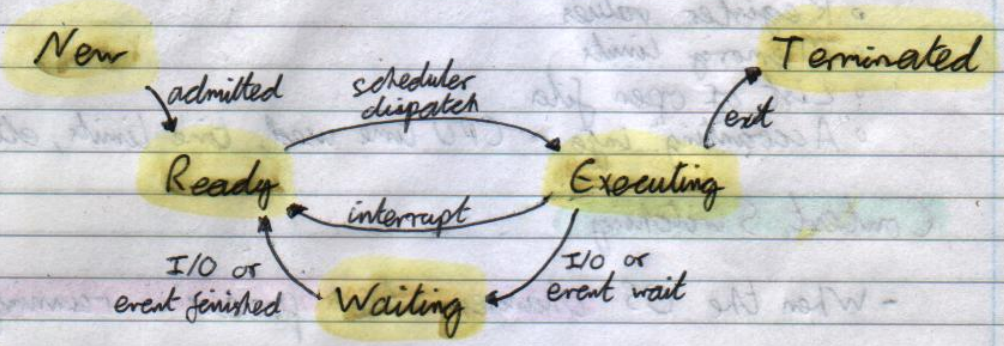
Operating Systems + Concurrency

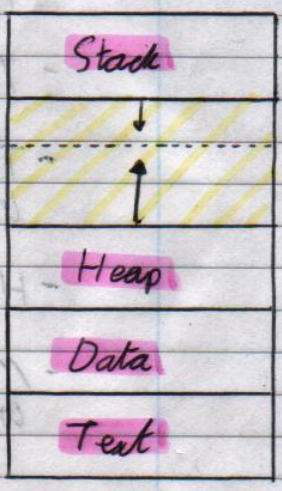
Processes

- A process is a program at some stage of execution
  - It has a program counter and variable values
- A program is an unchanging set of instructions.

Process States



Processes in Memory



- **Stack**: temporary data like function parameters, return addresses and local variables.
- **Heap**: dynamically allocated memory (new objects); the memory the process uses.
- **Data**: global variables (eg. Java statics)
- **Text**: compiled binary for the program
- NB: register values / program counter are stored in registers when a process is executing and in the PCB when it is not.

Next page →

- **Stack Overflow**
  - The stack has a fixed size - overflow is often caused by a non-returning recursion.

Process Control Block (PCB)

- The PCB is maintained by the OS to represent state and scheduling information for processes.
- It is stored separately from processes.
- It contains...
  - Process state - ready, waiting, etc...
  - Process number - a UID
  - Program counter
  - Register values
  - Memory limits
  - List of open files
  - "Accounting" info - CPU time used, time limits, etc...

Context Switching

- When the OS changes which process is running
- The "old" process has register values moved TO the PCB.
- The "new" process has register values restored FROM the PCB.
- This process is an overhead: it takes time, during which the CPU does no useful work.
- However, it allows multiple processes to "share" the CPU.
- More work gets done: CPU-bound processes can run at the same time as I/O-bound ones.
  - Most processes only require short bursts of CPU.

Process Scheduling

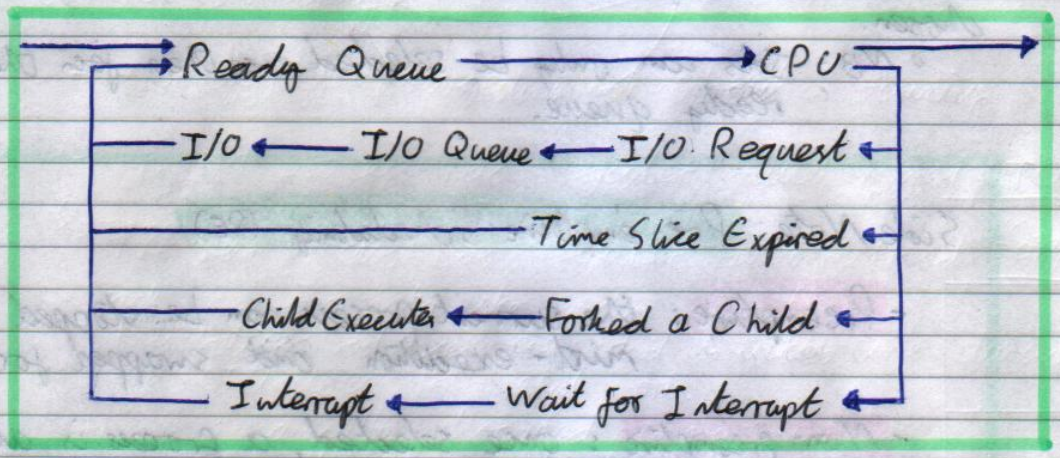
- An OS has several queues:
  - Job queue - all of the processes in the system
  - Ready queue - processes waiting for CPU time
  - Device queue - one per device; process waiting to use it
- Processes will move between queues during execution.

- There are two Phases of Scheduling:

  - Long-term scheduler:
    - Determines which processes join the ready queue.
    - Runs infrequently
    - Determines the degree of multiprogramming (how many processes can run at once.

  - Short-term scheduler
    - Pulls from the ready queue to run processes on the CPU
    - Runs frequently, so must be fast.

- Everything starts on the ready queue once added by the long-term scheduler, executes on the CPU for some time, then is moved to another queue:



- Good efficiency can be achieved by having multiple processes (I/O and CPU) at different stages in this system at once.

- Most processes do CPU work in short bursts, so we talk about "duration" as just the duration of the next burst

Scheduling Algorithm Performance Criteria

- CPU Utilization - what % of time is the CPU executing a process?
- Wait time - how long (total) does a process spend in the ready queue?
- Turnaround time - time between arrival and completion?
- Response time - time between arrival and first useful response
- Throughput - number of processes completed per time unit.

First Come, First Served (FCFS)

- Jobs are executed in the order they arrived.
- Average waiting / turnaround times are order-sensitive.
- Pros:
  - Simple algorithm
  - Minimal context switching
- Cons:
  - Average waiting time typically poor
  - " " " " is highly variable
  - CPU-bound processes can hog the CPU

Shortest Job First (SJF)

- When a scheduling decision is made, the shortest job is chosen.
  - NB - Jobs can only be selected once they join the ready queue.

> Side Note: Pre-emptive Scheduling (PE)
>
> - Preemptive: the current process can be stopped mid-execution and swapped for another
> - Non-preemptive: once selected, a process is allowed to finish its CPU burst.

- Pre-emptive Shortest Job First is also known as: Shortest Remaining Time First (SRTF)
  - If a new job arrives that is shorter than the remaining time of the current job, it is executed.
- Pros:
  - Long processes cannot hog the CPU (esp. with PE)
  - maximises throughput
  - average wait time is reduced
- Cons:
  - long processes can be starved
  - PE requires multiple context switches
  - execution time must be reliably estimated
  - overhead for maintaining a sorted queue.

Priority Scheduling

- Each job gets a **priority number**
  - Generally lower number = higher priority.
- Can be **PE** or **non-PE**.
- SRTF and SJF are special cases of this where the job time (or remaining time) is the priority.
- **Pros:**
  - high priority jobs (like user **interactive programs**) have short wait times
  - users/admins can have some **control** over scheduling
  - **deadlines** can be met.
- **Cons:**
  - risk of **starvation** for low-priority processes
    - can alleviate with an **ageing system**
  - more **context switching** if PE.
  - **overheads** for running a sorted queue.

Round Robin Scheduling (RR)

- Features a **time quantum**, **q**, usually around 10-100ms.
- When a process has ran for this time it is **pre-empted** and rejoins the empty queue.
- Performance varies according to **q**:
  - If q is too high, RR tends towards FCFS
  - If q is too low, context switching overhead becomes significant.
  - q should be longer than about **80%** of CPU bursts.
- **Pros:**
  - no starvation
  - predictable **wait time**
  - fair CPU sharing
- **Cons:**
  - poor average **response time**
  - waiting time linked to queue size, not priority
  - Lots of **context switching**
  - hard to meet **deadlines**.

Multi-Level Queue Scheduling (MLQ)

- If processes can be grouped, we can use MLQ, eg:
  - a queue for each priority level
  - two queues: foreground / background
  - etc

- The ready queue is split into multiple queues:
  - each process joins a queue
  - each queue can have a different scheduling algo.
  - an algo is needed to choose between queues.
    - often PE fixed priority.

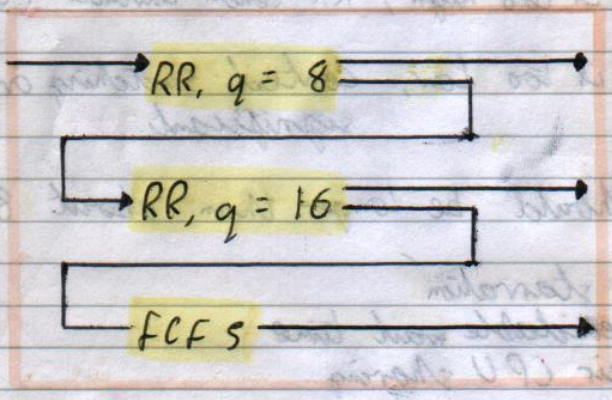
Multilevel Feedback Queue Scheduling (MLFQ)

- Like MLQ, but processes can move between queues.
  - Used to implement ageing in priority scheduling

- We also require:
  - algo to decide when to promote / demote processes
  - algo to decide which queue is joined first by a process

- Priority Scheduling w/ MLFQ:
  - one queue for each priority level
    - Higher priority => more CPU time
  - each process enters its correct priority queue
  - after some time, if not complete, it is promoted.

- Example:



Multi-Processor Scheduling

- Multiple CPUs can share the load
  - We assume all CPUs have the same capabilities
- Two variants: asymmetric and symmetric
- Asymmetric Multiprocessing (ASMP)
  - One processor makes all scheduling decisions, handles I/O processing and other system events.
  - Other CPUs execute user code only
  - The need for data sharing is reduced
- Symmetric Multiprocessing (SMP)
  - Each proc. does its own scheduling
  - May be one ready queue, or one for each processor
  - Each processor selects a process to execute from the ready queue.
  - High CPU-utilization requires load balancing:
    - Push migration: a specific process checks loads and redistributes processes as necessary
    - Pull migration: an idle processor pulls jobs from the queue of a busy processor.
  - All major OSs support SMP.

Hyperthreading

- Multiprocessing requires multiple physical CPUs
- Hyperthreading (aka. Symmetric Multithreading) uses multiple logical processors
- Logical Processors:
  - share CPU H/W: cache, bus, etc.
  - manage their own interrupt handling
- Appears as two CPUs: one for floating point (FPU) and one for arithmetic and logic (ALU).

Scheduling Algo Evaluation

- **Deterministic modelling**
  - Run a set of processes through each algo.

- **Queueing Model**
  - Use queueing theory to predict utilisation
  - Little's formula: $n = \lambda \times w$
    - n = queue length
    - λ = arrival rate (per time unit)
    - w = waiting time (time units)
  - **DO NOT USE** unless given an arrival rate

- **Problems with these approaches:**
  - unrealistic; they don't use real processes
  - hard to model complex algorithms with queueing theory
  - arrival rate / waiting time hard to predict

- **Evaluation vs. Implementation**
  - The only accurate way to evaluate is to implement
    - Lots of work
    - Users get inconsistent performance
    - Processes change a lot
    - Users will adapt to cheat.
  - The ideal is to have a system where the admin can select the scheduling algo, specific to needs.
  - An alternative is an API for users to change thread priorities.

Operations on Processes: Process Termination

- Processes can ask the OS to terminate it via **exit**
  - Exit status value from a child is recd by parent via **wait()**
  - Process resources are deallocated by the system
- Parent may terminate a child using **kill()**
- If parent is exiting:
  - Some OSs do not allow a child to continue
  - All children are terminated - cascading termination.

Threading

- A thread is a lightweight process: a basic unit of CPU utilisation.
- It has a thread ID, program counter, register set and stack.
- It shares code, data and OS resources (like files) with other threads in the same process.
- If a process has multiple threads, it can do multiple tasks at once.

Processes, Kernel Threads and User Threads

- Processes
  - Isolated with own virtual address space
  - Contains process data like file handles
  - Lots of overhead.
  - Every process has at least one kernel thread
- Kernel Threads
  - Shared virtual address space
  - Contains running state data
  - Less overhead
  - From an OS PoV, this is what scheduled for CPU time
- User Threads
  - Shared virtual address space
  - Contains running state data
  - Even less overhead
  - Kernel unaware: appears as a single threaded process
- Main difference: user threads are controlled by application

Multi-Threading

- Most applications are multi-threaded
- This is good: other threads can continue while one is blocked.
- It also allows a similar task to be done many times in parallel, such as a server responding to requests.

- Technically most of this can be done with processes, but they have a higher overhead.
- Multi-Threading Benefits:
  - Responsiveness, as mentioned earlier
  - Resource sharing (and easy interaction)
  - Efficiency: threads are cheap to create and context switch.
  - Multi-processor utilisation: a single-thread process only ever runs on one CPU core.

User Threads

- We could write code to appear as a single kernel thread that internally schedules many threads
  - Each "internal" thread is a user thread
- Alternatively, each "internal" thread can have its own kernel thread, allowing the OS to do the scheduling.

Threads in Java

- To make a thread, either extend Thread or implement Runnable!
  - Both have a "run()" method that must be implemented.
- To start a thread, either:

```
(new ThingThatExtendsThread()).start();
```

or

```
(new Thread(new ThingThatImplementsRunnable())).start();
```

- Do not call run() directly!

Wasting Time

```
try {
    Thread.sleep(30); // milliseconds
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

- Why? Prevents CPU hogging and allows another thread to take over.
- Thread.sleep() causes a blocking state, so a context switch happens.

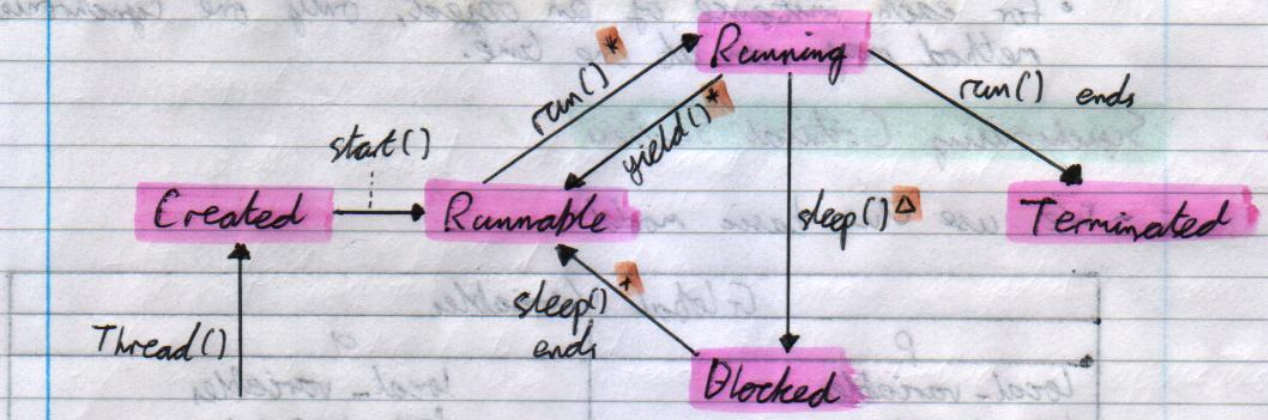
Waiting for Another Thread

`q.join();` → "Wait for q to finish before carrying on"

Killing Threads

- Asynchronous Cancellation: the thread is cancelled immediately.
- Deferred Cancellation: the thread checks periodically to see if it has been cancelled.
- A thread may be killed while it is sleeping, so the InterruptedException is thrown to allow for any cleanup.

Thread Lifecycle



* Called by scheduler

Δ Or some other blocking action

\+ Or more code to come.

Interleaving & Atomicity

- Threads share resources
- If two threads interact with the same resource, the interleaving of those threads means that conflicts may happen.
- Interleaving happens at runtime and is managed by the scheduler, so may be different for each run.
- An atomic statement is one that cannot be split/interrupted.
- Concurrency is the interleaving of atomic statements.
- We assume that each line of pseudo code is atomic (it isn't really!)

Critical Section

- A critical section must be executed atomically, WRT other related threads.
- Java provides the keyword synchronised to mark a method as a critical section.
  - For each instance of an object, only one synchronised method may run at one time.

Synchronising Critical Sections

- We use this basic model:

Global Variables	
p	q
local_variables	local_variables
loop_forever:	loop_forever:
non_critical_section	non_critical_section
preprotocol	preprotocol
critical_section	critical_section
postprotocol	postprotocol

preprotocol: check it is okay to enter CS
postprotocol: signal that CS is done
} Synchronisation mechanism

- Solution Properties

- **Mutual exclusion**: only one thread can enter a CS at one time
- **No deadlock**: if some threads are waiting to enter their CS, eventually one must succeed.
- **No starvation**: if a thread is waiting to enter its CS, it must be able to do so eventually.
- A good solution has **no interleaving** that breaks these rules.
- **Standard testing is no good** because we have to check every interleaving possible.

- Model Assumptions

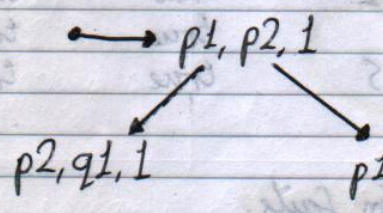
- A thread will **complete its CS**.
  - No termination
  - No infinite loop
- A process may **terminate during the non-CS**
- **Assignment is atomic**
- **CPU scheduler will not starve a process**

- First Attempt

```
                    int turn ← 1

          p                                   q
loop:                                loop:
  p1: non-CS                           q1: non-CS
  p2: await turn = 1                   q2: await turn = 2
  p3: CS                               q3: CS
  p4: turn ← 2                         q4: turn ← 1
```

- **State diagram**:



etc... The real tree is huge.

- **Mutual Exclusion**: there is no state $(p3, q3, \_)$, so this holds!
- **No Deadlock**: deadlock can only happen at $(p2, p2, \_)$, but at those states either turn=1 or turn=2, so eventually a thread proceeds.
- **No Starvation**: Suppose we have: $(p1, q1, 1) \to (p2, q1, 1) \to (p3, q1, 1) \to (p4, q1, 1) \to (p1, q1, 2)$

  then q quits during its non-CS. p will never execute now!

- Second Attempt

wantp ← false wantq ← false

p	q
loop:	loop:
non-CS	non-CS
p1: await wantq = false	q1: await wantp = false
p2: wantp ← true	q2: wantq ← true
CS	CS
p3: wantp ← false	q3: wantq ← false

- Possible Interleaving:

p	q	wantp	wantq
p1		false	false
	q1	false	false
p2		true	false
	q2	true	true
CS		true	true
	CS	true	true

Mutual exclusion fails.

- Third Attempt

wantp ← false wantq ← false

p	q
loop:	loop:
p1: non-CS	q1: non-CS
p2: wantp ← true	q2: wantq ← true
p3: await wantq = false	q3: await wantp = false
p4: CS	q4: CS
p5: wantp ← false	q5: wantq ← false

- Mutual Exclusion (Informal Proof)
  - We must show that (p4, q4, _, _) cannot occur.
  - To get that, one thread must be at 4 and the other must move 3 → 4.
    - Threads are symmetric, so it doesn't matter which one.
  - Lets say p is at p4. What is the value of wantp?
    - q cannot modify wantp.
    - If p is at p4, the last change was p2: wantp ← true
  - For q to move q3 → q4, we need wantp = false
    - But p4 only allows wantp = true.
  - ∴ (p4, q4, _, _) is not possible!
- No Deadlock

p	q	wantp	wantq
p1		false	false
	q1	false	false
p2		true	false
	q2	true	true
p3		true	true
	q3	true	true

Deadlock!

- Peterson's Algorithm

wantp ← false, wantq ← false, last ← 1

p	q
p1: non-CS	q1: non-CS
p2: wantp ← true	q2: wantq ← true
p3: last ← 1	q3: last ← 2
p4: await wantq = false or last = 2	q4: await wantp = false or last = 1
p5: CS	q5: CS
p6: wantp ← false	q6: wantq ← false

- **Mutual Exclusion** (Informal Proof)
  - We must show that (p5, q5, –, –, –) cannot occur.
  - We must have one process at 5 and one moving 4 → 5 for that to happen.
  - Assume p is at p5. wantp = true (see p2)
  - We know (p5, q4, true, —, 1 or 2)
  - We want to show that q4 → q5 cannot happen.
  - wantp = true, so q4 → q5 can only happen if last = 1, which can only be set at p3.
  - When p4 → p5 happened, either:
    - wantq = false
      - so q was at q1 or q2
      - and q must exec. q3 before q4, setting last = 2
      - p is at p5 so cannot set last = 1 at p3
      - so q waits at q4
    - last = 2
      - p is at p5 so cannot set last = 1 at p3
      - so q waits at q4

- No Deadlock (Informal Proof)
  - Both must be stuck at line 4.
  - wantp and wantq are true, and don't change.
  - Because last must =1 or =2, the program cannot get stuck here.
- No Starvation (Informal Proof)
  - We must show that if p is at p4 it will eventually reach p5.
  - If p is at p4, it can proceed unless wantq = true and last = 1.
  - Where could q be?
    - If wantq is true, q can be at q3, q4 or q5
    - If q is at q3
      - it will execute last = 2
      - it will remain at q4 until p executes
    - If q is at q4
      - it will move to q5 because last = 1
      - it will proceed to q6
      - it will proceed to q1
        - it may quit here; p can execute because wantq = false
      - it will proceed to q2, setting wantq = true
      - it will proceed to q3, setting last = 2
      - it will remain at q4 until p executes.
    - If q is at q5

Properties as Temporal Logic

- Mutual exclusion: $\Box\neg(p4 \wedge p5)$
- No Deadlock: $\Box((p4 \vee q4) \rightarrow \Diamond(p5 \vee q5))$
- No Starvation: $\Box(p4 \rightarrow \Diamond p5)$

Concurrency with n Processes

Bakery Algorithm

- Based on a service in a bakery or other shop.
- Each process takes a number; lowest number goes first.
- For N processes:

```
int array [1...N] ← [0,...,0]

loop-forever:
  p1: non-CS
  p2: number[i] ← max(number) + 1 *
  p3: for all other processes j:
        await number[j] = 0 or number[i] << number[j]
  p4: CS
  p5: number[i] = 0
```

* This is not atomic! This can be fixed, but the algo is more complex.

Good: ◦ each variable is written to by only one process
◦ satisfies all three properties

Bad: ◦ unbounded ticket numbers
◦ p3 queries every other process.

Ugly: ◦ too inefficient to actually use.

Synchronisation Hardware

- So far, we have assumed an atomic assignment operator.
- Now consider a more general idea:

"A thread must obtain a lock to enter its CS"

- Hardware can provide these locks as primitives to make synchronisation easier.

Side Note: in a single processor system we could solve the CS problem by disabling interrupts during a CS, but in a multi-proc system this message has to be passed to every core, which is slow.

Test and Set

```
boolean lock ← false

loop-forever:
p1: non-CS
p2: await (!testAndSet(lock))
p3: CS
p4: lock ← false
```

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

- Implemented as a single processor instruction
- Does no harm on set if already true
- False → no one is in CS

Swap

```
boolean lock ← false

        p
boolean key ← true

loop-forever:
p1: non-CS
p2: key ← true
p3: while (key = true):
        swap(lock, key)
p4: CS
p5: lock ← false
```

```
boolean swap(boolean a, boolean b) {
  boolean tmp = a.getValue();
  a.setValue(b.getValue());
  b.setValue(tmp.getValue());
}
```

Implemented as a single processor instruction.

Both of these could lead to starvation, but it is unlikely.

They solve deadlock and mutual exclusion.

Semaphores

- Programmers usually prefer high-level, software mechanisms for concurrency control:
  - More abstract
  - Less complex
  - Less error prone.
- Semaphores are provided by most OSs.

Busy-Wait Semaphore

- Contains:
  - An integer variable, v (always $v \geq 0$)
  - A set of blocked processes (initially empty)
- Has two methods:

```
wait (Semaphore s) {
    while (s.v <= 0)* {
        // do nothing
    }
    s.v = s.v - 1;
}
```

```
signal (Semaphore s) {
    s.v = s.v + 1;
}
```

* aka. a "spin-lock"

Blocked-Set Semaphore

- Contains the same as above
- Has two methods:

```
wait (Semaphore s) {
    s.v = s.v - 1;
    if (s.v < 0) {
        s.blocked = s.blocked ∪ this;
        this.state = blocked;
    }
}
```

Union, ie. add this to s.blocked.

```
signal (Semaphore s) {
    s.v = s.v + 1;
    remove one q. from s.blocked;
    q.state = runnable;
}
```

Semaphore Useage

Semaphore s ← (1, Q)

p	q
p1: non-CS	q1: non-CS
p2: wait(s)	q2: wait(s)
p3: CS	q3: CS
p4: signal(s)	q4: signal(s)

- Initially, s.v = 1
- Before a thread enters CS:
  - It calls wait; s.v is decremented; 0 means some thread in CS
  - If another thread calls wait, then s.v = 0 so it is blocked.
- When a process leaves CS:
  - signal(s) adds one to s.v, so another process can enter
  - if there are waiting processes, one is unblocked.

Properties e Notes

- Properties:
  - In busy-wait, s.v ≥ 0 always.
  - s.v = init + #signal(s) − #wait(s)
  - #CS + s.v = 1
  - #CS = #wait(s) − #signal(s)
  - Mutual Exclusion ✓
  - Freedom from Deadlock ✓
  - Starvation: Technically possible, but requires a specific interleaving so unlikely. Can be fixed with a busy FIFO queue, instead of a blocked set.
- Creating a semaphore with a value higher than 1 allows a control such as "max of 3 CSs at once".
  - Used for server connections.

The Producer/Consumer Problem.

- For **async** producer/consumer communication we use a **buffer**.
- This can create two issues:
  - **Buffer is empty**, so consumer cannot take data
  - **Buffer is full**, so producer cannot add data

Infinite Buffer Solution

Semaphore notEmpty ← (0, ∅), Buffer<d>[buffer] ← empty

producer	consumer
p1: d ← produce()	c1: wait(notEmpty)
p2: append(d, buffer)	c2: d ← take(buffer)
p3: signal(notEmpty)	c3: consume(d)

- We only need to **control removal**
- Semaphore starts at **zero** because we want the consumer to wait until something has been added
- **Semaphore = 0 ⇒ buffer is empty.**

Finite Buffer Solution

Semaphore notEmpty ← (0, ∅), Semaphore notFull ← (N, ∅),
size N queue <d> buffer ← ∅

producer	consumer
p1: d ← produce()	c1: wait(notEmpty)
p2: wait(notFull)	c2: d ← take(buffer)
p3: append(d, buffer)	c3: signal(notFull)
p4: signal(notEmpty)	c4: consume(d)

- **notFull** is initialised to **N**, so **N** things can be added before it starts blocking.
- This technique is called **split semaphores**.

Semaphores in Java

Constract: `new Semaphore (int permits);` → fair = false.

`new Semaphore (int permits, boolean fair);`

Wait: `s. acquire();`

Signal: `s. release();`

Monitor + Concurrency in Java

- Semaphores are great (v. user-faiendly), but prone to errors.
- Semaphores must be implemented perfectly and carefully, otherwise big errors can occur.
- Solution: Monitors
  - Similar to a class, a monitor encapsulates a resource and provides access only via public methods
  - Responsibility for mutual exclusion is contained within the monitor for a resource.
    - Less scope for bugs
    - Less duplicated code
  - Each resource can be handled by a separate monitor.

Implementing Monitors w/ Semaphores

```
class Monitor {

    Semaphore s = new Semaphore (1);

    method m1() {
        s.wait();
        // actual method code
        s.signal();
    }

    // ...
}
```

Simulating Semaphores w/ Monitors

```
monitor Semaphore {

    int s = k;

    method semaWait()
        while (s == 0)
            wait();
        s = s - 1;

    method semaSignal()
        s = s + 1
        notifyAll();
}
```

p	q
Loop-forever: p0: non-CS p1: Semaphore.semaWait() p2: CS p3: Semaphore.semaSignal()	Loop-forever: q0: non-CS q1: Semaphore.semaWait() q2: CS q3: Semaphore.semaSignal()

Monitor wait() and notifyAll()

- Maintain a list of waiting blocked threads
- wait():
  - adds the current thread to the blocked list
  - sets its state to blocked
  - releases the monitor lock
- notifyAll():
  - sets all blocked state to ready → the scheduler will select one - the rest will re-block.
  - empties the blocked list.
- These can only be called inside the monitor.

Producer/Consumer Problem w/ Monitors

```
monitor Buffer

  int[5] buffer;
  int spaceUsed = 0;

  method addItem(i)
    while(spaceUsed == 5)
      wait();
    buffer[spaceUsed] = i;
    ++spaceUsed;
    notifyAll();

  method removeItem()
    while(spaceUsed == 0)
      wait();
    i = buffer[spaceUsed];
    --spaceUsed;
    notifyAll();
    return i;
```

P

```
loop-forever:
  p0: i ← produce();
  p1: Buffer.addItem(i);
```

Q

```
loop-forever:
  q0: i ← Buffer.removeItem();
  q1: consume(i);
```

- It would be better if we could notify threads selectively.
  - ie. only wake producers when the buffer is empty.
- Classical monitors allow this, but Java does not.
  - wait() takes a condition that the thread is waiting for
  - notify/notifyAll() takes a condition to signal

Queue or Set in Monitors

- Only worthwhile if we'll use notify() to wake the head of the queue.
- This can avoid starvation caused by allowing the scheduler to select a thread after notifyAll();
- Need to be careful though: this only works if all threads are waiting for exactly the same condition.

Can Monitors Deadlock?

- Yes.
- But the deadlocking code is all in one place - easier to debug.

Notifying Before Returning

- We can't always make `notifyAll();` the last command
  - What if we have to return?
- What happens if the thread is in the monitor and notifies other threads?

Thread States WRT Monitors

- E = waiting to enter the monitor
- W = waiting to be notified
- N = executing notify (currently holds the lock)
- In Java, priority is $E = W < N$
- Traditionally, it is $E < N < W$
- $E < N < W$ Implications
  - If a single process is woken then it knows the condition it was waiting for is true because it has just been signalled: no thread has executed between it and the one that signalled it (which has now stopped) so it can continue without rechecking the condition.
  - This is the immediate resumption requirement.
- $E = W < N$ Implications
  - Good if we want to return before we notify: the signalling process can continue until it has left the monitor.
  - Now there is no guarantee that the signalling process didn't change the wait condition before it stopped
  - Further, `notifyAll()` could have caused a different process to wake and break the condition.

- In this case, each thread must re-check the condition before continuing
  - Achieved by using while(! condition) - it cannot exit until the condition has been checked and seen to be true.

Monitors in Java

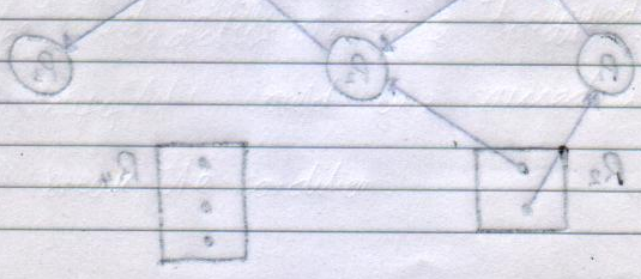
- Don't really exist.
- All Objects can be used as a monitor
- Not all methods must be mutually exclusive, but those that require ME are marked with synchronised.
- All Objects implement wait(), notify() and notifyAll()
- Refer to start of notes for Java thread lifecycle.

Deadlock

- Deadlock occurs during concurrent programming when, for some reason, no process is able to continue.
- It occurs on interaction between threads/processes, so the cause may not be in one piece of the code.

Conditions for Deadlock

- System Model:
  - A number of processes.
  - A number of resources
    - Disks
    - Semaphores
    - etc.
  - Each process requests resources as it needs them
  - " " " releases " " " finishes with them
  - Some processes use multiple resources
  - Some resources have multiple instances



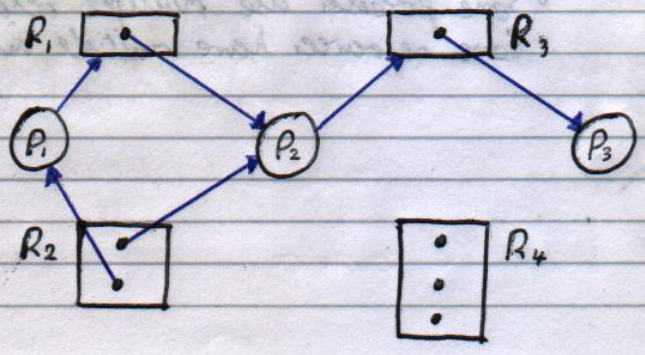
- Coffman Conditions for Deadlock:

- **Mutual exclusion** - only one process at a time can use an instance of a resource
- **Hold and Wait** - a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No Preemption** - a resource is only released voluntarily by the process holding it, after it has completed its task.
- **Circular Wait** - there exists a set of processes $\{P_0 \dots P_n\}$ such that:
  - P_0 is waiting for a resource held by P_1
  - P_1 " " " " " " " P_2
  - P_2 " " " " " " " P_3
  - . . . . .
  - P_{n-1} " " " " " " " P_n
  - P_n " " " " " " " P_0

Resource Allocation Graphs

- **Vertices:** processes (○) and resources (□)
- **Request Edge:** $P_i \rightarrow R_i$ = P_i has requested an instance of R_i
- **Assignment Edge:** $R_i \bullet \rightarrow P_i$ = P_i holds an instance of R_i
- **Example:**

$P = \{P_1, P_2, P_3\}$
$R = \{R_1, R_2, R_3, R_4\}$ Num. of instances: $R_1: 1$, $R_2: 2$, $R_3: 1$, $R_4: 3$
$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$



- RAG Cycles:
  - No cycles = no deadlock
  - If cycles exist...
    - If each resource has only one instance, we have deadlock
    - If some resources have multiple instances, we may have deadlock.

Approaches to Deadlock

- Prevention: ensure the system cannot enter a deadlocked state
- Cure: allow deadlock to happen and then recover
- Ignore it!
  - Dealing with it is expensive, and it's pretty rare anyway.

Preventing Deadlock

- We must break one of the Coffman conditions:
- Mutual exclusion: not easy, but we can reduce the risk of a problem by enforcing ME when only absolutely necessary.
- Hold and wait: guarantee that whenever a process requests a resource it does not hold any others
  - a process must request and hold all resources before it can begin
  - or a process can only request a resource when it has none
  - causes low resource utilisation and can cause starvation.
- Pre-emption:
  - if a process requests a resource that cannot be allocated immediately it must immediately release any resources already held
  - a list of all resources the process is waiting for is kept
  - the process is only restarted when all resources it needs are available.

- **Circular Wait** - Impose a total ordering on resources and make processes request them in increasing order.

- To avoid a deadlock we need extra information about processes:
  - eg. max number of resources of each type that will be requested
  - Unrealistic to assume will know this.

- We can use the resource allocation state of a system to ensure that we will never allow a circular wait
  - With single-instance resources we can do this with a RAG.
  - With multi-instance resources we use Banker's algorithm.

- **Resource Allocation State:** number of available/allocated resources and max number of resources each process may need.

Safe States

- Each time a process requests a resource, the system must decide if that allocation leaves the system in a safe state.

- A system is in a safe state if there exists a sequence of all processes $\langle P_0, P_1, \dots P_n \rangle$ such that for all P_i, the resources that P_i can request can be satisfied by currently available resources and the resources held by all P_j with $j < i$.

- Therefore...
  - A process can never be holding a resource and waiting for a process with a higher number to release one.
  - **Circular wait cannot occur:** P_j can hold and wait for P_i, but P_i cannot hold and wait for P_j. $(j > i)$

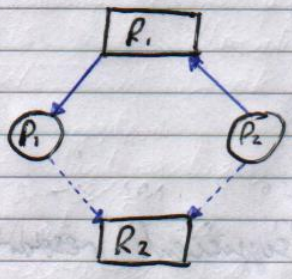
- **Safe state** = cannot deadlock

- **Unsafe state** = might deadlock

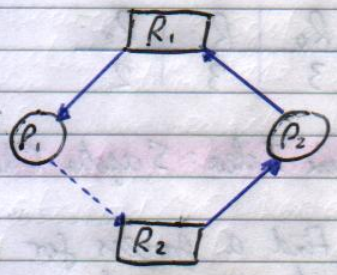
Safe States - RAG Method

- Works only with single-instance resources
- Add a new edge type: a claim edge. (dashed line)
- A claim edge from $P_i \rightarrow R_i$ shows that P_i can request R_i. This is specified apriori.
- Claim edge changes to assignment edge and back when R_i is allocated / released.
- All claim edges must be defined apriori.
- If a process P_i requests resource R_j, the request can only be granted if converting the $P_i \rightarrow R_j$ claim edge to an assignment edge does not create a cycle.

Safe:



Unsafe:



Safe States - Banker's Algorithm

- Works with multi-instance resources.
- Assumption: resources are eventually released.
- Processes may have to wait after they make a request.
- Not cheap: $O(mn^2)$ for m resources and n processes.

- Input Data Structures

  - For n processes and m resources

  - Allocation: n x m matrix
    Allocation[i,j] = k means P_i has k instances of R_j

	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

  - Max: n x m matrix
    Max[i,j] = k means P_i may request up to k instances of R_j

	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

  - Available: m-length vector
    Available[i] = k means there are k instances of R_i available now

R_0	R_1	R_2
3	3	2

- Basic Idea: Safety Check

  1. Find a process for which there are sufficient resources to meet its max need.
  2. Simulate executing the process by releasing all of its resources.
  3. Go back to step 1. Collect $200.
  4. If all processes have completed, the system is safe.
  5. If any processes remain, the system is unsafe.

- Algorithmic Safety Check

  1. Let work and finish be arrays of m (→ Resources) and n (→ Processes) respectively:
     - work = available
     - finish[i] = false for i = 0, 1, ..., n-1
  2. Find any i such that:
     - finish[i] = false and max[i] - allocation[i] ≤ work
     - If no i exists, go to step 4.

3. Work = Work + allocation[i]
finish[i] = true
Go back to step 2.

4. If all entries in finish are true, the system is safe.
Otherwise, it is unsafe.

- How to Use This?

- When a process P_i requests a resource R_j:
  - Update allocated as if the request was granted
  - Run the algo.
  - If the system is safe, allow the request. Otherwise, P_i must wait.

Deadlock Detection

- Instead of prevention, we can allow it to happen and then recover.
- With single-instance resources we can look for cycles in the RAG.
- With multi-instance resources we can use a modified Banker's algo.
  - Instead of using the max number of resources a process can request, use the number it is currently requesting.

- Banker's Algo for Deadlock Detection

1. Initialise as before, but if P_i has no resources allocated then finish[i] = true.

2. Find any i such that:
   - finish[i] = false and request[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, the system is safe.
If finish[i] = false then process P_i is deadlocked.

- In step 2 and 3 we assume that if Pi has enough resources to continue now then it will complete. It may actually request more resources a little later, but that's okay: we will detect a deadlock then, if it happens.

- When to Call This?

  - When, and how often, depends on:
    - How often a deadlock is likely to occur?
    - How many process will need to be rolled back?
      - A: one for each disjoint cycle.

  - If not frequent enough then many cycles could be created and we wouldn't know which was the "cause".

  - If too frequent then it is a waste of CPU time.

  - Assuming we detect a deadlock, how do we recover?

Deadlock Recovery

- Process termination

  - Abort all deadlocked processes

  - Abort one at a time until
    - What order to abort in?
    - Process priority:
      - Time elapsed / remaining
      - Resources used
      - Resources required
      - Process interactive or batch?

- Resource Pre-emption

  - Select a process and take resources away to break the deadlock

  - Rollback to a safe state and restart the process(es).

  - Select a victim that minimises cost.

  - Include number of rollbacks in cost factor to prevent starvation by always selecting the same process.