getParameter("passid") + "'";

Statement statement= connection.createStatement();
ResultSet rs= statement.executeQuery(query);
```

Parametrised Query

```
String query = "SELECT * FROM user_info WHERE userid= ? AND password= ?";

PreparedStatement statement=
                  connection.prepareStatement(query);
statement.setString(1,request.getParameter("uid"));
statement.setString(2,request.getParameter("passid"));
ResultSet rs= statement.executeQuery();
```

Input validation

- Validate user input by checking type, length, range, etc.
- Check for certain characters and character sequences such as ;, ' and --.
- Create low privileges accounts for use of applications
- Make database error reporting not descriptive.

Cryptography

Constrain the potential senders and/or receivers of a message.
Based on keys used for:

- **Confidentiality -** others cannot read the content of message
- **Authentication -** determine origin of message
- **Integrity -** verify that message has not been modified
- **Nonrepudiation -** sender should not be able to falsely deny that a message was sent.

Terminology

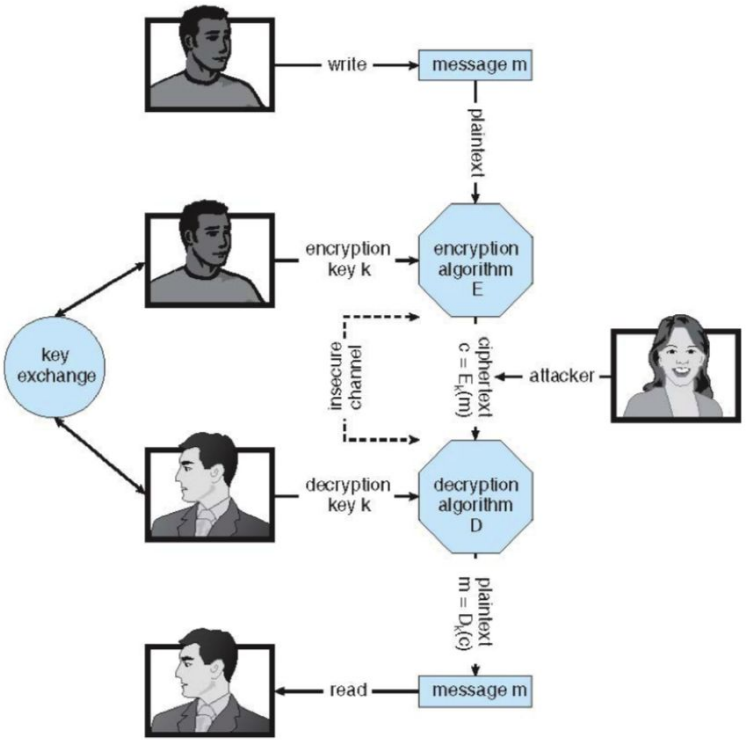
- ***M* — message or plaintext**
- ***C* — ciphertext**
- ***E_k* — encrypting function, using key *k***
- ***D_k* — decrypting function, using key *k***

Symmetric Cryptography

Encryption and decryption using the same key k. The key must be kept a secret.

$$C = E_k(M),\ M = D_k(C)$$

- Examples:
  - DES
  - 3DES
  - AES
  - RC5
- Key length determines number of possible keys
  - DES (56-bit key) — $2^{56} = 7.2 \times 10^{16}$ possible keys
  - AES (256-bit key) — $2^{256} = 1.1 \times 10^{77}$ possible keys

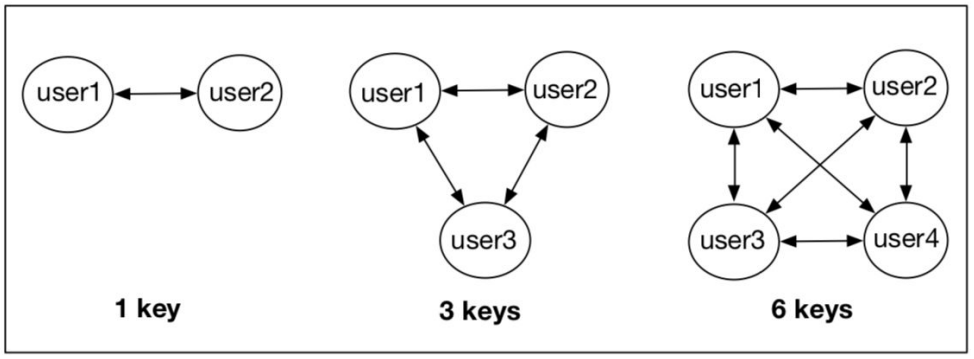


Key Distribution

Key distribution is a big challenge since it must occur with no interference.
Each pair of users needs a separate key for a secure communication.

- We need multiple keys for multiple users since we try to keep our keys as secure as possible.

$$n \text{ users} : \frac{n(n-1)}{2} \text{ keys.}$$



Secure key distribution is the biggest problem with symmetric cryptography.

Asymmetric Cryptography

Sometimes called public-key cryptography. This solves multiple problems regarding key distribution in symmetric cryptography.

Each user has **two** keys:

- **public key** k_1 — published key, known to everyone
- **private key** k_2 — only known to the user

$$C_1 = E_{k_1}(M), \quad M = D_{k_2}(C_1)$$
$$C_2 = E_{k_2}(M), \quad M = D_{k_1}(C_2)$$

Unlike symmetric cryptography, not every number is a valid key.

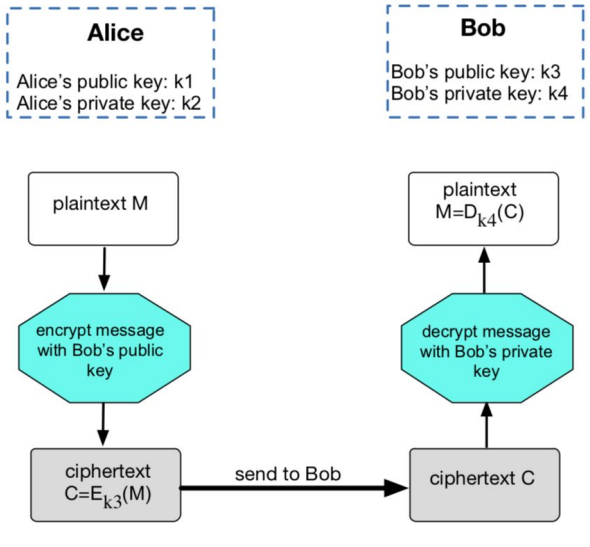
Examples:

- RSA
- DSS

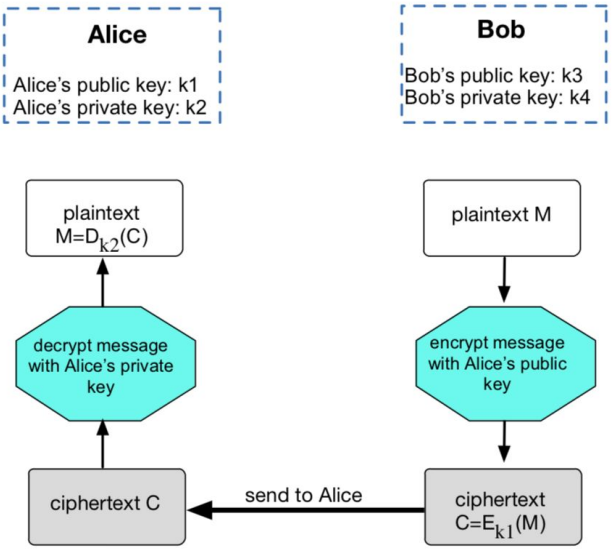
- **Authentication** is performed by having the sender send a message that is en-coded using the sender's private key.
- **Secrecy** is ensured by having the sender encode the message using the receiver's public key.
- Both **authentication** and **secrecy** is guaranteed by performing double encryption using both the sender's private key and the receiver's public key. Sign it and then encrypt it.

* A message that is encrypted with the public key, can only be decrypted with the public key and the other way around.

Alice wants to send a **confidential (secret)** message to Bob.



Bob wants to send a **confidential (secret)** message to Alice.



While symmetric encryption is *fast* in its execution, asymmetric encryption tends to be slower in execution as more complex algorithms are needed which come with a high computation burden.
Asymmetric encryption is not used for general-purpose encryption of large amounts of data.
Asymmetric encryption is used for:

- encryption of small amounts of data key distribution
- authentication and integrity
- NB this process is slower in execution as a result of more complex algorithms which come with a high computation burden.

Hybrid Cryptosystems can fix the issue.

Example

$D_{k_{private}}(E_{k_{public}}(M))$ does not provide user authentication since its encrypted with a public key and decrypted with a private key. This scheme is not sufficient to guarantee authentication since any entity can obtain public keys, thus anyone could've created the message. The entity that can decode the message is the one with the private key, this does ensures that the message is a secret from the sender to the entity with the private key (no one else can decrypt it).

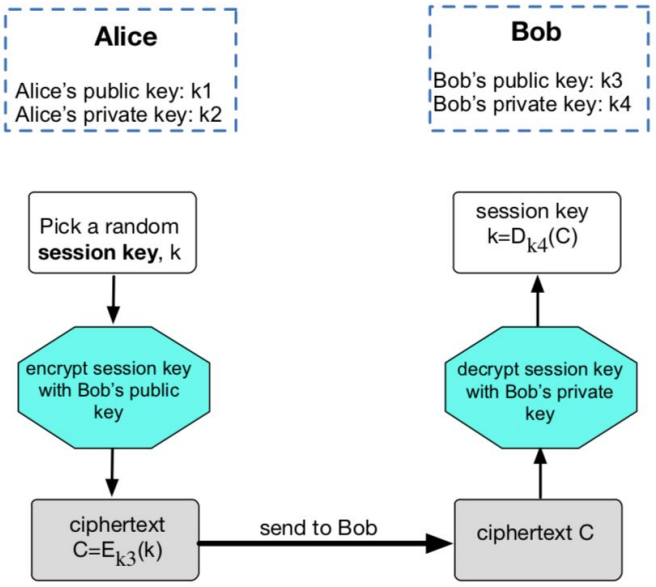
Hybrid Cryptosystems

We can combine both asymmetric and symmetric to create a more efficient solution. What we do is we create a one time only session key and we exchange it with Bob using Asymmetric cryptography. Once they have exchanged this key, they can exchange information freely since we rest assured no one know the session key.

Hybrid Cryptosystems can be used to**:**

- Session key: randomly generated key for one communication session.
- Use a public key algorithm to send the session key.
- Use a symmetric algorithm to encrypt data with the session key.

Session Key Exchange:



Authentication and Message Integrity

Means:

- Validate the creator of the content

- Validate that the content has not been modified
- The content itself does not have to be encrypted

Encrypting a message with a private key is the same as signing it. But:

- We do not want to hide the content
- What if the message is very long (asymmetric encryption is much slower than symmetric encryption)

Solution: Hash Functions

Hash Functions

Hash Function: a mathematical function which takes a message (a big number, a piece of text, or other data) and converts it into a small, fixed-length block of data which has been modified to a random set of characters which make it very difficult to go back to the original text.

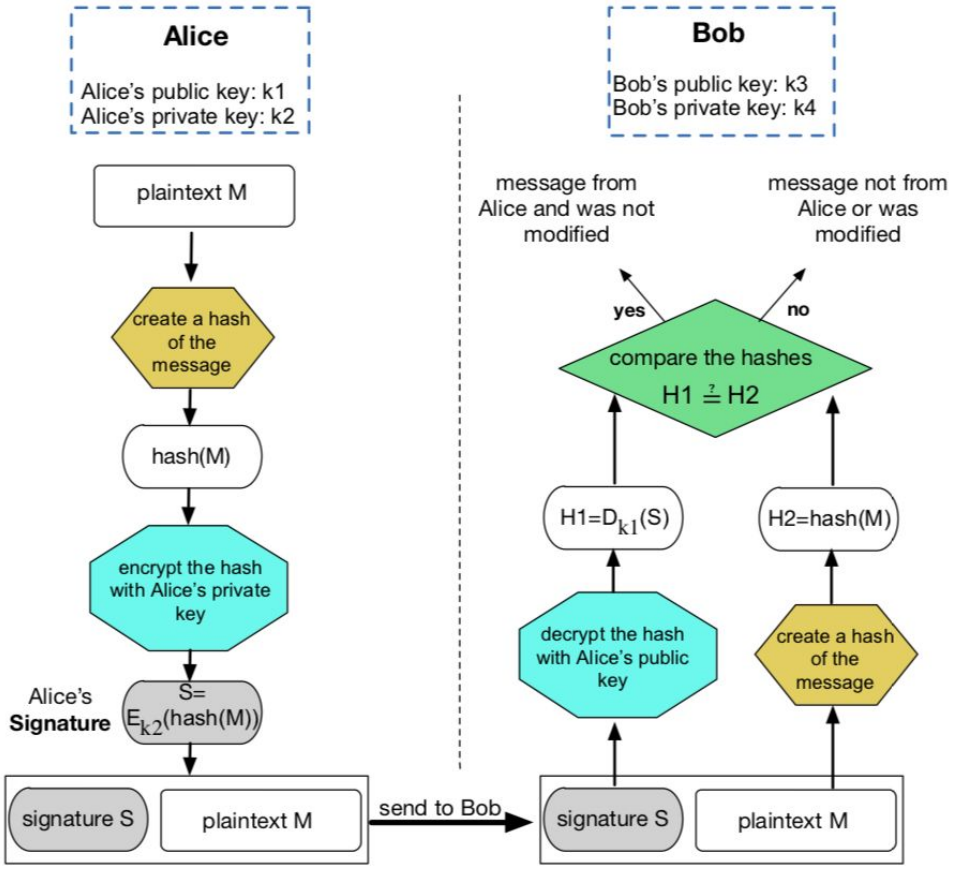
Properties

- **Efficient:** computing hash(M) should be computationally efficient.
- **One-way function:** should be difficult to compute M, given hash(M).
- **Collision resistant:** must be infeasible to find an M2≠ M1 such that hash(M2) = hash(M1)

Examples:

MD5 - which produces a 128-bit hash, SHA-1 - which outputs a 160-bit hash.

Digital Signature (asymmetric cryptography)



Key Distribution

Distribution of public keys requires care due to the man-in-the-middle attack. Attacker could replace the true public key by his public key. When we send the message back, he will be able to decrypt the information since it was encrypted with his own public key. To avoid this we may use Digital Certificates.

Digital Certificates

Given by an authoritative source, we can check that the public key is correct from the correct party.

- Public key digitally signed by a trusted party
- Trusted party receives proof of identification from entity and certifies that public key belongs to entity
- **Certificate authority** (CA) are trusted party - their public keys included with web browser distributions