udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

Interbloqueo (continuación)

Adín Ramírez
adin.ramirez@mail.udp.cl

Sistemas Operativos (CIT2003-1)
1er. Semestre 2015

Adaptado de Dave Eckhardt

Métodos alternativos

- Prevenir
  - Establecer alguna regla para evitar alguna de las propiedades del interbloqueo
  - Cualquier aplicación funcional debe de estar libre de interbloqueos
- Evitar
  - Procesos declaran patrones de uso de recursos
  - El administrador de peticiones evita estados “inseguros”
- Detectar y recuperar
  - Limpiar cuando paso algo malo

¿Es la prevención satisfactoria?

- La prevención utiliza reglas
  - No lo podemos forzar a todos los recursos —¿qué recursos no se pueden compartir?
  - Molesto
    - Ineficiente si existen varios recursos que se deben poseer
    - La obtención obligatoria de seguros en cierto orden puede producir inanición
    - **Muchas** oportunidades para caer en inanición
- ¿Necesitamos este tipo de reglas tan estrictas?
  - ¿Podríamos ser más situacionales?

Objetivos de la clase

- Evitar interbloqueos
- Detectar y recuperarnos de los interbloqueos

Procesos pre declaran patrones de uso de los recursos

- Podemos enumerar todos los caminos posibles para alojar espacio
  - Solicitar R_1, solicitar R_2, liberar R_1, solicitar R_3
  - Solicitar R_1, solicitar R_3, liberar R_3, solicitar R_2
- Más fácil: declaramos la cantidad máxima de recursos a utilizar
  - Nunca voy a necesitar más de 7 discos y 1 impresora

Procesos proceden hasta completar

- No nos aferramos a los recursos por siempre
  - Liberamos los recursos
- Completar en un tiempo razonable
  - Entonces, está bien si demoramos a P_2 hasta que P_1 termine
  - Evitaremos este tipo de técnicas

Secuencia de ejecución segura

Definición

$(P_1, P_2, \ldots, P_n)$ es una **secuencia segura** si cada proceso P_i puede ser satisfecho utilizando

- los recursos actuales disponibles, F, más
- los recursos alojados por P_1, P_2, $\ldots$, P_i

- P_i tiene una espera acotada por la secuencia
  - P_1 se ejecuta hasta completarse
  - P_2 puede completarse con $F + R(P_1) + R(P_2)$
  - P_3 puede completarse con $F + R(P_1) + R(P_2) + R(P_3)$
- Entonces, P_i no esperará por siempre (no hay un ciclo de espera, entonces no hay deadlock)

Estado seguro

Definición

El sistema se encuentra en un **estado seguro** si y solo si existe al menos una secuencia segura

- Peor caso
  - Cada proceso pregunta por cada recurso simultáneamente
  - Solución: seguimos una secuencia segura (ejecutamos los procesos en serial)
    - Lento, pero no como un interbloqueo!
- Ejecución serial es el peor caso, pero no es típico
  - Usualmente los procesos se ejecutan en paralelo

Administrador de peticiones

Solución ingenua

- Otorgar el recurso a una petición si hay suficientes recursos libres ahora
- Sino, decirle al proceso que solicitó que espere
  - Mientras mantiene los recursos (y que no podemos interrumpir ni obtener ☹)
- La receta perfecta para un interbloqueo

Segunda versión

- Otorgar el recurso a una petición si
  - hay suficientes recursos libres ahora, y
  - habrán suficientes recursos libres
    - Para que algún proceso obtenga el resto de sus recursos, complete, y pueda liberar los recursos que mantiene
    - Y luego otro
    - Y luego nosotros
- Sino, decirle al proceso que solicitó que espere
  - Mientras mantiene un conjunto menor de recursos
    - que ya probamos que está bien que mantenga, porque los otros procesos no los necesitan para terminar

Ejemplo

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7
Sistema	12	3	–

- Max = necesita
- Tiene = obtenidos
- Nec. = Max - Tiene
- Reservados 9 $(5 + 2 + 2)$
- Libres 3 (los del sistema)

- Es un estado seguro
- ¿Cómo mostramos que es un estado seguro?

$P_1 : 2 \Rightarrow 4$

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7
Sistema	12	3	–

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	4	0
P_2	9	2	7
Sistema	12	1	–

- El sistema entrega recursos a P_1

P_1 termina

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	4	0
P_2	9	2	7
Sistema	12	1	–

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_2	9	2	7
Sistema	12	5	–

- P_1 entrega recursos al sistema

$P_0 : 5 \Rightarrow 10$

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_2	9	2	7
Sistema	12	5	–

Proceso	Max	Tiene	Nec.
P_0	10	10	0
P_2	9	2	7
Sistema	12	0	–

- El sistema entrega recursos a P_0
- ¿Por qué no a P_2?

P_0 termina

Proceso	Max	Tiene	Nec.
P_0	10	10	0
P_2	9	2	7
Sistema	12	0	–

Proceso	Max	Tiene	Nec.
P_2	9	2	7
Sistema	12	10	–

- P_0 entrega los recursos al sistema
- "Ejecutar P_1, P_0, P_2" era una secuencia segura
- Entonces el sistema esta en un estado seguro

Ejemplo

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7
Sistema	12	3	–

- Max = necesita
- Tiene = obtenidos
- Nec. = Max - Tiene
- Reservados 9 $(5 + 2 + 2)$
- Libres 3 (los del sistema)

- ¿Podemos hacer que P_2 obtenga recursos?
- Muy inseguro... ☹

$P_2 : 2 \Rightarrow 3$

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7
Sistema	12	3	–

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	2	2
P_2	9	3	7
Sistema	12	2	–

- El sistema entrega recursos (1) a P_2
- Ahora, solo P_1 se puede ejecutar sin esperar

$P_1 : 2 \Rightarrow 4$

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	2	2
P_2	9	3	7
Sistema	12	2	–

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	4	0
P_2	9	3	7
Sistema	12	0	–

- El sistema entrega recursos a P_1

P_1 termina

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_1	4	4	0
P_2	9	3	7
Sistema	12	0	–

Proceso	Max	Tiene	Nec.
P_0	10	5	5
P_2	9	3	7
Sistema	12	4	–

- P_1 entrega recursos al sistema
- P_0 y P_2 necesitan más de 4 recursos cada uno
- Si alguno obtiene los recursos, espera que el otro entregue algunos
- Si alguno solicita más de 4 recursos, tenemos un deadlock!

Q ¿Es inevitable el deadlock?

Q ¿No vimos otra posible secuencia, aparte de iniciar con P_1?

Puntos claves

- Estado seguro
  - Existe una secuencia segura
  - Probarlo encontrando una
- Estado inseguro: no existe una secuencia segura
- Un estado inseguro puede ser no fatal
  - El proceso puede terminar antes
  - Procesos pueden no necesitar más recursos (ahora, y ¿en el siguiente intento?)

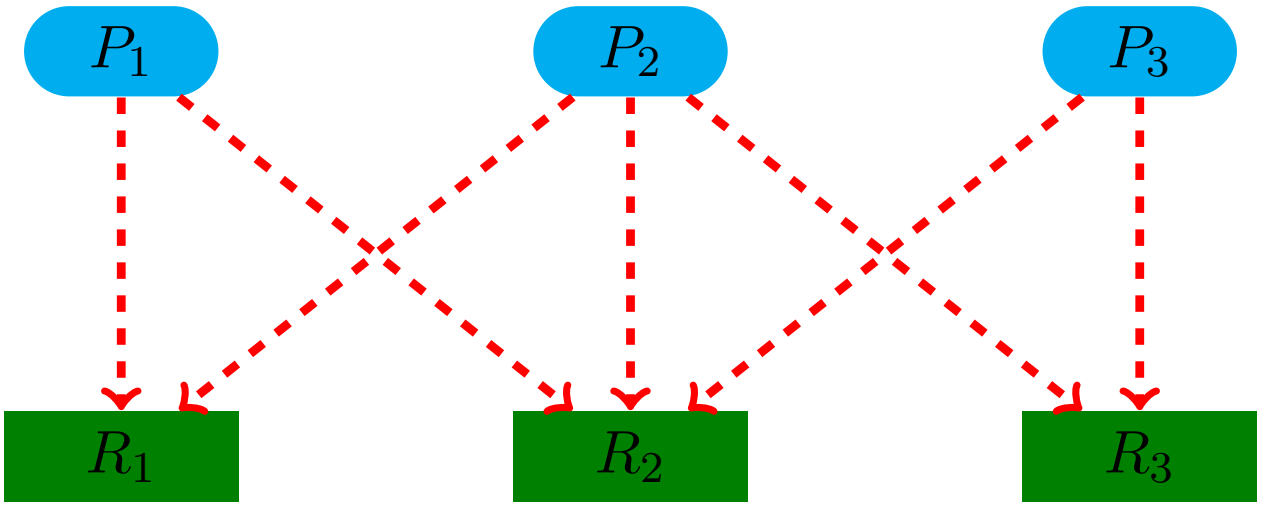
Compromisos

- Permitir solo estados seguros si son más flexibles que la prevención
  - Algunos de las reglas son convenientes de seguir
- Pero rechazar todos los estados inseguros reducen la eficiencia
  - El sistema puede entrar a un estado inseguro y retornar a un estado seguro
  - ¿Qué tan seguido el sistema se retirará del desastre?

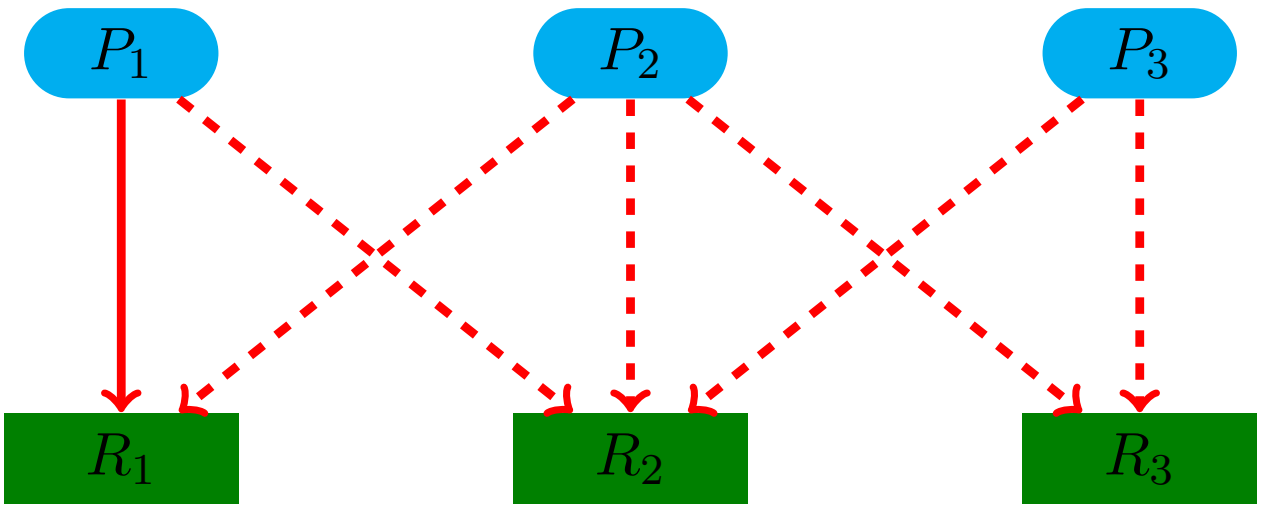
Recursos únicos

- Recursos únicos en lugar de múltiples instancias
  - ¿Cómo luce el grafo?
- Tres tipos de arcos
  - Reclamar (una petición futura)
  - Solicitar
  - Asignar

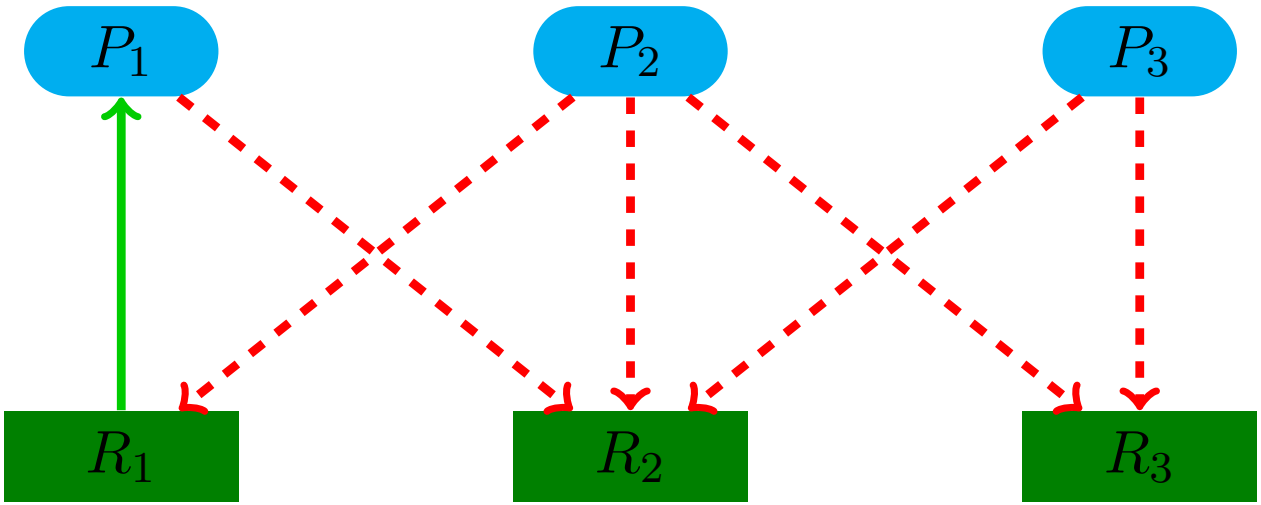
Arcos de reclamos (solicitudes futuras)



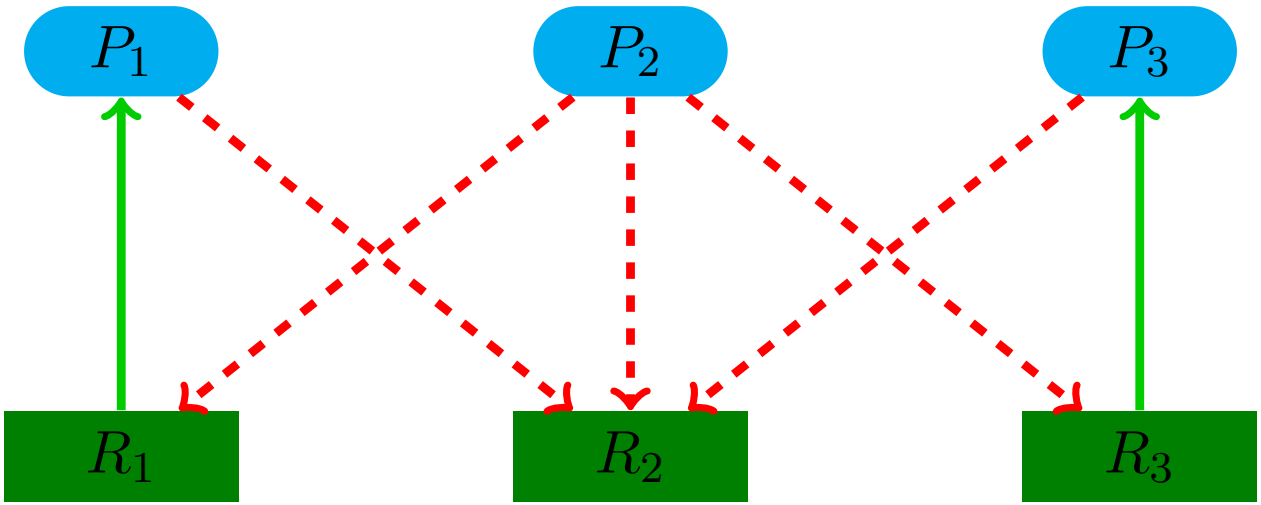
Un reclamo se convierte en solicitud



Una solicitud se convierte en una adquisición



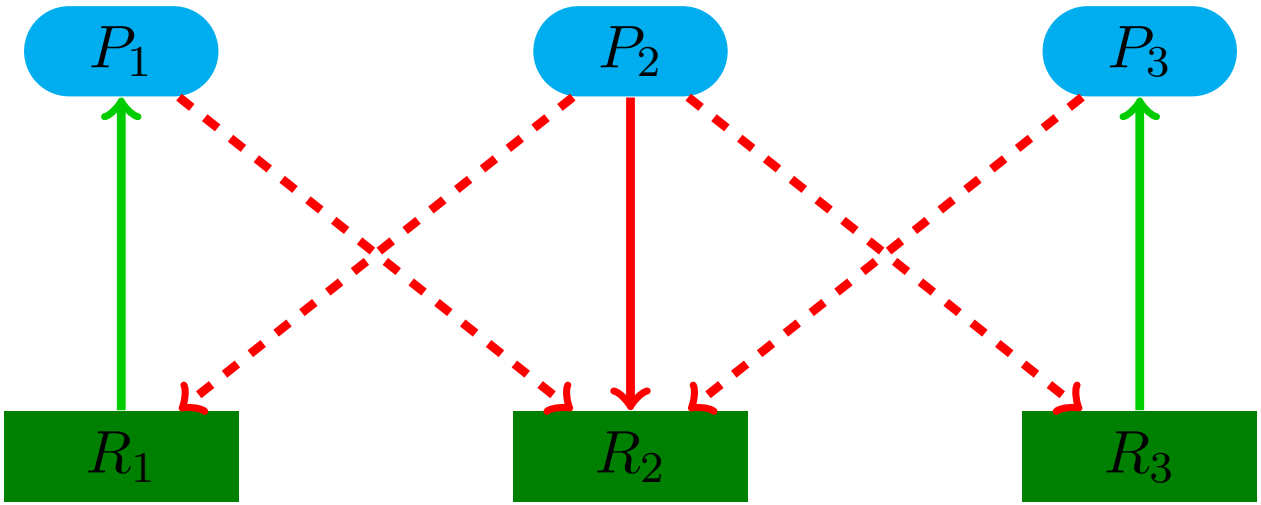
Estado seguro: no ciclos



¿Qué peticiones son seguras?

- Pretendamos satisfacer una solicitud
- Observemos el grafo en caso de ciclos

¿Solicitud peligrosa?



¿Son los ciclos de pretenciones fatales?

- ¿Debemos preocuparnos de todos los ciclos?
  - Nadie está esperando en un ciclo construido sobre supuestos
  - Muchos arcos son posibles solicitudes
  - No tenemos un interbloqueo aún
- ¿Es seguro?

¿Son los ciclos de pretenciones fatales?

- Ningún proceso puede, sin esperar, obtener el conjunto máximo de recursos declarados
- Entonces, ningún proceso puede obtener, completar, y liberar (sin esperar)
- Cualquier nueva solicitud puede generar un ciclo
  - Entonces no es seguro, debemos tener cuidado
- ¿Qué hacemos?
  - No entreguemos ninguna solicitud
  - Bloqueamos el proceso
    - Ahora
    - Antes de que obtenga el recursos
    - En lugar de bloquearlo después, cuando ya lo tenga en su poder

Múltiples instancias de recursos

- Ejemplo
  - N recursos intercambiables
  - Pueden representarse por N nodos de recursos
  - Computación innecesaria
- Modelo de linea de crédito
  - El banco asigna una cantidad máxima a prestar (límite de crédito)
  - Los negocios pagan interés en la cantidad prestada actualmente

Evitar bancarota

- El banco está bien mientras se encuentre en una secuencia segura
- Una compañía puede
  - Prestar hasta su límite de crédito
  - Irle bien
  - Pagar su deuda al banco
- Y otras compañías pueden hacer lo mismo

No existe una secuencia segura

- La compañía trata de prestar hasta el límite
  - El banco no tiene efectivo
  - Compañía C_1 debe esperar por el dinero que C_2 prestó
  - Tal vez C_2 debe esperar por una parte del dinero que C_1 prestó
- En la vida real
  - C_1 no puede pagar la planilla
  - C_1 va a la quiebra
  - El préstamo no se paga completo
    - Podemos modelarlo como una inanición infinita

Algoritmo del banquero

```
int cash;
int limit[N];     // crédito
int out[N];       // prestado
bool done[N];     // temporal global
int future;       // temporal global

int progressor (int cash){
  for (int i = 0; i < N; i++)
    if (!done[i] && cash >= limit[i] - out[i])
      return i;
  return -1;
}
```

Algoritmo del banquero

```
bool is_safe(void) {
  future = cash;
  for(int i = 0; i < N; i++) done[i] = false;

  while ( (int p=progressor(future)) > 0 ) {
    future += out[p];
    done[p] = true;
  }
  return (done[0..N] == true); // devolvemos la
      comparación del estado ideal
}
```

Posibles problemas

- ¿Qué pasa si escogemos los que progresan en un orden equivocado?
- ¿Podemos prestar más dinero a una compañía?
  - Pretendamos que lo hacemos
    - Actualizamos `cash` y `out[i]`
  - ¿Estamos en un estado seguro?
    - Si: prestemos el dinero
    - No: deshagamos el estado antes de pretender, y durmamos
- Versión multi-recurso
  - Generaliza fácilmente a N tipos de recursos independientes
  - Ver el texto

Evitar el interbloqueo

- Buenas noticias: no hay interbloqueo
  - \+ No hay reglas estáticas sobre las solicitudes de recursos
  - \+ Los alojamientos son flexibles acorde al estado del sistema
- Malas noticias
  - – Los procesos deben de declarar el uso máximo de recursos
  - – La acción de evitar es conservadora
    - Muchos estados inseguros son casi seguros
    - El rendimiento del sistema se ve reducido —dormimos de más

Enfoque

- No hay que ser paranoico
  - No denegar solicitudes que puedan producir problemas
    - Algún día algo malo puede pasar
    - La mayoría de las veces todo saldrá bien
- Aún los paranoicos tienen enemigos
  - A veces los interbloqueos sucederán
  - Necesitamos un plan para encontrarlo
  - Necesitamos una política para poder reaccionar
  - Alguien tiene que intentarlo otra vez

Ideas claves

- Ocasionalmente escanear por ciclos de espera
- Es caro
  - Debemos bloquear toda actividad: peticiones, alojar, y desalojar
  - Tomar un mutex global: variable global de concurrencia
  - Detectar ciclos nos toma $O(N^2)$

Política de escaneo

- Balancear el rendimiento
  - Escanear muy seguido: el sistema se vuelve muy lento
  - Escanear antes de dormir: solo en sistemas pequeños
  - Escanar en raras ocasiones: el sistema se vuelve (extremadamente) lento
- Candidatos de política
  - Escanear cada `<intervalo>`
  - Escanear cuando el CPU esté desocupado

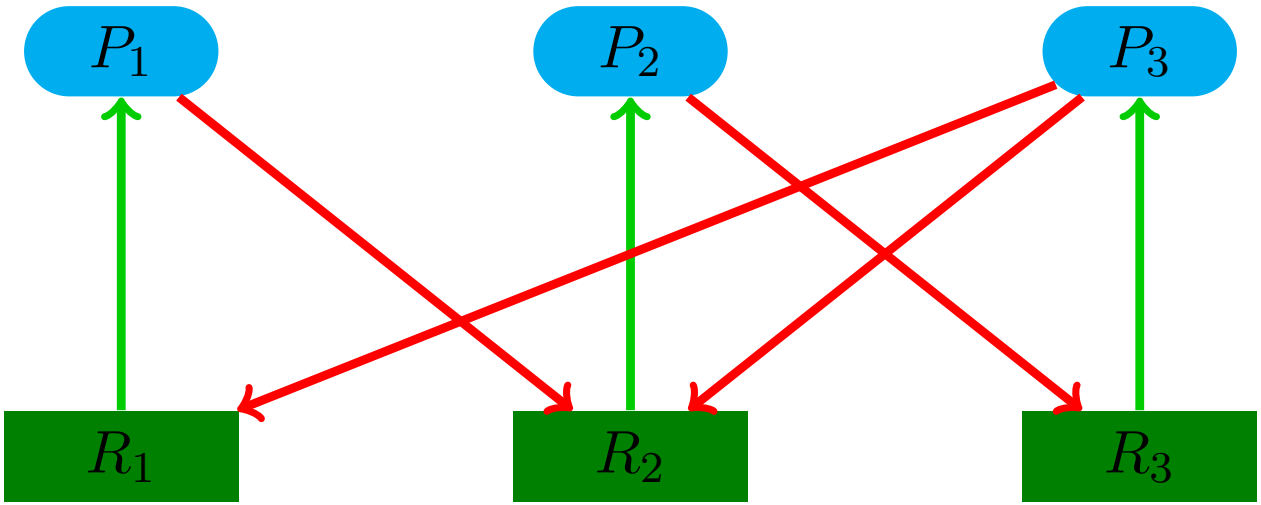
Algoritmos

- Detectar: recursos únicos
  - Buscar por ciclos en el grafo de recursos
- Detectar: múltiples instancias de recursos
  - Variación del algoritmo del Banquero
- Encontramos un deadlock, ¿que hacemos ahora?
  - Abortar
  - Interrumpir

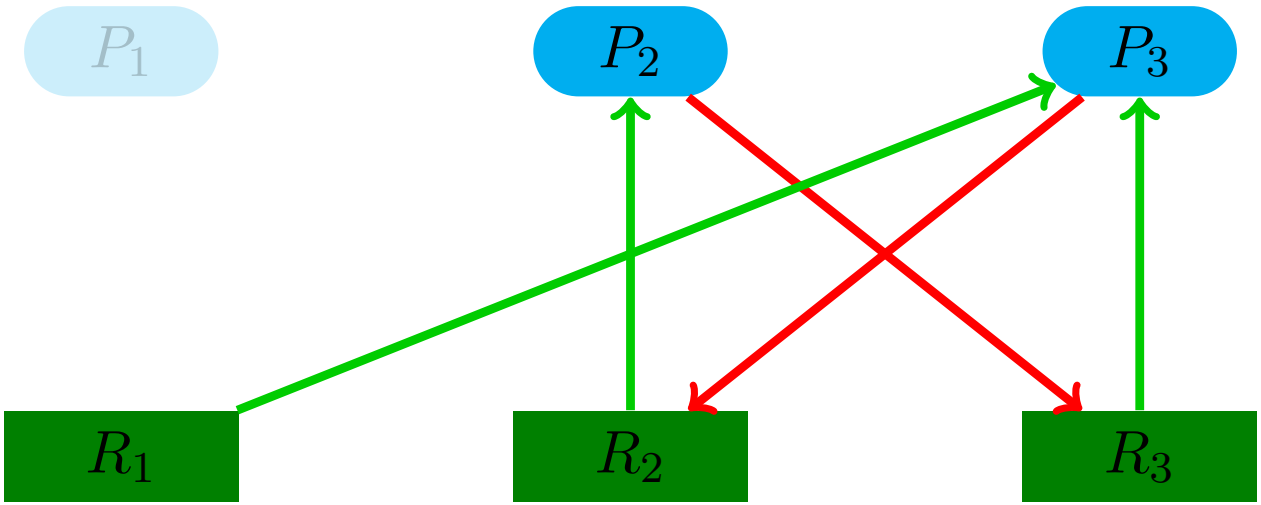
Abortar

- Sacar a los procesos del sistema
- ¿A todos los procesos del ciclo?
  - Simple, y una política sin culpa
  - Mucha re-ejecución posterior
- ¿Sólo un proceso del ciclo?
  - ¿Cuál?
  - Según prioridad, trabajo restante, trabajo para limpiar
  - Lo más probable es que luego de sacarlo del sistema se creará otro ciclo: ¿re escanear?

Abortemos uno



Abortemos uno



¿Podemos hacerlo mejor?

- Abortar procesos no es deseable
  - Re ejecutar los procesos es caro
  - Tareas de larga ejecución pueden nunca completarse
  - Inanición

Interrumpir y obtener recursos

- Decirle a un proceso: tiempo de dar, y no tomar
- Acciones
  - No podemos solo reintentar la solicitud
  - Debemos liberar los otros recursos (que ya tenemos), e intentar después
  - Una liberación forzosa de recursos requiere un *rollback*
- Políticas: ¿qué proceso pierde?
  - El más bajo número: inanición

Interbloqueos

- Un interbloqueo es
  - Un conjunto de procesos
  - Cada uno esperando por algo que otro proceso tiene
- Cuatro ingredientes para un *deadlock*
- Tres enfoques

Enfoques

- Prevención: establecer una regla para evitar
  - Exclusión mutua (no muy probable, depende del tipo de recurso)
  - Obtener y esperar (tal vez)
  - No interrupciones (tal vez)
  - Espera circular (popular, pero no efectiva)
- Una decisión de arquitectura puede excluir alguans características, algoritmos, etc.

Enfoques

- Evitar: mantenerse fuera del peligro
  - Requiere una pre declaración de los patrones de uso
  - No todo el peligro se convierte en problemas
- Detectar y recuperar
  - Frecuencia para escanear: un balance delicado
  - Interrumpir es difícil, y sucio
- Reiniciar
  - ¿Qué es lo que se colgó? ¿Todo el sistema?

Inanición

- Inanición no es sinónimo de interbloqueo
  - Comparten la propiedad de que al menos un proceso no está progresando
  - Con la inanición existe una calendarización donde algunos procesos progresan
- Inanición es un peligro global
- Las soluciones a un interbloqueo nos dejan vulnerables a la inanición
  - Si somos el tipo de aplicación que puede ser impactada por el problema, no estamos mejor que si estuvieramos en un deadlock