Concurrencia y Distribución

2017/2018

Tercero, Grado

Dr. Arno Formella

Departamento de Informática
Escola Superior de Enxeñaría Informática
Universidade de Vigo

17/18

Profesor: Arno FORMELLA
Web: http://formella.webs.uvigo.es
Correo: formella@uvigo.es
Tutorías Ma: 09:30–13:30 y 16:30–18:30

usamos: FAITIC/TEMA

profesorado (prácticas)

Profesora: Anália Lourenço García
Correo: analia@uvigo.es
Tutorías Lu: 16:00–19:00 y Vi: 10:00–13:00

Profesora: Lorena Otero Cerdeira

Profesor: David Olivieri Cecchi
Correo: olivieri@uvigo.es
Tutorías Lu: 16:00–18:00 y Vi: 10:00–14:00

tutorías

- Cambios puntuales de tutorías via aviso web
  (yo en mi página principal, y/o aviso via FaiTIC)
- Idiomas: Galego, Castellano, English, Deutsch
- Las transparencias serán en castellano con pinceladas en inglés.

horas de dedicación (planificación según guía)

	pres.	no-pres.	suma
Actividades introductorias	0.50	–	0.50
Sesión magistral	18.00	9.00	27.00
Estudios/actividades previos	–	17.00	17.00
Prácticas en aulas de informática	26.00	26.00	52.00
Resolución de problemas y/o exercicios	1.50	19.50	21.00
Presentacións/exposicións	–	1.75	1.75
Tutoría en grupo	1.25	1.25	2.50
Pruebas de respuesta corta	1.00	–	1.00
Pruebas de respuesta larga	2.00	–	2.00
Informes/memorias de prácticas	–	12.00	12.00
Probas prácticas	1.00	–	1.00
Resolución de problemas y/o exercicios	–	12.00	12.00
Otras	0.25	–	0.25
Suma	51.50	98.50	150.00

Teoría:	los viernes (13 sesiones), 11:00-12:30 horas, Aula 2.2
Prácticas:	5 grupos, Lab. 31B, 11 sesiones a partir del lunes 05/02/2018
	3 sem. Anália, 3 sem. Lorena, 5 sem. David

Hora	LUNES			MARTES			MIÉRCOLES			JUEVES			VIERNES		
09:00–11:00	DGP_2 [SO6]	SI_4 [L37]	CDI_6 [L31B]	CDI_1 [L31B]	TALF_3 [L38]	SI_5 [L37]	SI_2 [L37]	HAE_4 [TecElect]	TALF_6 [SO6]	HAE_1 [TecElect]	CDI_3 [L31B]	DGP_5 [SO6]	TALF_2 [L38]	DGP_4 [SO6]	HAE_6 [TecElect]
11:00–12:30	SI [2.2] Prof. Juan Carlos González Moreno			TALF [2.2] Prof. Manuel Vilares Ferro			HAE [2.2] Prof.Carlos Castro Miguens			DGP [2.2] Prof. Celso Campos Bastos			CDI		
12:30–14:30	DGP_1 [SO6]	SI_3 [L37]	CDI_5 [L31B]	CDI_2 [L31B]	TALF_4 [L38]	SI_6 [L37]	SI_1 [L37]	HAE_3 [TecElect]	TALF_5 [SO6]	HAE_2 [TecElect]	CDI_4 [L31B]	DGP_6 [SO6]	TALF_1 [L38]	DGP_3 [SO6]	HAE_5 [TecElect]

Están organizados los grupos de prácticas.

horas presenciales

Luns	Martes	Mércores	Xoves	Venres	Sábado	Domingo
					20/ene	21/ene
22/ene	23/ene	24/ene	25/ene	26/ene	27/ene	28/ene
29/ene	30/ene	31/ene	01/feb	02/feb	03/feb	04/feb
05/feb	06/feb	07/feb	08/feb	09/feb	10/feb	11/feb
12/feb	13/feb	14/feb	15/feb	16/feb	17/feb	18/feb
19/feb	20/feb	21/feb	22/feb	23/feb	24/feb	25/feb
26/feb	27/feb	28/feb	01/mar	02/mar	03/mar	04/mar
05/mar	06/mar	07/mar	08/mar	09/mar	10/mar	11/mar
12/mar	13/mar	14/mar	15/mar	16/mar	17/mar	18/mar
19/mar	20/mar	21/mar	22/mar	23/mar	24/mar	25/mar
26/mar	27/mar	28/mar	29/mar	30/mar	31/mar	01/abr
02/abr	03/abr	04/abr	05/abr	06/abr	07/abr	08/abr
09/abr	10/abr	11/abr	12/abr	13/abr	14/abr	15/abr
16/abr	17/abr	18/abr	19/abr	20/abr	21/abr	22/abr
23/abr	24/abr	25/abr	26/abr	27/abr	28/abr	29/abr
30/abr	01/may	02/may	03/may	04/may	05/may	06/may
07/may	08/may	09/may	10/may	11/may	12/may	13/may

horas en aula presenciales

- 02.02. actividad introductoria
- 12 sesiones magistrales + pruebas de respuesta corta
- 18.05. prueba final (10:00-14:00, 2 horas)
- 29.06. prueba terminal (09:00-12:00)
- 11 sesiones prácticas con entregas de trabajos en clase y entregas al final de la semana

horas de trabajo (este curso)

Actividad	pres.		no-pres.	horas	guía
Actividades introductorias	0.5	T	–	0.5	(0.5)
Sesión magistral	18.0	T	9.0	27.0	(27.0)
Estudios/actividades previos	–		6.5	6.5	(17.0)
Prácticas en aulas de informática	22.0	P	22.0	44.0	(52.0)
Resolución de problemas/exercicios	1.5	T	19.5	21.0	(21.0)
Presentacións/exposicións	–	P	1.0	1.0	(1.7)
Tutoría en grupo	1.0		1.0	2.0	(2.5)
Pruebas de respuesta corta	1.0	T	–	1.0	(1.0)
Pruebas de respuesta larga	2.0		–	2.0	(2.0)
Informes/memorias de prácticas	–		11.0	11.0	(12.0)
Probas prácticas	1.0	P		1.0	(1.0)
Resolución de problemas y/o exercicios	–		11.0	11.0	(12.0)
Otras	–		5.0	5.0	(0.3)
Suma	47.0		86.0	133.0	
segunda oportunidad			+17.0	+17.0	150.0

horas del profesor (yo, aproximado, optimista)

960	horas anuales de docencia (60 %)
	$\cdot 22.5/240$ fracción para CDI
90	-19.5 horas presenciales teoría
70.5	-19.5 horas preparación clases teoría
51	-1 horas preparación/gestión clases prácticas
50	$-106/4-4$ horas y corrección exámenes teoría
19.5	$/106/16$ media por estudiante por semana
0.01149	horas
41.4	segundos por estudiante por semana

prerrequisitos

- matemáticas
- algoritmos y estructuras de datos
- todo sobre programación
- arquitectura de computadoras
- redes
- sistemas operativos
- lenguajes de programación (Java, C++)

ORGANIZACIÓN TEMPORAL DEL TÍTULO DE GRADUADO/A EN INGENIERÍA INFORMÁTICA

SEMESTRES							
1/1S	**1/2S**	**2/1S**	**2/2S**	**3/1S**	**3/2S**	**4/1S**	**4/2S**
DERECHO:: FUNDAMENTOS ÉTICOS Y JURÍDICOS DE LAS TIC (CFB; 6 ECTS)	EMPRESA:: ADMINISTRACIÓN DE LA TECNOLOGÍA Y LA EMPRESA (CFB; 6 ECTS)	INGENIERÍA DE SOFTWARE I (OB; 6 ECTS)	INGENIERÍA DE SOFTWARE II (OB; 6 ECTS)	INTERFACES DE USUARIO (OB; 6 ECTS)	DIRECCIÓN Y GESTIÓN DE PROYECTOS (OB; 6 ECTS)	OPTATIVA (6 ECTS)	OPTATIVA (6 ECTS)
MATEMÁTICAS:: FUNDAMENTOS MATEMÁTICOS PARA LA INFORMÁTICA (CFB; 6 ECTS)	MATEMÁTICAS:: ÁLGEBRA LINEAL (CFB; 6 ECTS)	MATEMÁTICAS:: ESTADÍSTICA (CFB; 6 ECTS)	BASES DE DATOS I (OB; 6 ECTS)	BASES DE DATOS II (OB; 6 ECTS)	SISTEMAS INTELIGENTES (OB; 6 ECTS)	OPTATIVA (6 ECTS)	OPTATIVA (6 ECTS)
MATEMÁTICAS:: ANÁLISIS MATEMÁTICO (CFB; 6 ECTS)	INFORMÁTICA:: ALGORITMOS Y ESTRUCTURAS DE DATOS I (CFB; 6 ECTS)	ALGORITMOS Y ESTRUCTURAS DE DATOS II (OB; 6 ECTS)	REDES DE COMPUTADORAS I (OB; 6 ECTS)	REDES DE COMPUTADORAS II (OB; 6 ECTS)	CONCURRENCIA Y DISTRIBUCIÓN (OB; 6 ECTS)	SEGURIDAD EN SISTEMAS INFORMÁTICOS (OB; 6 ECTS)	TÉCNICAS DE COMUNICACIÓN Y LIDERAZGO (OB; 6 ECTS)
INFORMÁTICA:: PROGRAMACIÓN I (CFB; 6 ECTS)	PROGRAMACIÓN II (OB; 6 ECTS)	SISTEMAS OPERATIVOS I (OB; 6 ECTS)	SISTEMAS OPERATIVOS II (OB; 6 ECTS)	LÓGICA PARA LA COMPUTACIÓN (OB; 6 ECTS)	TEORÍA DE AUTÓMATAS Y LENGUAJES FORMALES (OB; 6 ECTS)	APRENDIZAJE BASADO EN PROYECTOS (OB; 6 ECTS)	**TRABAJO FIN DE GRADO (OB; 12 ECTS)**
FÍSICA:: SISTEMAS DIGITALES (CFB; 6 ECTS)	INFORMÁTICA:: ARQUITECTURA DE COMPUTADORAS I (CFB; 6 ECTS)	ARQUITECTURA DE COMPUTADORAS II (OB; 6 ECTS)	ARQUITECTURAS PARALELAS (OB; 6 ECTS)	CENTROS DE DATOS (OB; 6 ECTS)	HARDWARE DE APLICACIÓN ESPECÍFICA (OB; 6 ECTS)	OPTATIVA (6 ECTS)	
30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS	30 ECTS

evaluación continua

- (P1) preguntas cortas

  [A4 A5 A7 A8 A12 A13 A14 A15 A16 A19 A20 A22 A25 A26 A27 A28 A30 A31 A33 A35 A36 B1 B2 B5 B6 B8 B10]
- (P2) preguntas largas (hay que aprobar ≥ 4)

  [A4 A5 A7 A8 A12 A13 A14 A15 A16 A19 A20 A22 A25 A26 A27 A28 A30 A31 A33 A35 A36 B1 B2 B5 B6 B8 B10]
- (P3) informes

  [A4 A5 A7 A8 A12 A13 A14 A15 A16 A19 A20 A22 A25 A26 A27 A28 A30 A31 A33 A35 A36 B1 B2 B3 B5 B6 B8 B10]
- (P4) programación (hay que aprobar ≥ 4)

  [A4 A5 A7 A8 A12 A13 A14 A15 A16 A19 A20 A22 A25 A26 A27 A28 A30 A31 A33 A35 A36 B1 B2 B5 B6 B8 B10]
- (P5) análisis

  [A4 A5 A7 A8 A12 A13 A14 A15 A16 A19 A20 A22 A25 A26 A27 A28 A30 A31 A33 A35 A36 B1 B2 B5 B6 B8 B10]
- (P6) presentaciones

  [A4 A5 A7 A8 A12 A13 A14 A15 A16 A19 A20 A22 A25 A26 A27 A28 A30 A31 A33 A35 A36 B1 B2 B3 B5 B6 B8 B10]

- $\min(10, 0.1P_1 + 0.4P_2 + 0.25P_3 + 0.25P_4 + 0.05P_5 + 0.05P_6) \geq 5$
- pero $P_2 \geq 4$ y $P_4 \geq 4$

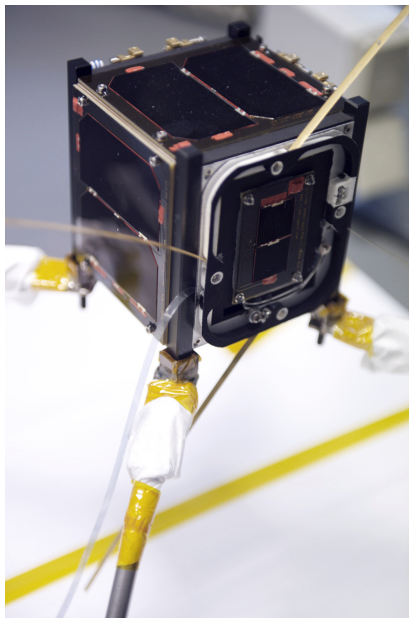
evaluación no-asistentes

- examen (18.05.) de 3.5 horas (entre 10:00-14:00) que cubre todo el contenido de la asignatura (teoría y prácticas)
- y/o examen (29.06.) de 3.0 horas (entre 9:00-12:00) que cubre todo el contenido de la asignatura (teoría y prácticas)
- un alumno o bien se autodeclara no-asistente o lo muestra por no asistir a por lo menos 80 % de las actividades presenciales (como mucho se puede faltar a 8.5 horas de las 19.5+22=41.5 horas de presencialidad principal)
- control mediante firmas en clase y/o entregas en FaiTIC

vos y nos

¿Quién soy yo?

- página web de docencia
  http://formella.webs.uvigo.es
- página web de investigación
  http://lia.ei.uvigo.es



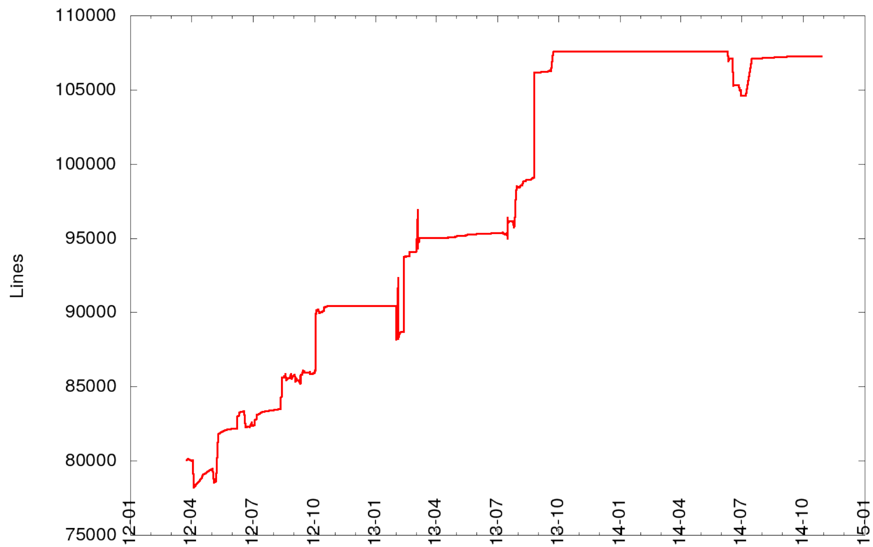
3 Satélites:

XaTcobeo, 14/02/2012

HumSAT, 21/11/2013

Serpens, 20/08–17/09/2015

HumSAT software evolution (lines of code)



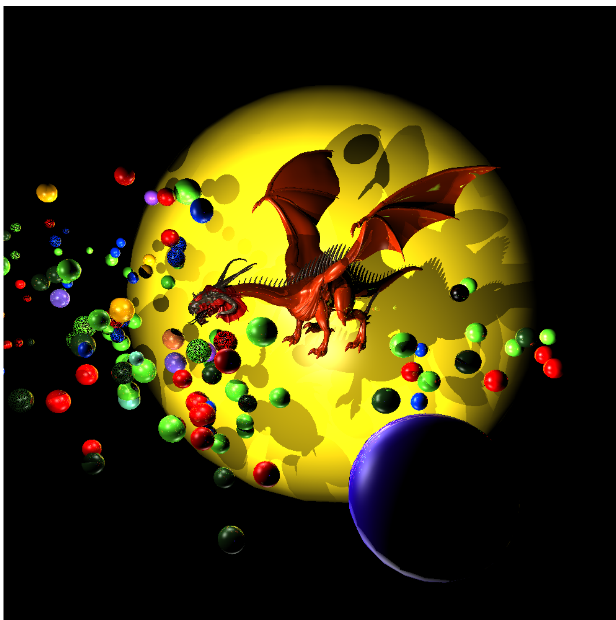
reconocimiento de formas

- investigación y desarrollo en el ámbito de reconocimiento de formas
  - reconocer formas
  - aprender formas
- uso en aplicaciones de
  - educación infantil,
  - educación matemática,
  - interfaces amigables
  - interfaces para motores de búsqueda
- miramos un poco el prototipo Shapewiz...

trazado de rayos

- descripción de un modelado de una escena (objecto, colores, texturas, luces, etc.)
- simulación de una cámara digital
- obtención de una imágen *foto-realista*

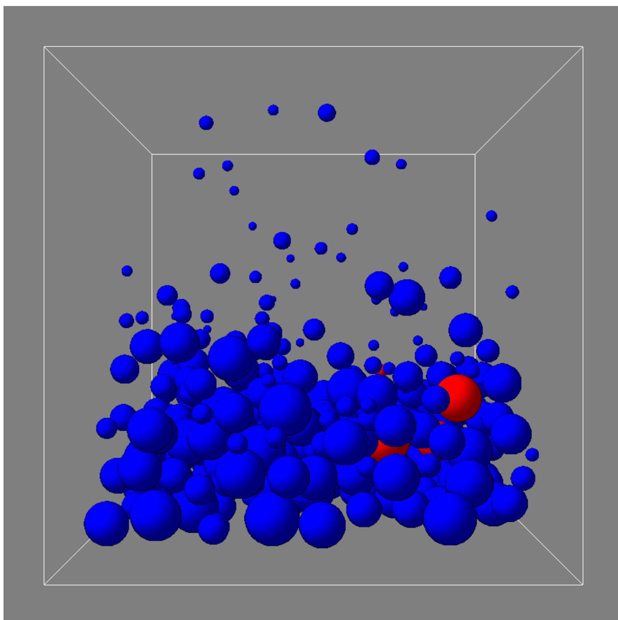
trazado de rayos



simulación de partículas

- descripción de un modelado de partículas
  (tamaño, propiedades pre/pos-colisión, otros objetos, etc.)
- simulación según modelado física
  (p.ej. con gravitación)
- simulación basado en eventos discretos
- o simulación basado en monte carlo
- estudio de efectos en materiales granulares y gases

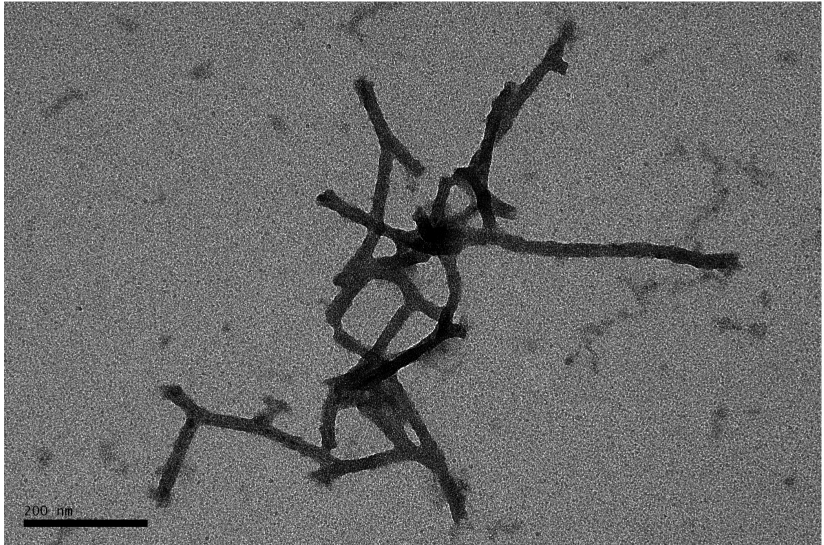
pemd



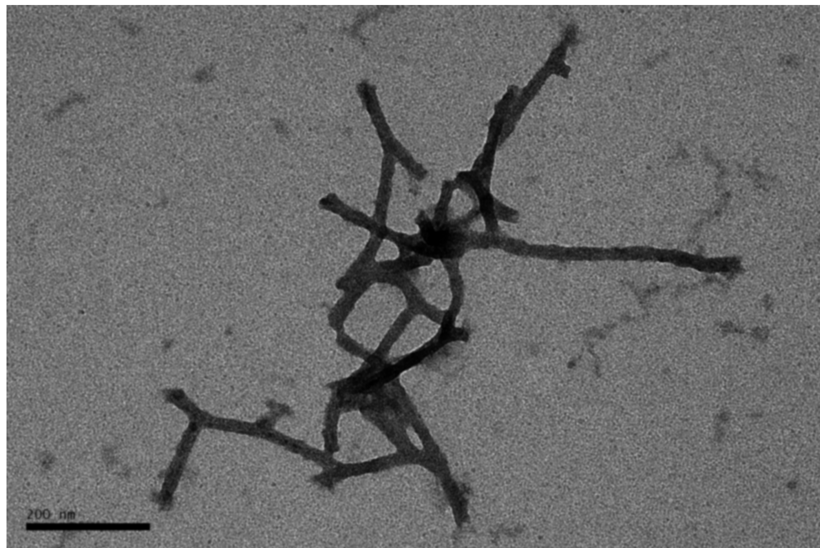
pemd

```
pemd_g -Ri 0.5 -Ra 3.0 -c 1000000 -en 0.75 -i
cdi.pemd -g 2 -pn 343 -ps 343
```

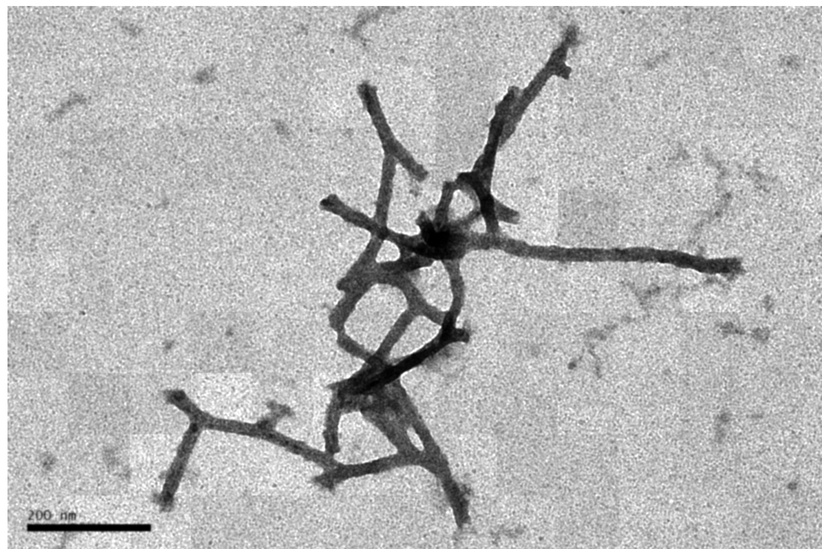
procesamiento de imágenes
bio-nanotubos (original)



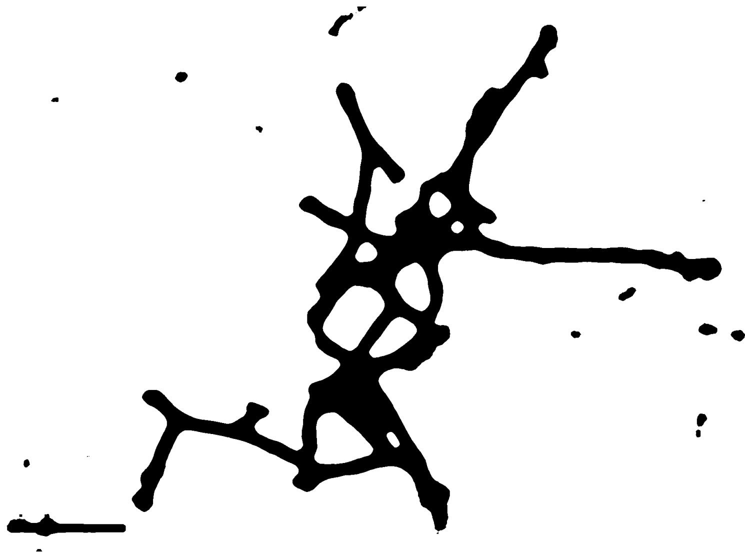
procesamiento de imágenes
bio-nanotubos (menos ruido)



procesamiento de imágenes
bio-nanotubos (más contraste)

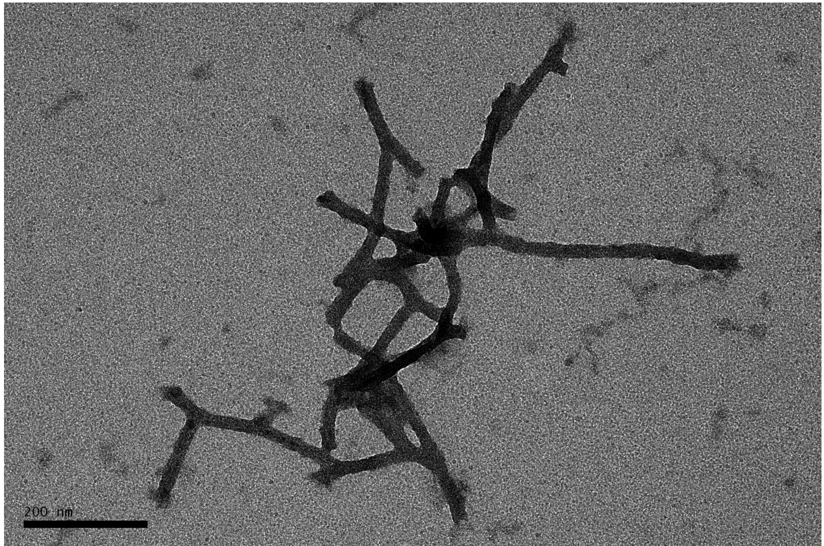


procesamiento de imágenes
bio-nanotubos (binarizado)



procesamiento de imágenes
bio-nanotubos (esqueleto)

procesamiento de imágenes
bio-nanotubos (original)



posible colaboración, ejemplos durante el curso

- Todas estas aplicaciones se usarán como vehículos de ejemplo para mostrar aspectos de concurrencia y distribución.
- Quien quiere colaborar, por ejemplo en su TFG, puede ponerse en contacto conmigo.

contenido (optimista)

Tema	Contenido
Sistemas concurrentes y distribuidos	Concepto de la programación concurrente y distribuida, Introducción al modelado de sistemas concurrentes y distribuidos, Arquitecturas hardware para la concurrencia y distribución, Herramientas para el desarrollo de aplicaciones concurrentes y distribuidos
Procesos	Concepto de procesos, Planificador, Atomicitad y exclusión mutua, Concurrencia transactional, Reloj y estado distribuido

contenido (optimista)

Tema	Contenido
Sincronización y comunicación	Sincronización y comunicación en sistemas concurrentes y distribuidos, Sincronización y comunicación a nivel bajo y alto, Seguridad y vivacidad en sistemas concurrentes y distribuidos
Herramientas de programación y desarrollo de aplicaciones	Programación concurrente y distribuida con JAVA (y C/C++), Patrones de diseño para el desarrollo de aplicaciones concurrentes y distribuidos, Herramientas y metodologías de diseño, verificación y depuración de aplicaciones concurrentes y distribuidos

Nota

Prácticamente todas las asignaturas optativas en uno u otro aspecto requieren del concepto de concurrencia y distribución en sistemas modernos para lograr sus objetivos específicos.

- Este documento crecerá durante el curso,
  ojo, no necesariamente solamente al final.
- Habrá más documentos (capítulos de libros, manuales, etc.) con que trabajar durante el curso.
- Los ejemplos de programas y algoritmos serán en inglés.
- Las transparencias no están (posiblemente/probablemente) ni correctos ni completos.
- Las transparencias no son suficientes (incluso çhapadas") para aprobar la asignatura.

Competencias	Tipo	Cod.
Que os estudantes demostren posuír e comprender coñecementos nunha área de estudo que parte da base da educación secundaria xeral e adoita atoparse a un nivel que, malia se apoiar en libros de texto avanzados, inclúe tamén algúns aspectos que implican coñecementos procedentes da vangarda do seu campo de estudo.	facer	CB1
Que os estudantes saiban aplicar os seus coñecementos ó seu traballo ou vocación dunha forma profesional e posúan as competencias que adoitan demostrarse por medio da elaboración e defensa de argumentos e a resolución de problemas dentro da súa área de estudo.	facer	CB2
Que os estudantes teñan a capacidade de reunir e interpretar datos relevantes (normalmente dentro da súa área de estudo) para emitir xuízos que inclúan unha reflexión sobre temas relevantes de índole social, científica ou ética.	facer	CB3
Que os estudantes desenvolvan aquelas habilidades de aprendizaxe necesarias para emprender estudos posteriores cun alto grao de autonomía.	facer	CB4

competencias, un intento...

Competencias	Tipo	Cod.
Capacidade para concebir, redactar, organizar, planificar, desenvolver e asinar proxectos no ámbito da enxeñaría en informática que teñan por obxecto, de acordo cos coñecementos adquiridos , a concepción, o desenvolvemento ou a explotación de sistemas, servizos e aplicacións informáticas.	facer	CG1
Capacidade para dirixir as actividades obxecto dos proxectos do ámbito da informática de acordo cos coñecementos adquiridos.	saber	CG2
Capacidade para deseñar, desenvolver, avaliar e asegurar a accesibilidade, ergonomía, usabilidade e seguridade dos sistemas, servizos e aplicacións informáticas, así como da información que xestionan.	facer	CG3
Capacidade para definir, avaliar e seleccionar plataformas hardware e software para o desenvolvemento e a execución de sistemas, servizos e aplicacións informáticas, de acordo cos coñecementos adquiridos.	facer	CG4

Competencias	Tipo	Cod.
Capacidade para concebir, desenvolver e manter sistemas, servizos e aplicacións informáticas empregando os métodos da enxeñería de software como instrumento para o aseguramento de sua calidade, de acordo cos coñecementos adquiridos.	facer	CG5
Capacidad para concebir e desenvolver sistemas ou arquitecturas informáticas centralizadas ou distribuidas integrando hardware, software e redes de acordo cos coñecementos adquiridos.	facer	CG6
Capacidade para coñecer, comprender e aplicar a lexislación necesaria durante o desenvolvemento da profesión de Enxeñeiro Técnico en Informática e manexar especificacións, regulamentos e normas de obrigado cumprimento.	saber	CG7
Coñecemento das materias básicas e tecnoloxías, que capaciten para a aprendizaxe e desenvolvemento de novos métodos e tecnoloxías, así como as que lles doten dunha gran versatilidade para adaptarse a novas situacións.	facer	CG8

competencias, un intento...

Competencias	Tipo	Cod.
Capacidade para resolver problemas con iniciativa, toma de decisións, autonomía e creatividade. Capacidade para saber comunicar e transmitir os coñecementos, habilidades e destrezas da profesión de Enxeñeiro Técnico en Informática.	saber	CG9
Capacidade para analizar e valorar o impacto social e medioambiental das solucións técnicas, comprendendo a responsabilidade ética e profesional da actividade de Enxeñeiro Técnico en Informática.	saber	CG11
Coñecemento e aplicación de elementos básicos de economía e de xestión de recursos humáns, organización e planificación de proxectos, así como a lexislación, regulación e normalización no ámbito dos proxectos informáticos, de acordo cos coñecementos adquiridos.	saber	CG12

competencias, un intento...

Competencias	**Tipo**	**Cod.**
Coñecementos básicos sobre o uso e programación dos ordenadores, sistemas operativos, bases de datos e programas informáticos con aplicación na enxeñería	facer	CE4
Coñecemento da estrutura, organización, funcionamento e interconexión dos sistemas informáticos, os fundamentos da súa programación, e a súa aplicación para a resolución de problemas propios da enxeñería	saber	CE5
Capacidade para deseñar, desenvolver, seleccionar e avaliar aplicacións e sistemas informáticos, asegurando a súa fiabilidade, seguridade e calidade, conforme aos principios éticos e á lexislación e normativa vixente	saber	CE7

competencias, un intento...

Competencias	**Tipo**	**Cod.**
Capacidade para planificar, concibir, despregar e dirixir proxectos, servizos e sistemas informáticos en tódolos ámbitos, liderando a súa posta en marcha e mellora continua e valorando o seu impacto económico e social	saber	CE8
Coñecemento e aplicación dos procedementos algorítmicos básicos das tecnoloxías informáticas para deseñar solucións a problemas, analizando a idoneidade e complexidade dos algoritmos propostos	facer	CE12

Competencias	Tipo	Cod.
Coñecemento, deseño e utilización de forma eficiente dos tipos e estruturas de datos máis axeitados á resolución dun problema	facer	CE13
Capacidade para analizar, deseñar, construír e manter aplicacións de forma robusta, segura e eficiente, elixindo o paradigma e as linguaxes de programación máis axeitadas	facer	CE14
Capacidade de coñecer, comprender e avaliar a estrutura e arquitectura dos computadores, así como os compoñentes básicos que os conforman	facer	CE15

Competencias	Tipo	Cod.
Coñecemento das características, funcionalidades e estrutura dos Sistemas Operativos e deseñar e implementar aplicacións baseadas nos seus servizos	saber	CE16
Coñecemento e aplicación das ferramentas necesarias para o almacenamento, procesamento e acceso aos Sistemas de información, incluídos os baseados en web	saber	CE19
Coñecemento e aplicación dos principios fundamentais e técnicas básicas da programación paralela, concurrente, distribuída e de tempo real	facer	CE20

competencias, un intento...

Competencias	**Tipo**	**Cod.**
Coñecemento e aplicación dos principios, metodoloxías e ciclos de vida da enxeñería de software	saber	CE22
Capacidade para desenvolver, manter e avaliar servizos e sistemas software que satisfagan todos os requisitos do usuario e se comporten de forma fiable e eficiente, sexan asequibles de desenvolver e manter e cumpran normas de calidade, aplicando as teorías, principios, métodos e prácticas da Enxeñería do Software	saber	CE25

competencias, un intento...

Competencias	Tipo	Cod.
Capacidade para valorar as necesidades do cliente e especificar os requisitos software para satisfacer estas necesidades, reconciliando obxectivos en conflito mediante a procura de compromisos aceptables dentro das limitacións derivadas do custo, do tempo, da existencia de sistemas xa desenvolvidos e das propias organizacións	saber	CE26
Capacidade de dar solución a problemas de integración en función das estratexias, estándares e tecnoloxías dispoñibles	saber	CE27
Capacidade de identificar e analizar problemas e deseñar, desenvolver, implementar, verificar e documentar solucións software sobre a base dun coñecemento axeitado das teorías, modelos e técnicas actuais	facer	CE28

Competencias	Tipo	Cod.
Capacidade para deseñar solucións apropiadas nun ou máis dominios de aplicación utilizando métodos da enxeñería do software que integren aspectos éticos, sociais, legais e económicos	saber	CE30
Capacidade para comprender a contorna dunha organización e as súas necesidades no ámbito das tecnoloxías da información e as comunicacións	saber	CE31
Capacidade para empregar metodoloxías centradas no usuario e a organización para o desenvolvemento, avaliación e xestión de aplicacións e sistemas baseados en tecnoloxías da información que aseguren a accesibilidade, ergonomía e usabilidade dos sistemas	saber	CE33

competencias, un intento...

Competencias	**Tipo**	**Cod.**
Capacidade para seleccionar, despregar, integrar e xestionar sistemas de información que satisfagan as necesidades da organización, cos criterios de custo e calidade identificados	saber	CE35
Capacidade de concibir sistemas, aplicacións e servizos baseados en tecnoloxías de rede, incluíndo Internet, web, comercio electrónico, multimedia, servizos interactivos e computación móbil	saber	CE36

competencias, un intento...

Competencias	Tipo	Cod.
Capacidade de análise, síntese e avaliación	ser	CT1
Capacidade de organización e planificación	ser	CT2
Comunicación oral e escrita na lingua nativa	ser	CT3
Capacidade de abstracción: capacidade de crear e utilizar modelos que reflictan situacións reais	ser	CT5
Capacidade de deseñar e realizar experimentos sinxelos e analizar e interpretar os seus resultados	ser	CT6
Capacidade de buscar, relacionar e estruturar información proveniente de diversas fontes e de integrar ideas e coñecementos	ser	CT7
Resolución de problemas	ser	CT8

Competencias	Tipo	Cod.
Capacidade de tomar decisións	ser	CT9
Capacidade para argumentar e xustificar loxicamente as decisións tomadas e as opinións	ser	CT10
Capacidade de actuar autonomamente	ser	CT11
Capacidade de traballar en situacións de falta de información e/ou baixo presión	ser	CT12
Capacidade de relación interpersoal	ser	CT15
Razoamento crítico	ser	CT16
Aprendizaxe autónoma	ser	CT18
Creatividade	ser	CT20
Ter iniciativa e ser resolutivo	ser	CT22
Ter motivación pola calidade e a mellora continua	ser	CT24

Bibliografía (de interés) I

1. J.T. Palma Méndez, M.C. Garrido Carrera, F. Sánchez Figueroa, A. Quesada Arencibia. *Programación Concurrente.* Thomson, ISBN 84-9732-184-7, 2003.
2. G. Coulouris, J. Dollimore, T. Kindberg. *Sistemas Distribuidos, Conceptos y Diseño.* Addison Wesley, ISBN 84-7829-049-4, 2001.
3. M.L. Liu. *Computación Distribuida* Peason/Addison Wesley, ISBN 84-7829-066-4, 2004.
4. C. Breshears. *The Art of Concurrency.* O'Reilly, ISBN 978-0-596-52153-0, 2009.

Bibliografía (de interés) II

5. M. Herlihy, N. Shavit. *The Art of multiprocessor programming.* Elsevier-Morgan Kaufmann Publishers, ISBN 978-0-12-370591-4, 2008 (978-0-12-397337-5, 2012 revised edition).
6. D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture, Pattern for Concurrent and Networked Objects.* John Wiley & Sons, ISBN 0-471-60695-2, 2000.

Bibliografía (Java)

1. K. Arnold et.al. *The Java Programming Language.* Addison-Wesley, 3rd Edition, ISBN 0-201-70433-1, 2000.
2. B. Eckel. *Piensa en Java.* Prentice Hall, 2002.
3. D. Lea. *Programación Concurrente en Java.* Addison-Wesley, ISBN 84-7829-038-9, 2001.

- cualquier libro sobre Java que cubre programación con hilos

Bibliografía (antigua)

1. M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, ISBN 0-13-711821-X, 1990.
2. G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.
3. J.C. Baeten and W.P. Wiejland. *Process Algebra.* Cambridge University Press, 1990.
4. A. Burns and G. Davies. *Concurrent Programming.* Addison-Wesley, 1993.
5. C. Fencott. *Formal Methods for Concurrency.* Thomson Computer Press, 1996.
6. M. Henning, S. Vinoski. *Programación Avanzada en CORBA con C++.* Addison Wesley, ISBN 84-7829-048-6, 2001.

Bibliografía (más antigua) I

6. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
7. R. Milner. *Concurrency and Communication.* Prentice-Hall, 1989.
8. R. Milner. *Semantics of Concurrent Processes.* in J. van Leeuwen (ed.), Handbook of Theoretical Computer Science. Elsevier and MIT Press, 1990.
9. J.E. Pérez Martínez. *Programación Concurrente.* Editorial Rueda, ISBN 84-7207-059-X, 1990.
10. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

Bibliografía (on–line)

- Apuntes de esta asignatura:
  http://formella.webs.uvigo.es/doc/cdg17/index.html
- Concurrency JSR-166 Interest Site (antiguado)
  http://gee.cs.oswego.edu/dl/concurrency-interest/index.html
- El paquete de Doug Lea que funciona con Java 1.4 (antiguado)
  http://formella.webs.uvigo.es/doc/concurrent.tar.gz
  (.tar.gz [502749 Byte])
- The Java memory model (antiguado)
  http://www.cs.umd.edu/%7Epugh/java/memoryModel

Bibliografía VI (adicional)

- G. Bracha. *Generics in the Java Programming Language.* July 5, 2004.

- buscadores en la red

Existen diversas definiciones de los términos en la literatura:

- programación concurrente
- programación paralela
- programación distribuida

definición

Una posible distinción (según mi opinión) es:

- la programación concurrente se dedica más a desarrollar y aplicar conceptos para el uso de recursos en paralelo (desde el punto de vista de varios actores)
- la programación en paralelo se dedica más a solucionar y analizar problemas bajo el concepto del uso de recursos en paralelo (desde el punto de vista de un sólo actor)

definición

Otra posibilidad de separar los términos es:

- un programa concurrente define las acciones que se pueden ejecutar simultaneamente, es decir, están en progreso
- un programa paralelo es un programa concurrente diseñado de ser ejecutado en hardware paralelo
- un programa distribuido es un programa paralelo diseñado de ser ejecutado en hardware distribuido, es decir, donde varios procesadores no tengan memoria compartida, tienen que intercambiar la información mediante de transmisión de mensajes/datos.

intuición

Intuitivamente, todos tenemos una idea básica de lo que significa el concepto de concurrencia.

©http://kartenspiel.org/knack-kartenspiel/

ordenamos

- Formamos ordenadores.
- Cada uno tiene procesadores que se pueden comunicar entre ellos (sabeis hablar y escuchar...).
- Cada procesador tiene un neipe.
- Queremos que al final la barraja esté ordenada.

- ¿Cómo procedemos?
- ¿Cuáles son los aspectos/problemas a tratar?
- ¿Cómo lo trasladamos a un entorno programable?

ordenamos

Creative Common: Sönke Kraft aka Arnulf zu Linden

ordenamos

Creative Common: Sönke Kraft aka Arnulf zu Linden

multiplicamos

```
837649587637 * 984758392081 = ?
```

¿Con qué problemas nos enfrentamos?

- selección del algoritmo
- RAE: Conjunto ordenado (?!) y finito (?!) de operaciones que permite hallar la solución de un problema.
- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los resultados

problemas

¡Y también con!

- medición de características
- depuración del programa
- (fiabilidad de los componentes)
- (fiabilidad de la comunicación)
- (detección de la terminación)

La programación concurrente puede ser divertida y frustante a la vez :-)

multiplicamos otra vez

```
837649587637 * 984758392081 = ?
824882461048724816302597
```

Java

Este repasito de algunas características de Java no es

- ni completo
- ni exhaustivo
- ni suficiente

para programar en Java.
Debe servir solamente

- para refrescar conocimiento ya adquerido,
- para introducir algunos conceptos útil para la concurrencia y
- para animar de profundizar el estudio del lenguaje con otras fuentes, por ejemplo, con la bibliografía añadida y los manuales correspondientes (mirad boletín de prácticas).

Java

- Se destacan ciertas diferencias con C++ (otro lenguaje de programación orientado a objetos importante).
- Se comentan ciertos detalles del lenguaje que muchas veces no se perciben a primera vista.
- Se introducen los conceptos ya instrínsicos de Java para la programación concurrente.

hola mundo

El famoso *hola mundo* se programa en Java así:

```
class Hello {
  public static void main(String[] args) {
    System.out.println(
      "Hello world, this is "+args[0]
    );
  }
}
```

El programa principal se llama `main()` y tiene que ser declarado público y estático. No devuelve ningún valor (por eso se declara como `void`).

Los parámetros de la línea de comando se pasan como un vector de cadenas de letras (`String`).

¿Qué se comenta?

Existen varias posibilidades de escribir comentarios:

`//`	comentario de línea
`/// ...`	comentario de documentación
`/* ... */`	comentario de bloque
`/** ... */`	comentario de documentación

- Se usa doxygen (o javadoc, u otro bueno) para generar automáticamente la documentación. Todos tienen unos comandos para aumentar la documentación.
- Se documenta sobre todo lo que no es obvio, las interfaces (en el sentido amplio de la palabra), y los casos límite.
- Es decir: Los comentarios son las respuestas a preguntas del *¿Cómo?* y del *¿Por qué?*.

Se usan los hilos para ejecutar varias secuencias de instrucciones de modo cuasi–paralelo.
Es decir, la ejecución parece ser en paralelo, pero si realmente es paralelo (y no secuencial) depende del hardware (más preciso, del número de CPUs involucrados) que se está usando en cada momento.

creación de un hilo (para empezar)

- Se crea un hilo con
  `Thread worker = new Thread()`
- Se inicializa el hilo y se define su comportamiento.
- Normalmente se usa una clase derivada o la interfaz `Runnable`.
- Se lanza el hilo con
  `worker.start()`
- Pero en esta versión simple no hace nada. Hace falta sobreescribir el método `run()` especificando algún código útil.
- ¿Cuándo arranca de verdad este nuevo hilo?

la interfaz `Runnable`

- A veces no es conveniente extender la clase `Thread` porque se pierde la posibilidad de extender otro objeto.
- Es una de las razones por que existe la interfaz `Runnable` que declara nada más que el método `public void run()` y que se puede usar fácilmente para crear hilos trabajadores.

ping I

```
class RunPing implements Runnable {
  private String word;
  private int delay;

  RunPing(String word, int delay) {
    this.word =word;
    this.delay=delay;
  }
  public void run() {
    try {
      for(;;) {
        System.out.print(word+" ");
        Thread.sleep(delay);
      }
    }
    catch(InterruptedException e) { return; }
  }
}
```

ping II

```java
class Ping {
  public static void main(String[] args) {
    Runnable ping = new RunPing("ping", 50);
    Runnable PONG = new RunPing("PONG", 100);
    new Thread(ping).start();
    new Thread(PONG).start();
  }
}
```

resumen respecto a concurrencia

- `Thread, Runnable`
- `join, isAlive, activeCount`
- `try-catch-finally`
- `volatile`
- `synchronized`
- `wait, notify, notifyAll`
- `finalize`
- `transient`
- `java.util.concurrent`
- `java.util.concurrent.atomic`
- etc.

construcción por defecto

- Sólo si una clase no contiene ningún constructor Java propone un constructor por defecto que tiene el mismo modificador de acceso que la clase.
- Constructores pueden lanzar excepciones como cualquier otro método.

constructores

- Para facilitar la construcción de objetos, es posible usar bloques de código sin que pertenezcan a constructores.
- Esos bloques están prepuestos (en su orden de aparencia) delante de los códigos de todos los constructores.
- El mismo mecanismo se puede usar para inicializar miembros estáticos poniendo un `static` delante del bloque de código.
- Inicializaciones estáticas no pueden lanzar excepciones chequeadas (ya que no se pueden envolver en bloques `try-catch`).

bloques de inicialización

```
class Bloques {
  private int i=1;
  Bloques(int j) {
    i+=j;
  }
  public void Show() {
    System.out.println("i="+i);
  }


  public static void main(String[] args) {
    Bloques b= new Bloques(5);
    b.Show();
  }

  { i=5; }
}
```

inicialización estática

```java
class Static {
  static int[] powertwo=new int[10];
  static {
    powertwo[0]=1;
    for(int i=1; i<powertwo.length; ++i)
      powertwo[i]=powertwo[i-1]<<1;
  }
  public static void main(String[] args) {
    for(int i=0; i<powertwo.length; ++i)
      System.out.println(
        powertwo[i]
      );
  }
}
```

- No existe un operador para eliminar objetos *de la memoria*, eso es tarea del recolector de memoria incorporado en Java (diferencia con C++ donde se tiene que liberar memoria con `delete` explícitamente).
- Para dar pistas de ayuda al recolector se puede asignar `null` a una referencia indicando al recolector que no se va a referenciar con esta referencia dicho objeto nunca jamás.
- Antes de ser destruido se ejecuta el método `finalize()` del objeto (por defecto no hace nada).
- En un programa concurrente obviamente el recolector de memoria tiene que ser un algoritmo sobre una estructura de datos concurrente.

finalize

```
class Something {
  static int cnt=0;
  private int i;

  Something() { i=1; }

  protected void finalize() {
    System.out.println("ooh, I got killed finally");
    System.out.println(++cnt);
  }
}

class Final {
  public static void main(String[] args) {
    Something s;
    for(int i=0; i<Integer.parseInt(args[0]); ++i) {
      s=new Something();
      s=null;
    }
```

parámetros no modificables no existen

- No se puede evitar posibles modificaciones de un parámetro (que sí se puede evitar hasta cierto punto en C++ declarando el parámetro como `const`-referencia).
- Declarando el parámetro como `final` solamente protege la propia referencia (paso por valor).
- La declaración se puede/suele usar como indicación al usuario que se pretende no cambiar el objeto (aunque el compilador no lo garantiza).
- Ojo con las clases `Integer` y primas: son clases y no lo son. (Más preciso: son clases *inmutables*, aunque la documentación de Java no habla mucho sobre ello, entonces se comportan cuando se usan como parámetros igual como tipos simples.)

efectos secundarios

```
import java.util.concurrent.atomic.*;
class Int {
  public int i;
  Int(int i) { this.i=i; }
}
class Change {
  public static void Print(AtomicInteger i) {
    System.out.println("i="+i);      i.getAndIncrement();
  }
  public static void Print(Integer i) {
    System.out.println("i="+i);      ++i;
  }
  public static void Print(Int i) {
    System.out.println("i="+i.i);    ++i.i;
  }
  public static void main(String[] args) {
    AtomicInteger i=new AtomicInteger(0); Print(i); Print(i);
    Integer j=new Integer(0);             Print(j); Print(j);
    Int k=new Int(0);                     Print(k); Print(k);
  }
}
```

- Los vectores se declaran solamente con su límite súperior dado que el límite ínferior siempre es cero (0).
- El código

  ```
  int[] vector = new int[15]
  ```

  crea un vector de números enteros de longitud 15.

control de acceso

- Java comprueba si los accesos a vectores con índices quedan dentro de los límites permitidos (diferencia con C++ donde no hay una comprobación).
- Si se detecta un acceso fuera de los límites se produce una excepción `IndexOutOfBoundsException`.
- Dependiendo de las capacidades del compilador eso puede resultar en una pérdida de rendimiento.

vectores son objetos

- Los vectores son objetos implícitos que siempre conocen sus propias longitudes (`values.length`) (diferencia con C++ donde tal vector no es nada más que un puntero) y que se comportan como clases finales.
- No se pueden declarar los elementos de un vector como constantes (como es posible en C++), es decir, el contenido de los componentes siempre se puede modificar en un programa en Java.
- No se puede modificar el tamaño de un vector una vez creado (diferencia con C++ donde tal longitud es dinámico).

- Para facilitar la programación de casos excepcionales Java usa el concepto de lanzar excepciones.
- Una excepción es una clase predefinida y se accede con la sentencia

```
try { ... }
catch(SomeExceptionObject e) { ... }
catch(AnotherExceptionObject e) { ... }
finally { ... }
```

orden de ejecución I

- El bloque `try` contiene el código normal por ejecutar.
- Un bloque `catch(ExceptionObject)` contiene el código excepcional por ejecutar en caso de que durante la ejecución del código normal (que contiene el bloque `try`) se produzca la excepción del tipo adecuado.
- Pueden existir más de un (o ningún) bloque `catch` para reaccionar directamente a más de un (ningún) tipo de excepción.
- Hay que tener cuidado en ordenar las excepciones correctamente, es decir, las más específicas antes de las más generales.

orden de ejecución II

- El bloque `finally` se ejecuta siempre una vez terminado o bien el bloque `try` o bien un bloque `catch` o bien una excepción no tratada o bien antes de seguir un `break`, un `continue` o un `return` hacia fuera de la sentencia `try-catch-finally`.
- Por eso es un buen punto donde ejecutar código en aplicaciones concurrentes que deben realizar cierto *trabajo* final, incluso en casos excepcionales.

Normalmente se extiende la clase `Exception` para implementar clases propias de excepciones, aún también se puede derivar directamente de la clase `Throwable` que es la superclase (interfaz) de `Exception` o de la clase `RuntimeException`.

```
class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s) { super(s); }
}
```

declaración de excepciones lanzables

- Entonces, una excepción no es nada más que un objeto que se crea en el caso de aparición del caso excepcional.
- La clase principal de una excepción es la interfaz `Throwable` que incluye un `String` para mostrar una línea de error legible.
- Para que un método pueda lanzar excepciones con las sentencias `try-catch-finally`, es imprecindible declarar las excepciones posibles antes del bloque de código del método con `throws ...`.

```
public void myfunc(...) throws MyException {...}
```

- En C++ es al revés, se declara lo que se puede lanzar como mucho.

propagación de excepciones

- Durante la ejecución de un programa se propagan las excepciones desde su punto de aparición subiendo las invocaciones de los métodos hasta que se haya encontrado un bloque `catch` que se ocupa de tratar la excepción.
- En el caso de que no haya ningún bloque responsable, la excepción será tratada por la máquina virtual con el posible resultado de abortar el programa.

lanzar excepciones

- Se pueden lanzar excepciones directamente con la palabra `throw` y la creación de un nuevo objeto de excepción, por ejemplo:

```
throw new MyException("this is an exception");
```

- También los constructores pueden lanzar excepciones que tienen que ser tratados en los métodos que usan dichos objetos construidos.

- Además de las excepciones así declaradas existen siempre excepciones que pueden ocurrir en qualquier momento de la ejecución del programa, por ejemplo, `RuntimeException` o `Error` o `IndexOutOfBoundException`.
- La ocurrencia de dichas excepciones refleja normalmente un flujo de control erróneo del programa que se debe corrigir antes de distribuir el programa a posibles usuarios.

- Recomendación: Se usan excepciones solamente para casos excepcionales, es decir, si pasa algo no esperado.
- Excepciones en programas concurrentes se convierten rápidamente en pesadillas.
- Mira Safe (Disciplined) Exception Handling principle con sus dos posibilidades de acción
  - fallo: re-establecer una invariante necesaria
  - re-intentar: re-hacer la operación con (quizá) cierta modificación

algoritmo secuencial

- Asumimos que tengamos solamente las operaciones aritméticas *sumar* y *subtraer* disponibles en un procesador ficticio y queremos multiplicar dos números positivos.
- Un posible algoritmo sequencial que multiplica el número `p` con el número `q` produciendo el resultado `r` es:

```
Initially:  set p and q to positive numbers

a: set r to 0
b: loop
c:   if q equal 0 exitloop
d:   set r to r+p
e:   set q to q-1
f: endloop
g: ...
```

¿Cómo se comprueba si el algoritmo es correcto?

- Primero tenemos que decir que significa correcto.
- El algoritmo (secuencial) es correcto si
  - una vez se llega a la instrucción `g:` el valor de la variable `r` contiene el producto de los valores de las variables `p` y `q` (se refiere a sus valores que han llegado a la instrucción `a:`)
  - se llega a la instrucción `g:` en algún momento
  - y la entrada habia sido la correcta.

¿Cómo se comprueba si el algoritmo es correcto?

- Tenemos que saber que las instrucciones atómicas son correctas,
- es decir, sabemos exactamente su significado, incluyendo todos los efectos segundarios posibles.
- Luego usamos el concepto de inducción para comprobar el bucle.

Invariante e inducción

- Sean p_i, q_i, y r_i los contenidos de los registros `p`, `q`, y `r`, después de la iteración i del bucle.
- Una invariante cuya corrección hay que comprobar con el concepto de inducción es entonces:

$$r_i + p_i \cdot q_i = p \cdot q$$

- ¿Cómo encontrar una invariante adecuada?
- usar *ingenio*...
  (RAE: Facultad del ser humano para discurrir o inventar con prontitud y facilidad.)
- Además, si comprobramos que q_i al final (saliendo del bucle) es cero, entonces, obviamente, el registro `r` contendrá el producto.

Re-escribimos el algoritmo secuencial para que “*funcione*” con dos procesos:

```
Initially:  set p and q to positive numbers

a: set r to 0
 P0                         P1
b: loop                       loop
c:   if q equal 0 exit          if q equal 0 exit
d:   set r to r+p               set r to r+p
e:   set q to q-1               set q to q-1
f: endloop                    endloop
g: ...
```

Implementamos la multiplicación con Java

- Realizamos una clase para valores enteros.
- Implementamos el bucle que realiza la multiplicación.
- Colocamos todo en un programa completo.

Realizamos una clase para tener los enteros como clase propia (`Integer` no nos vale):

```
class Int {
  int i;
  Int(int i) { this.i=i; }
  void Add(Int I) { i=i+I.i; }
}
```

Preparar trabajador

Preparamos los hilos trabajadores con acceso a las variables comunes:

```
class Mul implements Runnable {
  int id;    // thread identity
  Int p;     // reference to shared first factor
  Int q;     // reference to shared second factor
  Int r;     // reference to shared result
  Mul(int id, Int p, Int q, Int r) {
    this.id=id;
    this.p=p;
    this.q=q;
    this.r=r;
  }
```

El trabajo del trabajador

Implementamos el método `run()` para realizar el bucle de multiplicación:

```
  public void run() {
    try {
      System.out.println("starting worker... "+id);
      Int minusOne=new Int(-1);
      while(q.i!=0) {
        r.Add(p);
        q.Add(minusOne);
      }
    }
    catch(Exception E) { System.out.println("??? "+id);
    finally { System.out.println("exiting... "+id); }
  }
}
```

El programa principal I

Tratar la entrada:

```
class Multi {
  static Int p,q,r;

  public static void main(String[] args) {
    try {
      if(args.length!=3) {
        System.out.println("please 3 arg's: p q n");
        System.exit(1);
      }

      p=new Int(Integer.parseInt(args[0]));
      q=new Int(Integer.parseInt(args[1]));
      r=new Int(0);
```

El programa principal II

Crear, lanzar, y sincronizar los trabajadores:

```
final int n=Integer.parseInt(args[2]);
final Thread[] threads=new Thread[n];

for(int i=0; i<threads.length; ++i) {
  threads[i]=new Thread(new Mul(i,p,q,r));
}

for(int i=0; i<threads.length; ++i) {
  threads[i].start();
}

for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
```

El programa principal III

Visualización del resultado:

```
        System.out.println(
          args[0]+"*"+args[1]+"="+r.i+" ??"
        );
      }
      catch(Exception E) {
        System.out.println("catched an exception...");
        System.exit(1);
      }
      finally {
        System.out.println("exiting...");
      }
    }
}
```

¿Qué es que observamos?

- El algoritmo es no-determinista,
- en el sentido que no se sabe de antemano en qué orden (en un procesador o en un conjunto de procesadores) se van a ejecutar las instrucciones,
- o más preciso, cómo se van a intercalar las instrucciones atómicas de ambos procesos.
- El no-determinismo puede provocar situaciones con errores, es decir, el fallo ocurre solamente si las instrucciones se ejecutan en un orden específico.
- El resultado del programa no es predecible, y lo peor es, a veces es correcto;
- deste el punto de vista de concurrencia tiene varios *problemas*.

Generalmente se dice que un programa es correcto, si dada una entrada, el programa produce los resultados deseados.

Más formal:

- Sea $P(x)$ una propiedad de una variable x de entrada (aquí el símbolo x refleja cualquier conjunto de variables de entradas).
- Sea $Q(x, y)$ una propiedad de una variable x de entrada y de una variable y de salida.

funcionamiento correcto II

Se define dos tipos de funcionamiento correcto de un programa:

funcionamiento correcto parcial:
: dada una entrada a, si $P(a)$ es verdadero, y si se lanza el programa con la entrada a, entonces si el programa termina habrá calculado b y $Q(a,b)$ también es verdadero.

funcionamiento correcto total:
: dado una entrada a, si $P(a)$ es verdadero, y si se lanza el programa con la entrada a, entonces el programa termina y habrá calculado b con $Q(a,b)$ siendo también verdadero.

funcionamiento correcto III

Para un programa secuencial existe solamente un orden total de las instrucciones atómicas (en el sentido que un procesador secuencial siempre sigue el mismo orden de las instrucciones... bueno, es mentira..., hay *out-of-order and speculative execution*), mientras que para un programa concurrente puedan existir varios órdenes.
Por eso se tiene que exigir:

funcionamiento correcto concurrente:
: un programa concurrente funciona correctamente, si el resultado $Q(x, y)$ no depende del orden de las instrucciones atómicas entre todos los órdenes posibles.

Funcionamiento correcto IV

Entonces:

- Se debe asumir que los hilos pueden intercalarse en cualquier punto en cualquier momento.
- Los programas no deben estar basados en la suposición de que habrá un intercalado específico entre los hilos por parte del planificador (que conmuta los procesos).

funcionamiento correcto V

- Para comprobar si un programa concurrente es *incorrecto* basta con encontrar una sola intercalación de instrucciones que nos lleva en un fallo.
- Para comprobar si un programa concurrente es *correcto* hay que comprobar que no se produce ningún fallo en ninguna de las intercalaciones posibles.

comprobación exhaustiva no es práctico

- El número de posibles intercalaciones de los procesos en un programa concurrente crece exponencialmente con el número de unidades que maneja el planificador y líneas por intercalar.
- ¿Cuántos son? (Ayuda: calcular las combinaciones posibles de una lista dentro de otra)
- Por eso es prácticamente imposible comprobar con la mera enumeración si un programa concurrente es correcto bajo todas las ejecuciones posibles.
- En la argumentación hasta ahora era muy importante que las instrucciones se ejecutaran de forma atómica, es decir, sin interrupción ninguna.
- Por ejemplo, se observa una gran diferencia si el procesador trabaja directamente en memoria o si trabaja con registros.

dependencia de atomicidad I

Si `increment` es atómico:

```
P1:  inc N
P2:  inc N
```

```
P2:  inc N
P1:  inc N
```

Se observa: las dos intercalaciones posibles producen el resultado correcto.

dependencia de atomicidad II

Si `increment` no es atómico:

```
P1:  load  R1,N
P2:  load  R2,N
P1:  inc   R1
P2:  inc   R2
P1:  store R1,N
P2:  store R2,N
```

Es decir, existe una intercalación que produce un resultado falso.

Ejemplos de Java:

- accesos a variables con más de 4 bytes no son atómicos.
- el operador ++ no es atómico.

¿Y?

- ¿El algoritmo concurrente de multiplicación con dos hilos de arriba es correcto? (correcto parcial? correcto total?)
- ¿Cómo lo compruebas?
- ¿Te ocurre un algoritmo concurrente correcto que funcione con varios hilos?
- ¿Te ocurre un algoritmo concurrente correcto y eficiente, es decir, donde varios hilos juntos son más rápido que uno sólo?

Sincronización

- A veces se quiere que un hilo actua sin que otros interfieran en su tarea.
- Es decir, se quiere una ejecución con exclusión mutua del código.
- Dicho concepto se llama también atomicidad de las operaciones,
- o también ejecución segura con hilos (*threadsafe*).
- En Java existen diferentes posibilidades para conseguir exclusión mutua.

Atomicidad en Java

- Solo las asignaciones a variables de tipos simples de 32 bits son atómicas.
- `long` y `double` no son simples en este contexto porque son de 64 bits.
- Hay que declarar esas variables como `volatile` para obtener acceso atómico.

synchronized

- En Java es posible forzar la ejecución del código en un bloque de modo sincronizado, es decir, como mucho un hilo, que tenga acceso compartido a `obj`, puede ejecutar el código dentro de dicho bloque.

```
synchronized(obj) { ... }
```

- La expresión entre paréntesis `obj` tiene que evaluar a una referencia a un objeto o a un vector.
- Declarando un método con el modificador `synchronized` garantiza que dicho método se ejecuta por un sólo hilo (y ningún otro método sincronizado del mismo objeto tampoco).
- La máquina virtual instala un cerrojo (mejor dicho, un monitor, ya veremos dicho concepto más adelante) que se cierra de forma atómica antes de entrar en la región crítica y que se abre antes de salir.

Métodos syncronizados

- Declarar un método como
  `synchronized void f(...) { ... }`
  es equivalente a usar un bloque sincronizado en su interior:
  `void f(...) { synchronized(this) { ... } }`
- Los monitores (implementados en la MVJ) permiten que el mismo hilo puede acceder a otros métodos o bloques sincronizados del mismo objeto sin problema.
- Se libera el cerrojo sea el modo que sea que termine el método.
- Los constructores no se pueden declarar `synchronized`.

Uso de clases de `java.util.concurrent`

Java proporciona el paquete `java.util.concurrent`
Se puede implementar el algoritmo de multiplicación de antes con, por ejemplo:

- métodos sincronizados
- `AtomicInteger`
- `LongAdder` (a partir de Java8)
- `LongAccumulator` (a partir de Java8)
- Hay diferencias en eficiencia.

Enteros con `int` y método sincronizado

```
class Int {
  private int i;
  Int(int i) { this.i=i; }
  synchronized void Add(Int I) { i=i+I.i; }
  int Get() { return i; }
}
```

Enteros con `Integer` y método sincronizado

```
class Int {
  private Integer i;
  Int(int i) { this.i=i; }
  synchronized void Add(Int I) { i=i+I.i; }
  int Get() { return i; }
}
```

Enteros con AtomicInteger

```
import java.util.concurrent.atomic.*;

class Int {
  private AtomicInteger i;
  Int(int i) { this.i=new AtomicInteger(i); }
  void Add(Int I) { i.getAndAdd(I.Get()); }
  int Get() { return i.get(); }
}
```

Enteros con LongAdder

```
import java.util.concurrent.atomic.*;

class Int {
  private LongAdder i;
  Int(int i) {
    this.i=new LongAdder();
    this.i.add(i);
  }
  void Add(Int I) { i.add(I.Get()); }
  int Get() { return i.intValue(); }
}
```

Enteros con LongAccumulator

```
import java.util.concurrent.atomic.*;

class Int {
  private LongAccumulator i;
  Int(int i) {
    this.i=new LongAccumulator(Long::sum, i);
  }
  void Add(Int I) { i.accumulate(I.Get()); }
  int Get() { return i.intValue(); }
}
```

Multiplicación concurrente correcto

Cuidado que los hilos no hagan nada demás...

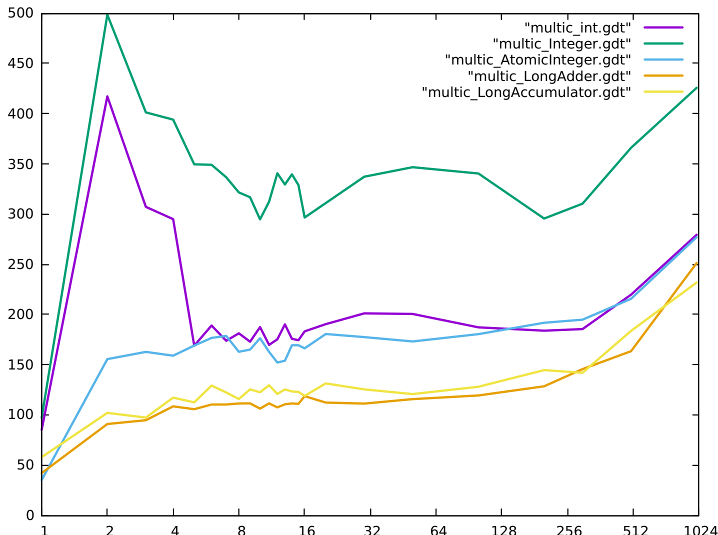
```
public void run() {
  try {
    final Int minusOne=new Int(-1);
      // Here, we must check greater that n
      // to avoid negative q !!
    while(q.Get()>n) {
      r.Add(p);
      q.Add(minusOne);
    }
  }
  catch(Exception E) {
    System.out.println("some error..."+id);
  }
```

Multiplicación concurrente correcto

Cuidado que se realiza todo lo que hay que hacer...

```
for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
  // Here we must take care of the left-overs.
final Int minusOne=new Int(-1);
while(q.Get()>0) {
  r.Add(p);
  q.Add(minusOne);
}
```

Pero el algoritmo no es eficiente, ya que hay mucha congestion accediendo a `q` y `r`:



Multiplicación concurrente correcto y eficiente

```
class Mul implements Runnable {
  private final int id;
  private final int n;
  private int p;
  private int q;
  private Int r;

  Mul(int id, int n, Int p, Int q, Int r) {
    this.id=id;
    this.n=n;
    this.p=p.Get();
    this.q=q.Get()/n;
    this.r=r;
  }
```

Multiplicación concurrente correcto y eficiente

```
public void run() {
  try {
      // Here, we run on local variables.
    int my_r=0;
    while(q>0) {
      my_r+=p;
      --q;
    }
    final Int My_r=new Int(my_r);
    r.Add(My_r);
  }
  catch(Exception E) {
    System.out.println("some error..."+id);
  }
}
```

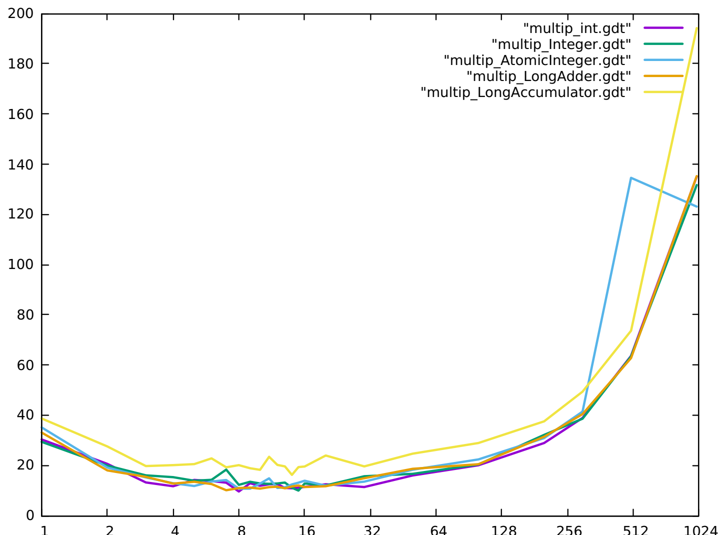
Multiplicación concurrente correcto y eficiente

```
for(int i=0; i<threads.length; ++i) {
  threads[i].start();
}
  // Here we help with the left-overs.
int my_p=p.Get();
int my_q=q.Get()%n;
int my_r=0;
while(my_q>0) {
  my_r+=my_p;
  --my_q;
}

for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
final Int My_r=new Int(my_r);
r.Add(My_r);
```

Multiplicación concurrente correcto y eficiente

Ahora el algoritmo es eficiente, ya que hay mucha mucho menos congestion... (1000 veces más rápido...)



Excursión: miramos el problema de filtrar una matriz

miramos un posible código, y como llegar a él...

Sincronización y herencia

- No hace falta mantener el modo sincronizado sobre-escribiendo métodos síncronos mientras se extiende una clase.
- No se puede *forzar* un método sincronizada en una interfaz.
- Sin embargo, una llamada al método de la clase súperior (con `super.`) sigue funcionando de modo síncrono.
- Los métodos estáticos también se pueden declarar `synchronized` garantizando su ejecución de manera exclusiva entre varios hilos.

Protección de miembros estáticos

En ciertos casos se tiene que proteger el acceso a miembros estáticos con un cerrojo. Para conseguir eso es posible sincronizar con un cerrojo de la clase, por ejemplo:

```
class MyClass {
  static private int nextID;
  ...
  MyClass() {
    synchronized(MyClass.class) {
      idNum=nextID++;
    }
  }
  ...
}
```

¡Ojo con el concepto!

- Declarar un bloque o un método como síncrono solo prevee que ningún otro hilo pueda ejecutar al mismo tiempo dicha región crítica (u otra sincronizada con el mismo objeto),
- sin embargo, cualquier otro código asíncrono puede ser ejecutado mientras tanto y su acceso a variables críticas puede dar como resultado fallos o efectos inesperados en el programa.

Objetos síncronos

Se obtienen objetos totalmente sincronizados siguiendo las reglas:

- todos los métodos son `synchronized`,
- no hay miembros/atributos públicos,
- todos los métodos son `final`,
- se inicializa siempre todo bien,
- el estado del objeto se mantiene siempre consistente incluyendo los casos de excepciones.

Limitaciones para la programación concurrente

- no se puede interrumpir la espera a un cerrojo (una vez llegado a un `synchronized` no hay vuelta atrás)
- no se puede influir mucho en la política del cerrojo (distinguir entre lectores y escritores, diferentes justicias, etc.)
- no se puede confinar el uso de los cerrojos (en cualquier línea se puede escribir un bloque sincronizado de cualquier objeto)
- no se puede adquirir/liberar un cerrojo en diferentes flujos de control, se está obligado a una estructura de bloques

Paquete especial para la programación concurrente

- Por eso, y otros motivos, se ha introducido desde Java 5 un paquete especial para la programación concurrente.

  `java.util.concurrent`
- Hay que leer/estudiar todo su manual.
- Partes mencionaremos en clase en su momento.

Páginas del manual

Se recomienda estudiar detenidamente las páginas del manual de Java que estén relacionados con el concepto de hilo (mira también las referencias en el boletín de prácticas).

un programa concurrente

- Asumimos que tengamos un programa concurrente que quiere realizar acciones con recursos.
  (por ejemplo, los factores de la multiplicacion, o mirad las prácticas)
- Si los recursos de los diferentes procesos son diferentes no hay problema (mira por ejemplo la práctica de filtrado de matriz),
- Si dos (o más procesos) quieren manipular el mismo recurso ¿Qué hacemos?
- Vimos ya ayudas Java como `AtomicInteger`, o el `synchronized`...
- levantamos el nivel a algo más abstracto y luego implementamos a nivel bajo.

un programa concurrente

Tenemos básicamente tres opciones:

- se implementa exclusión mutua, es decir, solamente un proceso tiene acceso, los demás esperan;
- se implementa comportamiento idéntico, es decir, desde el algoritmo se garantiza que todos los procesos actuan igual (sobre todo: escriben lo mismo en caso que escriban concurrentemente);
- se implementa comportamiento transaccional, es decir, solo un proceso gana, lo que hacen los demás no influe en su resultado (con la opción que los demás se notifiquen en caso de fracaso) (con la opción que cualquiera o uno específico gana).

¿Qué es exclusión mutua?

- Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control
  - que permite la entrada de un proceso si el recurso está disponible y
  - que prohibe la entrada de un proceso si el recurso está ocupado.
- Es importante entender cómo se implementan los protocolos de entrada y salida para realizar la exclusión mutua.
- Obviamente no se puede implementar exclusión mutua usando exclusión mutua: se necesita algo más básico.
- Un método es usar un tipo de protocolo de comunicación basado en las instrucciones básicas disponibles en el hardware.
  (Eso veremos más adelante.)

Entonces el protocolo para cada uno de los participantes refleja una estructura como sigue (si protegemos código):

```
P0                          ... Pi
...                             ...
entrance protocol               entrance protocol
critical section                critical section
exit protocol                   exit protocol
...                             ...
```

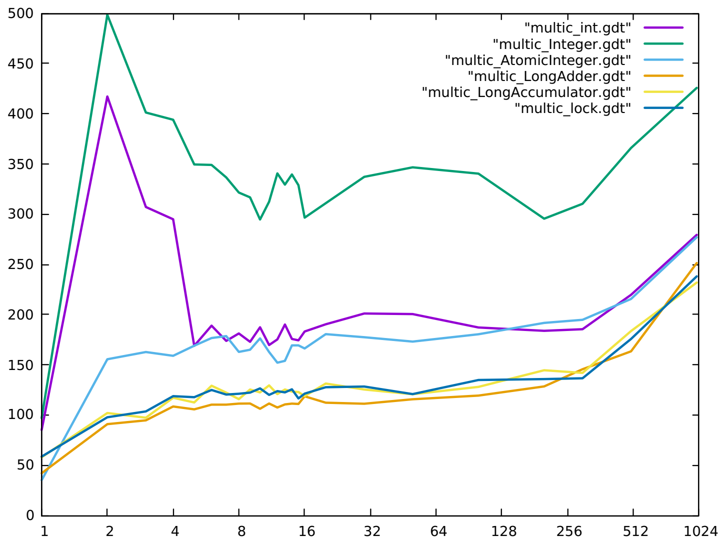
Enteros con lock

```java
import java.util.concurrent.locks.*;

class Int {
  private int i;
  private ReentrantLock lock;
  Int(int i) {
    this.i=i;
    lock=new ReentrantLock();
  }
  void Add(Int I) {
    lock.lock(); // Entrance protocol.
    try {
      i+=I.i;
    } finally {
      lock.unlock(); // Exit protocol.
    }
  }
  int Get() { return i; }
}
```

Multiplicación concurrente correcto con `lock`

El algoritmo no eficiente implementado con cerrojo:



- El concepto de usar estructuras de datos a nivel alto libera al/a programador/a de los detalles de su implementación.
- Se puede asumir que las operaciones están implementadas correctamente y se puede basar el desarrollo del programa concurrente en un funcionamiento correcto de las operaciones de los tipos de datos abstractos.
- Para que se puedan utilizar con provecho hay que entender en detalle las propiedades de tales estructuras de datos.

- Un lenguaje de programación puede realizar directamente una implementación de una región crítica.
- Así parte de la responsabilidad se traslada desde el programador al compilador.
- De alguna manera (depende del lenguaje de programación en concreto) se identifica que algún bloque de código se debe tratar como región crítica
  (así funciona Java con sus bloques sincronizados):

```
V is shared variable
region V do
  code of critical region
```

observaciones

- El compilador asegura que la variable V tenga un protocolo de entrada y salida adjunto que se usa para controlar el acceso exclusivo de un solo proceso a la región crítica.
- De este modo no hace falta que el programador use directamente las operaciones de los protocolos para controlar el acceso con el posible error de olvidarse de alguna parte esencial.
- Adicionalmente es posible que dentro de la región crítica se llame a otra parte del programa que a su vez contenga una región crítica. Si ésta está controlada por la misma variable V el proceso obtiene automáticamente también acceso a dicha región.

regiones críticas condicionales

- En muchas situaciones es conveniente controlar el acceso de varios procesos a una región crítica por una condición.
- Con las regiones críticas simples, vistas hasta ahora, no se puede realizar tal control. Hace falta otra construcción, por ejemplo:

```
V is shared variable
C is boolean expression
region V when C do
  code of critical region
```

detalles de implementación I

Las regiones críticas condicionales funcionan internamente de la siguiente manera:

- Un proceso que quiere entrar en la región crítica espera hasta que tenga permiso.
- Una vez obtenido permiso comprueba el estado de la condición, si la condición lo permite, entra en la región, en caso contrario, libera el cerrojo y se pone de nuevo esperando en la cola de acceso.

- Se implementa una región crítica normalmente con dos colas diferentes.
- Una cola principal controla los procesos que quieren acceder a la región crítica, una cola de eventos controla los procesos que ya han obtenido una vez el cerrojo pero que han encontrado la condición en estado falso.
- Si un proceso sale de la región crítica todos los procesos que quedan en la cola de eventos pasan de nuevo a la cola principal porque tienen que recomprobar la condición.

detalles de implementación III

- Nota que esta técnica puede derivar en muchas comprobaciones de la condición, todos en modo exclusivo, y puede causar pérdidas de eficiencia.
- En ciertas circunstancias puede ser interesante realizar un control más sofisticado del acceso a la región crítica dando paso directo de un proceso a otro.

Un semáforo es un tipo de datos abstracto que permite el uso de un recurso de manera exclusiva cuando varios procesos están competiendo y que cumple la siguiente semántica:

- El estado interno del semáforo cuenta cuantos procesos todavía pueden utilizar el recurso. Se puede realizar, por ejemplo, con un número entero que nunca debe llegar a ser negativo.
- Existen tres operaciones con un semáforo: `init()`, `wait()`, y `signal()` que realizan lo siguiente:

operaciones del semáforo I

`init()`:

- Inicializa el semáforo antes de que cualquier proceso haya ejecutado ni una operación `wait()` ni una operación `signal()` al límite de número de procesos que tengan derecho a acceder el recurso.
- Si se inicializa con 1, se ha construido un semáforo binario.
- En lenguajes orientados a objectos, la operación `init()` se suele realizar en la construcción del objeto correspondiente.

`wait()`:

- Si el estado indica cero, el proceso se queda atrapado en el semáforo hasta que sea despertado por otro proceso.
- Si el estado indica que un proceso más puede acceder el recurso se decrementa el contador y la operación termina con exito.
- La operación `wait()` tiene que estár implementada como una instrucción atómica. Sin embargo, en muchas implementaciones la operación `wait()` puede ser interrumpida.
- Normalmente existe una forma de comprobar si la salida del `wait()` es debido a una interrupción o porque se ha dado acceso al semáforo.
- Importante: esta comprobación se debe hacer!

operaciones del semáforo III

`signal()`:

- Una vez se ha terminado el uso del recurso, el proceso lo señaliza al semáforo. Si queda algún proceso bloqueado en el semáforo, uno de ellos sea despertado, sino se incrementa el contador.
- La operación `signal()` también tiene que estár implementada como instrucción atómica. En algunás implementaciones es posible comprobar si se ha despertado un proceso con exito en caso que había alguno bloqueado.
- Para despertar los procesos se pueden implementar varias formas que se destinguen en su política de justicia (p.ej. FIFO).

El acceso mutuo a secciones críticas se arregla con un semáforo que permita el acceso a un sólo proceso

```
S.init(1)

P1                          P2
a: loop                         loop
b:   S.wait()                     S.wait()
c:   critical section             critical section
d:   S.signal()                   S.signal()
e:   non-critical section         non-critical section
f: endloop                      endloop
```

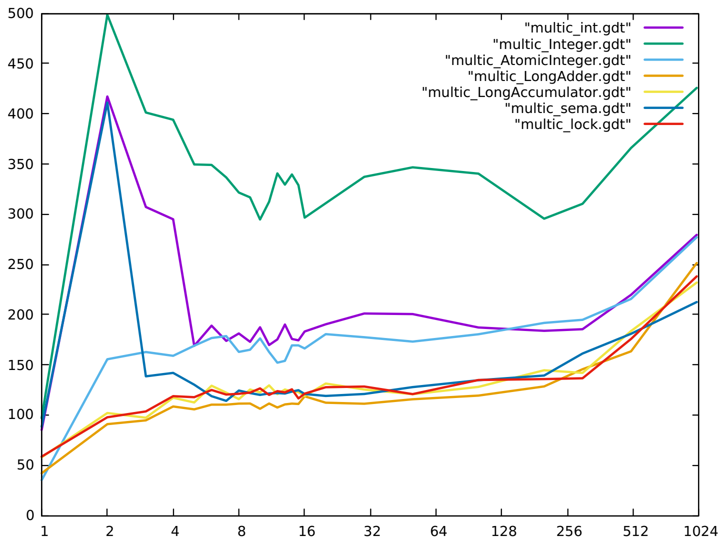
Enteros con Semaphore

```
import java.util.concurrent.Semaphore;

class Int {
  private int i;
  private Semaphore semaphore;
  Int(int i) {
    this.i=i;
    semaphore=new Semaphore(1);
  }
  void Add(Int I) {
    try {
      semaphore.acquire(); // Entrance protocol.
      i+=I.i;
    }
    catch(InterruptedException E) {
      System.out.println("got interrupted...??");
    }
    finally {
      semaphore.release(); // Exit protocol.
    }
  }
```

Multiplicación concurrente correcto con `semaphore`

El algoritmo no eficiente implementado con semáforo:



Si existen en un entorno solamente semáforos binarios, se puede simular un semáforo general usando dos semáforos binarios y un contador, por ejemplo, con las variables `delay`, `mutex` y `count`.

- La operación `init()` inicializa el contador al número máximo permitido.
- El semáforo `mutex` asegura acceso mutuamente exclusivo al contador.
- El semáforo `delay` atrapa a los procesos que superan el número máximo permitido.

detalles de la implementación I

La operación `wait()` se implementa de la siguiente manera:

```
delay.wait()
mutex.wait()
decrement count
if count greater 0 then delay.signal()
mutex.signal()
```

La operación `signal()` se implementa de la siguiente manera:

```
mutex.wait()
increment count
if count equal 1 then delay.signal()
mutex.signal()
```

- No se puede imponer el uso correcto de las llamadas a los `wait()`s y `signal()`s.
- No existe una asociación entre el semáforo y el recurso.
- Entre `wait()` y `signal()` el usuario puede realizar cualquier operación con el recurso.

observaciones II

- Las regiones críticas no son lo mismo que los semáforos, porque no se tiene acceso directo a las operaciones `init()`, `wait()` y `signal()`.
- Con semáforos se puede emular regiones críticas pero no al revés.

Un monitor es un tipo de datos abstracto que contiene

- un conjunto de procedimientos de control de los cuales solamente un subconjunto es visible desde fuera,
- un conjunto de datos privados, es decir, no visibles desde fuera.

detalles de implementación de un monitor

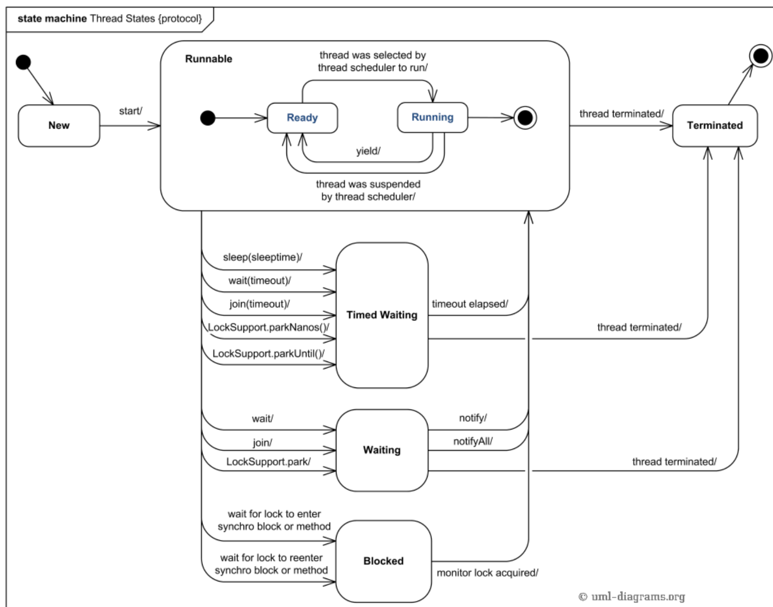
- El acceso al monitor está permitido solamente a través de los métodos públicos y el compilador garantiza exclusión mutua para todos los accesos.
- La implementación del monitor controla la exclusión mutua con colas de entrada que contengan todos los procesos bloqueados.
- Pueden existir varias colas y el controlador del monitor elige de cual cola se va a escoger el siguiente proceso para actuar sobre los datos.
- Un monitor no tiene acceso a variables exteriores con el resultado de que su comportamiento no puede depender de ellos.
- Una desventaja de los monitores es la exclusividad de su uso, es decir, la concurrencia está limitada si muchos procesos hacen uso del mismo monitor.

- Un aspecto que se encuentra en muchas implementaciones de monitores es la sincronización condicional, es decir, mientras un proceso está ejecutando un procedimiento del monitor (con exclusión mutua) puede dar paso a otro proceso liberando el cerrojo. (Estas operaciones se suele llamar `wait` o `delay`).
- El proceso que ha liberado el cerrojo se queda bloqueado hasta que otro proceso le despierta de nuevo. Este bloqueo temporal está realizado dentro del monitor.
- Dicha técnica se refleja en Java con `wait()` y `notify()/notifyAll()`.
- La técnica permite la sincronización entre procesos porque actuando sobre el mismo recurso los procesos pueden cambiar el estado del recurso y pasar así información de un proceso a otro.

disponibilidad de monitores

- Lenguajes de alto nivel que facilitan la programación concurrente suelen tener monitores implementados dentro del lenguaje (por ejemplo en Java: todos los objetos derivan de `Object` que contiene los métodos `wait()` y `notify`/`notifyAll()`.
- El uso de monitores es bastante costoso, porque se puede perder eficiencia por bloquear los procesos innecesariamente y el trabajo adicional por el uso del monitor.
- Por eso se intenta aprovechar de primitivas más potentes para aliviar este problema (alternativas libres de cerrojos (*lock free*) y/o libres de espera (*wait free*)).

Java máquina de estados de hilos



instrucciones básicas

- obviamente tenemos que asumir que ciertas acciones de un proceso se puede realizar correctamente independientemente de las acciones de los demás procesos
- dichas acciones se llaman atómicas (porque son indivisibles) y se garantizan por hardware
- asumimos que podemos acceder a variables de cierto tipo (p.ej. enteros) de forma atómica con lectura y escritura (`load` y `store`)

Un posible protocolo (asimétrico)

```
  P0                              P1
a: loop                            loop
b:  non-critical section            non-critical section
c:  set v0 to true                  set v1 to true
d:  wait until v1 equals false      while v0 equals true
e:                                   set v1 to false
f:                                   wait until v0 equals false
g:                                   set v1 to true
h:  critical section                critical section
i:  set v0 to false                 set v1 to false
j: endloop                         endloop
```

principio de la bandera

- Si ambos procesos primero levantan sus banderas
- y después miran al otro lado
- por lo menos un proceso ve la bandera del otro levantado.

comprobación con contradicción

- asumimos P0 era el último en mirar
- entonces la bandera de P0 está levantada
- asumimos que P0 no ha visto la bandera de P1
- entonces P1 ha levantado la bandera después de la mirada de P0
- pero P1 mira después de haber levantado la bandera
- entonces P0 no era el último en mirar

propiedades de interés del protocolo

Normalmente, un protocolo de entrada y salida debe cumplir con las siguientes condiciones:

- sólo un proceso debe obtener acceso a la sección crítica (garantía del acceso con exclusión mutua)
- por lo menos un proceso debe obtener acceso a la sección crítica después de un tiempo de espera *finito*.
- Obviamente se asume que ningún proceso ocupa la sección crítica durante un tiempo infinito.

La propiedad de espera finita se puede analizar según los siguientes criterios:

justicia:
: hasta que medida influyen las peticiones de los demás procesos en el tiempo de espera de un proceso

espera:
: hasta que medida influyen los protocolos de los demás procesos en el tiempo de espera de un proceso

tolerancia a fallos:
: hasta que medida influyen posibles errores de los demás procesos en el tiempo de espera de un proceso.

Analizamos el protocolo de antes respecto a dichos criterios:

- ¿Está garantizado la exclusión mutua?
- ¿Influye el estado de uno (sin acceso) en el acceso del otro?
- ¿Quién gana en caso de peticiones simultaneas?
- ¿Qué pasa en caso de error?

soporte hardware

- Dependiendo de las capacidades del hardware la implementación de los protocolos de entrada y salida es más fácil o más difícil, además las soluciones pueden ser más o menos eficientes.
- Vimos y veremos que se pueden realizar protocolos seguros solamente con las instrucciones `load` y `store` de un procesador.
- Las soluciones no suelen ser muy eficientes, especialmente si muchos procesos compiten por la sección crítica. *Pero: su desarrollo y la presentación de la solución ayuda en entender el problema principal.*
- A veces no hay otra opción disponible.
- Todos los microprocesadores modernos proporcionan instrucciones básicas que permiten realizar los protocolos de forma más directo y en muchas ocaciones más eficiente.

protocolo simple: primer intento

Usamos una variable v que nos indicará cual de los dos procesos tiene su turno.

```
  P0                            P1
a: loop                          loop
b:  wait until v equals P0        wait until v equals P1
c:  critical section              critical section
d:  set v to P1                   set v to P0
e:  non-critical section          non-critical section
f: endloop                       endloop
```

primer intento: propiedades

- Está garantizada la exlusión mutua porque un proceso llega a su línea `c:` solamente si el valor de `v` corresponde a su identificación (que asumimos siendo única).
- Obviamente, los procesos pueden acceder al recurso solamente alternativamente, que puede ser inconveniente porque acopla los procesos fuertemente.
- Un proceso no puede entrar más de una vez seguido en la sección crítica.
- Si un proceso termina el programa o no llega más por alguna razón a su línea `d:`, el otro proceso puede resultar bloqueado.
- La solución se puede ampliar fácilmente a más de dos procesos.

primer intento en Java (comenzar es fácil)

El protocolo del primer intento para implementar un ping pong.

```
public class PingPong {
  volatile static int turn;  // It's v.

  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);

    ping1.start();
    ping2.start();

    turn=1;
  }
```

primer intento en Java (comenzar es fácil)

```java
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                          // a:
      while(
        id!=PingPong.turn                  // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                // d:
    }                                      // f:
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

un método de para abreviar:

```
static void Wait(int us) {
  try {
    Thread.sleep(us);
  } catch(InterruptedException e) {
    System.out.println("sleeping interrupted");
  }
 }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
public class PingPong {
  volatile static int turn;  // It's v.
  public static void main(String[] args) {
    Thread ping1 = new Player(1);
    Thread ping2 = new Player(2);
    ping1.start();
    ping2.start();
    System.out.println("playing some seconds");
    turn=1;
    Wait(2000);
    System.out.println("waiting for players");
    turn=3;  // Try to stop :-)
    try {
      ping1.join();
      ping2.join();
    }
    catch(InterruptedException e) {
      System.out.println("got interrupted");
    }
    System.out.println("finished");
  }
```

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(true) {                             // a:
      while(
        id!=PingPong.turn                     // b:
      );
      System.out.print("ping"+id+" ");// c:
      PingPong.turn=id%2+1;                   // d:
    }                                         // f:
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

Excurso: primer intento en Java (terminar no tanto)

```
class Player extends Thread {
  private int id;
  public Player(int id) { this.id=id; }
  public void run() {
    while(PingPong.turn!=3) {
      while(
        id!=PingPong.turn &&
        PingPong.turn!=3
      );
      System.out.print("ping"+id+" ");
      if(PingPong.turn==id) {
        PingPong.Wait(100);       // And blocking!!!
        PingPong.turn=id%2+1;
      }
    }
  }
}
```

Intentamos evitar la alternancia. Usamos para cada proceso una variable, `v0` para `P0` y `v1` para `P1` respectivamente, que indican si el correspondiente proceso está usando el recurso.

```
  P0                             P1
a: loop                           loop
b:  wait until v1 equals false     wait until v0 equals false
c:  set v0 to true                 set v1 to true
d:  critical section               critical section
e:  set v0 to false                set v1 to false
f:  non-critical section           non-critical section
g: endloop                        endloop
```

segundo intento: propiedades

- Ya no existe la situación de la alternancia.
- Sin embargo: el algoritmo no está seguro, porque los dos procesos pueden alcanzar sus secciones críticas simultáneamente.

- El problema está escondido en el uso de las variables de control. `v0` se debe cambiar a verdadero solamente si `v1` sigue siendo falso.
- ¿Cuál es la intercalación maligna?

tercer intento

Cambiamos el lugar donde se modifica la variable de control:

```
  P0                             P1
a: loop                           loop
b:  set v0 to true                 set v1 to true
c:  wait until v1 equals false     wait until v0 equals false
d:  critical section               critical section
e:  set v0 to false                set v1 to false
f:  non-critical section           non-critical section
g: endloop                        endloop
```

- Está garantizado que ambos procesos no entren al mismo tiempo en sus secciones críticas.
- Pero se bloquean mutuamente en caso que lo intenten simultaneamente que resultaría en una espera infinita.

- ¿Cuál es la intercalación maligna?

Modificamos la instrucción `c:` para dar la oportunidad que el otro proceso encuentre su variable a favor.

```
  P0                        P1
a: loop                      loop
b:  set v0 to true            set v1 to true
c:  repeat                    repeat
d:   set v0 to false           set v1 to false
e:   set v0 to true            set v1 to true
f:  until v1 equals false     until v0 equals false
g:  critical section          critical section
h:  set v0 to false           set v1 to false
i:  non-critical section      non-critical section
j: endloop                   endloop
```

cuarto intento: propiedades

- Está garantizado la exclusión mutua.
- Se puede producir una variante de bloqueo: los procesos hacen algo pero no llegan a hacer algo útil (*livelock*)
- ¿Cuál es la intercalación maligna?

algoritmo de Dekker: quinto intento

```
Initially:  v0,v1 are equal to false, v is equal to P0 o P1

  P0                           P1
a: loop                         loop
b:  set v0 to true               set v1 to true
c:  loop                         loop
d:   if v1 equals false exit      if v0 equals false exit
e:   if v equals P1               if v equals P0
f:    set v0 to false              set v1 to false
g:    wait until v equals P0       wait until v equals P1
h:    set v0 to true               set v1 to true
i:   fi                           fi
j:  endloop                      endloop
k:  critical section             critical section
l:  set v0 to false              set v1 to false
m:  set v to P1                  set v to P0
n:  non-critical section         non-critical section
o: endloop                      endloop
```

El algoritmo de Dekker resuelve el problema de exclusión mutua en el caso de dos procesos, donde se asume que la lectura y la escritura de un valor íntegro de un registro se puede realizar de forma atómica.

algoritmo de Peterson

```
  P0                         P1
a: loop                       loop
b:  set v0 to true             set v1 to true
c:  set v to P0                set v to P1
d:  wait while                 wait while
e:   v1 equals true             v0 equals true
f:   and v equals P0            and v equals P1
g:  critical section           critical section
h:  set v0 to false            set v1 to false
i:  non-critical section       non-critical section
j: endloop                    endloop
```

algoritmo de Lamport

o algoritmo de la panadería:

- cada proceso tira un ticket (que están ordenados en orden ascendente)
- cada proceso espera hasta que su valor del ticket sea el mínimo entre todos los procesos esperando
- el proceso con el valor mínimo accede la sección crítica

algoritmo de Lamport: observaciones

- ya se necesita un cerrojo (acceso con exclusión mutua) para acceder a los tickets,
- el número de tickets no tiene límite,
- los procesos tienen que comprobar continuadamente todos los tickets de todos los demás procesos.
- El algoritmo no es verdaderamente practicable dado que se necesita un número infinito de tickets y un número elevado de comprobaciones.
- Si se sabe el número máximo de participantes basta con un número fijo de tickets.

- Como vimos, el algoritmo de Lamport (algoritmo de la panadería) necesita muchas comparaciones de los tickets para n procesos.
- Existe una versión de Peterson que usa solamente variables confinadas a cuatro valores.
- Existe una generalización del algoritmo de Peterson para n procesos (*filter algorithm*).
- Se puede evitar la necesidad de un número infinito de tickets, si se conoce antemano el número máximo de participantes (uso de grafos de precedencia).
- Otra posibilidad es al algoritmo de Eisenberg–McGuire (que garantiza una espera mínima para n procesos).

límites

- Se puede comprobar que se necesita por lo menos *n* campos en la memoria común para implementar un algoritmo (con `load` and `store`) que garantiza la exclusión mutua entre *n* procesos.

Operaciones en la memoria

- Si existen instrucciones más potentes (que los simples `load` y `store`) en el microprocesador se puede realizar la exclusión mutua más fácil.
- Hoy (casi) todos los procesadores implementan un tipo de instrucción atómica que realiza algún cambio en la memoria al mismo tiempo que devuelve el contenido anterior de la memoria.
- La idea principal es implementar ciclos de *read-modify-write* de forma atómica directamente en memoria compartida.

TAS

La instrucción `test-and-set` (TAS) implementa

- una comprobación a cero del contenido de una variable en la memoria
- al mismo tiempo que varía su contenido
- en caso que la comprobación se realizó con el resultado verdadero.

TAS

```
Initially:  vi is equal false
            C  is equal true

a: loop
b:   non-critical section
c:   loop
d:     if C equals true                ; atomic
         set C to false and exit
e:   endloop
f:   set vi to true
g:   critical section
h:   set vi to false
i:   set C to true
j: endloop
```

TAS: propiedades

- En caso de un sistema multi-procesador hay que tener cuidado que la operación `test-and-set` esté realizada en la memoria compartida.
- Teniendo solamente una variable para la sincronización de varios procesos el algoritmo arriba no garantiza una espera limitada de todos los procesos participando.
  ¿Por qué?
- ¿Cómo se puede garantizar una espera limitada?
- Para conseguir una espera limitada se implementa un protocolo de paso de tal manera que un proceso saliendo de su sección crítica da de forma explícita paso a un proceso esperando (en caso que tal proceso exista).

EXCH

La instrucción `exchange`:

- intercambia un registro del procesador
- con el contenido de una dirección de la memoria
- en una instrucción atómica.

EXCH

```
Initially:  vi is equal false
            C  is equal true

a: loop
b:   non-critical section
c:   loop
d:     exchange C and vi        ; atomic exchange
e:     if vi equals true exit
f:   endloop
g:   critical section
h:   exchange C and vi
i: endloop
```

EXCH: propiedades

- Se observa lo mismo que en el caso anterior de TAS, no se garantiza una espera limitada.
- ¿Cómo se consigue?

F&A

La instrucción `fetch-and-increment` o `fetch-and-add`

- aumenta el valor de una variable en la memoria
- y devuelve el resultado
- en una instrucción atómica.

- Con dicha instrucción se puede realizar los protocolos de entrada y salida.
- ¿Cómo?
- También existe en la versión `fetch-and-add` que en vez de incrementar suma un valor dado de forma atómica.

CAS

- La instrucción `compare-and-swap` (CAS) es una generalización de la instrucción `test-and-set`.
- La instrucción trabaja con dos variables, les llamamos `C` (de *compare*) y `S` (de *swap*).
- Se intercambia el valor en la memoria por `S` si el valor en la memoria es igual que `C`.
- Es la operación que se usa por ejemplo en los procesadores de Intel y es la base para facilitar la concurrencia en la máquina virtual de Java 1.5 para dicha familia de procesadores.
- Con CAS se pueden realizar los protocolos de entrada y salida. ¿Cómo?

double CAS

Existen también unas mejoras del `CAS`, llamado *double-compare-and-swap* DCAS (Motorola), que realiza dos CAS normales a la vez, o *double-wide compare-and-swap* (Intel/AMD x86), que opera con dos punteros a la vez para el intercambio, o *single-compare double-swap* (Intel itanium), que compara un valor (puntero) pero escribe dos punteros en memoria adyacente.
El código, expresado a alto nivel, para DCAS sería:

```
if  C1 equal to V1 and C2 equal to V2
  then
    swap S1 and V1
    swap S2 and V2
    return true
  else
    return false
```

- como hemos visto todos los protocolos necesitan variables con acceso atómico en memoría común.
- Tal hecho puede resultar en pérdidas de rendimiento, si existe una jerarquía de memoria con diferentes niveles de cachés.
- Muchos compiladores ofrecen acceso directo a las instrucciones de nivel bajo, por ejemplo la gama de compiladores GCC con sus *atomic builtins*, (https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html)
- Nuevas versiones ya lo hacen diferente (https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/_005f_005fatomic-Builtins.html)
- Ojo con código que tiene que ser compatible entre arquitecturas y versiones.

ABA problem

no se transmite information

- los protocolos de entrada y salida como implementados hasta ahora no transmiten información de un hilo al otro
- en el sentido que si un hilo entra en su sección crítica dos veces
- no puede averiguar si otro hilo ha (o otros hilos han) entrado mientras tanto
- este problema se concoce como ABA-problema (la secuencia *primero A, luego B, después A*, no se puede distinguir del simple hecho *solo A*)
- el problema es, por ejemplo, relevante si se compara solo punteros o referencias para averiguar posibles cambios de estado
- por ejemplo en acciones sobre listas compartidas (secuencias de borrar e insertar)

Un bloqueo se produce cuando un proceso está esperando algo que nunca se cumple.

Ejemplo:

Cuando dos procesos P_0 y P_1 quieren tener acceso simultaneamente a dos recursos r_0 y r_1, es posible que se produzca un bloqueo de ambos procesos. Si P_0 accede con éxito a r_1 y P_1 accede con éxito a r_0, ambos se quedan atrapados intentando tener acceso al otro recurso.

condiciones necesarias

Se tienen que cumplir cuatro condiciones para que sea posible que se produzca un bloqueo entre procesos:

1. los procesos tienen que compartir recursos con exclusión mutua
2. los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro
3. los recursos solo permiten ser usados por menos procesos que lo intentan al mismo tiempo
4. existe una cadena circular entre peticiones de procesos y asignaciónes de recursos

funcionamiento parcial

Un problema adicional con los bloqueos es que es posible que el programa siga funcionando correctamente según la definición, es decir, el resultado obtenido es el resultado deseado, pero algunos de sus procesos están bloqueados durante la ejecución (es decir, se produjo solamente un bloqueo parcial).

técnicas de actuar

Existen algunas técnicas que se pueden usar para que no se produzcan bloqueos:

- Detectar y actuar
- Evitar
- Prevenir

Se implementa un proceso adicional que vigila si los demás forman una cadena circular.
Más preciso, se define el grafo de asignación de recursos:

- Los procesos y los recursos representan los nodos de un grafo.
- Se añade cada vez una arista entre un nodo tipo recurso y un nodo tipo proceso cuando el proceso ha obtenido acceso exclusivo al recurso.
- Se añade cada vez una arista entre un nodo tipo recurso y un nodo tipo proceso cuando el proceso está pidiendo acceso exclusivo al recurso.
- Se eliminan las aristas entre proceso y recurso y al revés cuando el proceso ya no usa el recurso.

actuación

Cuando se detecta en el grafo resultante un ciclo, es decir, cuando ya no forma un grafo acíclico, se ha producido una posible situación de un bloqueo.
Se puede reaccionar de dos maneras si se ha encontrado un ciclo:

- No se da permiso al último proceso de obtener el recurso.
- Sí, se da permiso, pero una vez detectado el ciclo se aborta todos o algunos de los procesos involucrados.

desventaja

Sin embargo, las técnicas pueden dar como resultado que el programa no avance, incluso, el programa se puede quedar atrapado haciendo trabajo inútil: crear situaciones de bloqueo y abortar procesos continuamente.

evitar

El sistema no da permiso de acceso a recursos si es posible que el proceso se bloquee en el futuro.
Un método es el algoritmo del banquero (Dijkstra) que es un algoritmo centralizado y por eso posiblemente no muy practicable en muchas situaciones.
Se garantiza que todos los procesos actuan de la siguiente manera en dos fases:

1. primero se obtiene todos los cerrojos necesarios para realizar una tarea, eso se realiza solamente si se puede obtener todos a la vez,
2. después se realiza la tarea durante la cual posiblemente se liberan recursos que no son necesarias.

prevenir

Se puede prevenir el bloqueo siempre y cuando se consiga que alguna de las condiciones necesarias para la existencia de un bloqueo no se produzca.

1. los procesos tienen que compartir recursos con exclusión mutua
2. los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro
3. los recursos no permiten ser usados por más de un proceso al mismo tiempo
4. existe una cadena circular entre peticiones de procesos y alocación de recursos

prevenir exclusión mutua

los procesos tienen que compartir recursos con exclusión mutua:

- No se de a un proceso directamente acceso exclusivo al recurso, si no se usa otro proceso que realiza el trabajo de todos los demás manejando una cola de tareas (por ejemplo, un demonio para imprimir con su cola de documentos por imprimir).

prevenir accesos consecutivos

los procesos quieren acceder a un recurso más mientras ya tienen acceso exclusivo a otro:

- Se exige que un proceso pida todos los recursos que va a utilizar al comienzo de su trabajo

prevenir uso único

los recursos no permiten ser usados por más de un proceso al mismo tiempo:

- Se permite que un proceso aborte a otro proceso con el fin de obtener acceso exclusivo al recurso. Hay que tener cuidado de no caer en *livelock*
- (Separar lectores y escritores alivia este problema también.)

existe una cadena circular entre peticiones de procesos y alocación de recursos:

- Se ordenan los recursos línealmente y se fuerza a los procesos que accedan a los recursos en el orden impuesto. Así es imposible que se produzca un ciclo.

Bloqueo en Java

Un ejemplo de un bloqueo en Java muestra el siguiente trozo de código, incluso si se asume que un hilo ya está durmiendo. ¿Por qué?

```
hilo0:

synchronized(A) {
  ...
  synchronized(B) {
    ...
    A.notify();
    B.wait();
  }
}
```

```
hilo1:

synchronized(B) {
  ...
  synchronized(A) {
    ...
    B.notify();
    A.wait();
  }
}
```

seguridad y vivacidad/viveza

Un programa concurrente puede fallar por varias razones, las cuales se pueden clasificar entre dos grupos de propiedades:

seguridad: Esa propiedad indica que no está pasando nada malo en el programa, es decir, el programa no ejecuta instrucciones que no deba hacer ("safety property").

vivacidad: Esa propiedad indica que está pasando continuamente algo bueno durante la ejecución, es decir, el programa consigue algún progreso en sus tareas o en algún momento en el futuro se cumple una cierta condición ("liveness property").

propiedades de seguridad

Las propiedades de seguridad suelen ser algunas de las invariantes del programa que se tienen que introducir en las comprobaciones del funcionamiento correcto.

Corrección: El algoritmo usado es correcto.

Exclusión mutua: El acceso con exclusión mutua a regiones críticas está garantizado

Sincronización: Los procesos cumplen con las condiciones de sincronización impuestos por el algoritmo

Interbloqueo: No se produce ninguna situación en la cual todos los procesos participantes quedan atrapados en una espera a una condición que nunca se cumpla.

- Un proceso puede "morirse" por inanición (*starvation*), es decir, un proceso o varios procesos siguen con su trabajo pero otros nunca avanzan por ser excluidos de la competición por los recursos (por ejemplo en Java el uso de `suspend()` y `resume()` no está recomendado por esa razón).
- Existen problemas donde la inanición no es un problema real o es muy improbable que ocurra, es decir, se puede aflojar las condiciones a los protocolos de entrada y salida.

propiedades de vivacidad

Bloqueo activo: Puede ocurrir el caso que varios procesos están continuamente competiendo por un recurso de forma activa, pero ningúno de ellos lo consigue (“livelock”).

Cancelación: Un proceso puede ser terminado desde fuera sin motivo correcto, dicho hecho puede resultar en un bloqueo porque no se ha considerado la necesidad que el proceso debe realizar tareas necesarias para liberar recursos (por ejemplo, en Java el uso del `stop()` no está recomendado por esa razón).

Espera activa: Un proceso está comprobando continuamente una condición malgastando de esta manera tiempo de ejecución del procesador.

justicia entre procesos

Cuando los procesos compiten por el acceso a recursos compartidos se pueden definir varios conceptos de justicia, por ejemplo:

justicia débil: si un proceso pide acceso continuamente, le será dado en algún momento,

justicia estricta: si un proceso pide acceso infinitamente veces, le será dado en algún momento,

espera limitada: si un proceso pide acceso una vez, le será dado antes de que otro proceso lo obtenga más de una vez,

espera ordenada en tiempo: si un proceso pide acceso, le será dado antes de todos los procesos que lo hayan pedido más tarde.

- Los dos primeros conceptos son conceptos teóricos porque dependen de términos *infinitamente* o *en algún momento*, sin embargo, pueden ser útiles en comprobaciones formales.
- En un sistema distribuido la ordenación en tiempo no es tan fácil de realizar dado que la noción de tiempo no está tan clara.
- Normalmente se quiere que todos los procesos manifesten algún progreso en su trabajo (pero en algunos casos inanición controlada puede ser tolerada).

- El algoritmo de Dekker y sus parecidos provocan una espera activa de los procesos cuando quieren acceder a un recurso compartido. Mientras están esperando a entrar en su región crítica no hacen nada más que comprobar el estado de alguna variable.
- Normalmente no es aceptable que los procesos permanezcan en estos bucles de espera activa porque se está gastando potencia del procesador inútilmente.
- Un método mejor consiste en suspender el trabajo del proceso y reanudar el trabajo cuando la condición necesaria se haya cumplido. Naturalmente dichos técnicas de control son más complejas en su implementación que la simple espera activa.

evitar espera infinita o inanición de procesos

- Se implementa, por ejemplo, el acceso a recursos compartidos siguiendo un orden FIFO, es decir, los procesos tienen acceso en el mismo orden en que han pedido vez.
- Se asignan prioridades a los procesos de tal manera que cuanto más tiempo un proceso tiene que esperar más alto se pone su prioridad con el fin que en algún momento su prioridad sea la más alta.
- ¡Ojo! ¿Qué se hace si todos tienen la prioridad más alta?
- Existen más técnicas... ¿Cuáles?

El problema del productor y consumidor es un ejemplo clásico de programa concurrente y consiste en la situacion siguiente:
de una parte se produce algún producto (datos en nuestro caso) que se coloca en algún lugar (una cola en nuestro caso) para que sea consumido por otra parte. Como algoritmo obtenemos:

```
producer:
  forever
    produce(item)
    place(item)
```

```
consumer:
  forever
    take(item)
    consume(item)
```

Queremos garantizar que el consumidor no coja los datos más rápido de lo que los está produciendo el productor. Más concreta:

1. el productor puede generar sus datos en cualquier momento, pero no debe producir nada si no lo puede colocar
2. el consumidor puede coger un dato solamente cuando hay alguno
3. para el intercambio de datos se usa una estructura de datos compartida a la cual ambos tienen acceso,
4. si se usa una cola se garantiza un orden temporal
5. ningún dato no está consumido una vez haber sido producido (por lo menos se descarta...)

Si la cola puede crecer a una longitud infinita (siendo el caso cuando el consumidor consume más lento de lo que el productor produce), basta con la siguiente solucion que garantiza exclusion mutua a la cola:

```
producer:
  forever
    produce(item)
    place(item)
    itemsReady.signal()
```

```
consumer:
  forever
    itemsReady.wait()
    take(item)
    consume(item)
```

donde `itemsReady` es un semáforo general que se ha inicializado al principio con el valor `0`.

más participantes

Queremos ampliar el problema introduciendo más productores y más consumidores que trabajen todos con la misma cola. Para asegurar que todos los datos estén consumidos lo más rápido posible por algún consumidor disponible tenemos que proteger el acceso a la cola con un semáforo binario (llamado `mutex` abajo):

```
producer:
  forever
    produce(item)
    mutex.wait()
    place(item)
    mutex.signal()
    itemsReady.signal()
```

```
consumer:
  forever
    itemsReady.wait()
    mutex.wait()
    take(item)
    mutex.signal()
    consume(item)
```

cola finita

- Normalmente no se puede permitir que la cola crezca infinitamente, es decir, hay que evitar produccion en exceso también.
- Como posible solucion introducimos otro semáforo general (llamado `spacesLeft`) que cuenta cuantos espacios quedan libres en la cola.
- Se inicializa el semáforo con la longitud máxima permitida de la cola.
- Un productor queda bloqueado si ya no hay espacio en la cola y un consumidor señala su consumision.

cola finita

```
producer:
  forever
    spacesLeft.wait()
    produce(item)
    mutex.wait()
    place(item)
    mutex.signal()
    itemsReady.signal()
```

```
consumer:
  forever
    itemsReady.wait()
    mutex.wait()
    take(item)
    mutex.signal()
    consume(item)
    spacesLeft.signal()
```

- En un sistema con multiples productores y/o consumidores, puede ser difícil establecer un orden temporal con una semántica adecuada.
- Se puede aflojar la condición de usar una cola, y usar estructuras de datos que permitan más concurrencia.
- Un ejemplo simple serían vectores de contenedores inspeccionados en orden cíclico por los productores y consumidores.
- O se usa estructuras de datos concurrentes ya disponibles en las librerías de programación.

comunicación

Programas concurrentes o/y distribuidos necesitan algún tipo de comunicación entre los procesos.
Hay dos razones principales:

1. Los procesos compiten para obtener acceso a recursos compartidos.
2. Los procesos quieren intercambiar datos.

Para cualquier tipo de comunicación hace falta un método de sincronización entre los procesos que quieren comunicarse entre ellos.
Al nivel del programador existen tres variantes como realizar las interacciones entre procesos:

1. Usar memoria compartida (*shared memory*).
2. Mandar mensajes (*message passing*).
3. Lanzar procedimientos remotos (*remote procedure call* RPC).