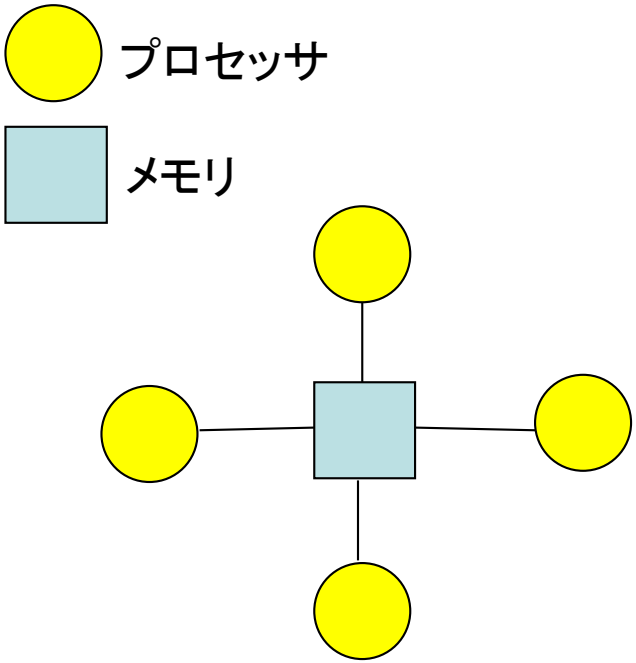
# 共有メモリ型計算機

慶應義塾大学理工学部
天野英晴
hunga@am. ics. keio. ac. jp

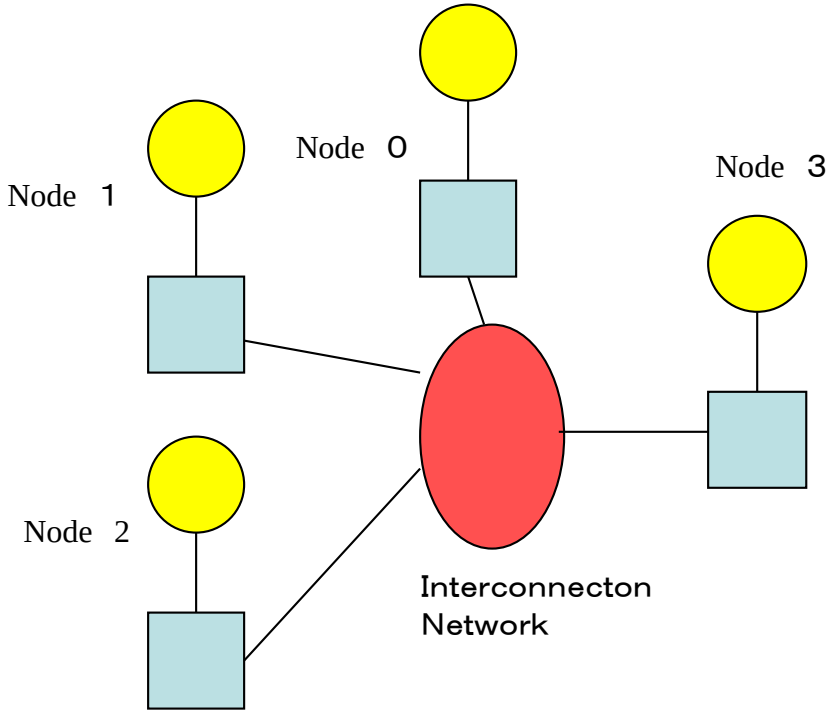
# 共有メモリ型計算機

- 共有するメモリに対する読み書きでデータ交換を行う
- 並列OSが動作する→従来のコンピュータと同じに見える
  - プログラムを高速化するには並列化が必要
    - プログラマによる明示的並列化（OpenMPなど）
    - 並列化コンパイラ
- 理解のポイント
  - メモリの構造をどうするか？
  - 同期をどうするか？
  - キャッシュの一貫性をどうするか？

# 1．メモリの構造をどうするか？
# 集中メモリ型と分散メモリ型

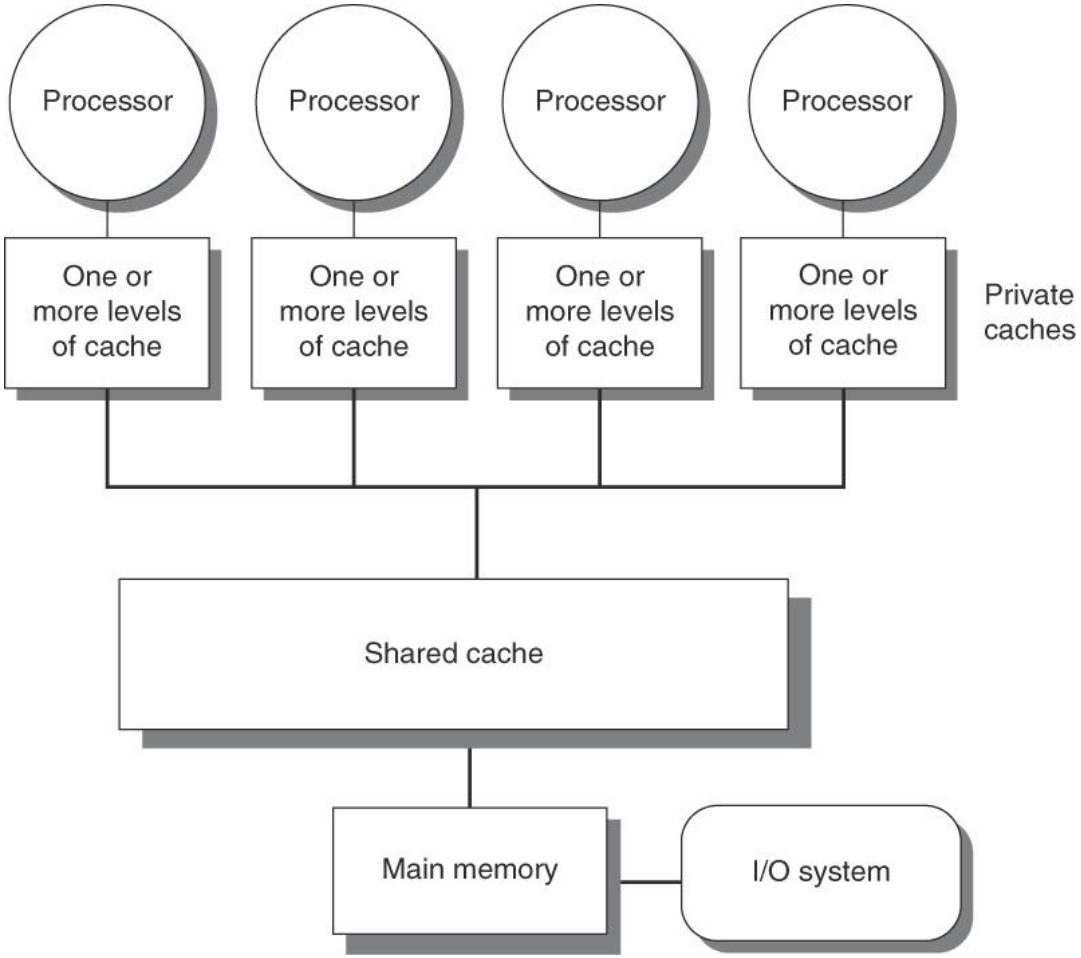



メモリが一か所に集中
UMA(Uniform memory access model)
いわゆるマルチコア



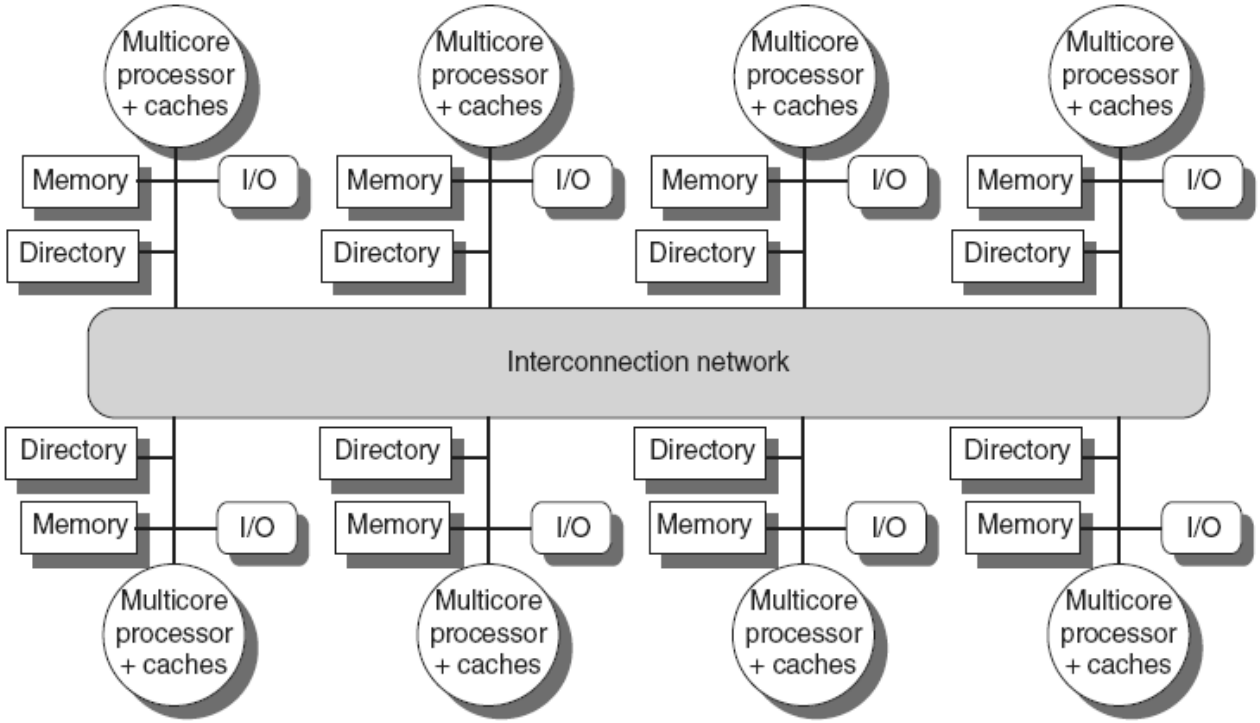

メモリが分散
NUMA(Non-Uniform memory access model)

# 典型的な集中メモリ型


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

# 典型的な分散メモリ型

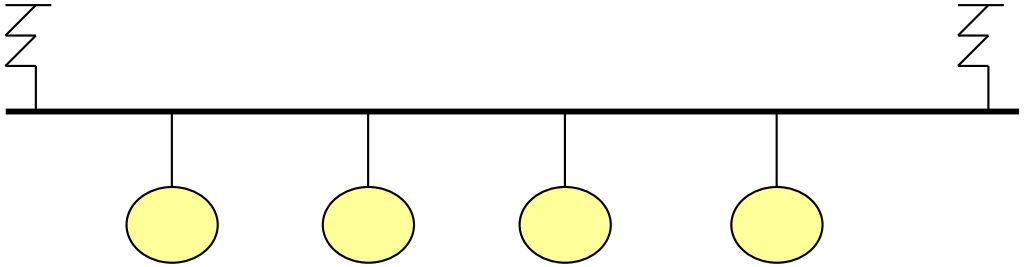
**IBM Power 7**
**AMD Opteron 8430**



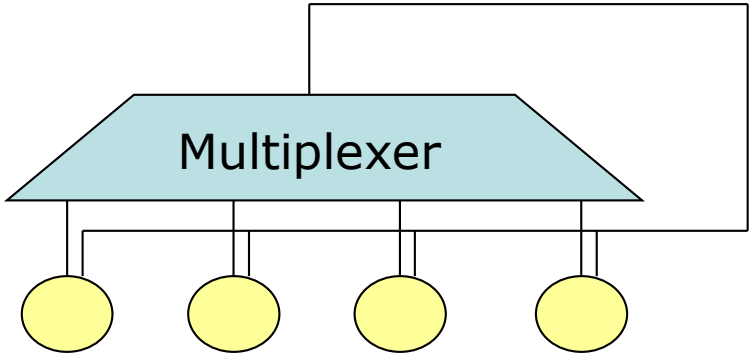
# 共有メモリをどう繋ぐか？

- 集中メモリ型
  - 共有バス:
    - 一度に一つのプロセッサのみ転送可能
    - ボトルネックになるので小規模なシステムに限られる
    - スヌープキャッシュに利用できる→後程解説
  - クロスバスイッチ
    - コストが大きい
- 分散メモリ型
  - 直接網:メッシュ、ハイパーキューブ
  - 間接網:Fat Tree、Dragonfly
  - この辺の話は次々回に解説

# 共有バスの構造



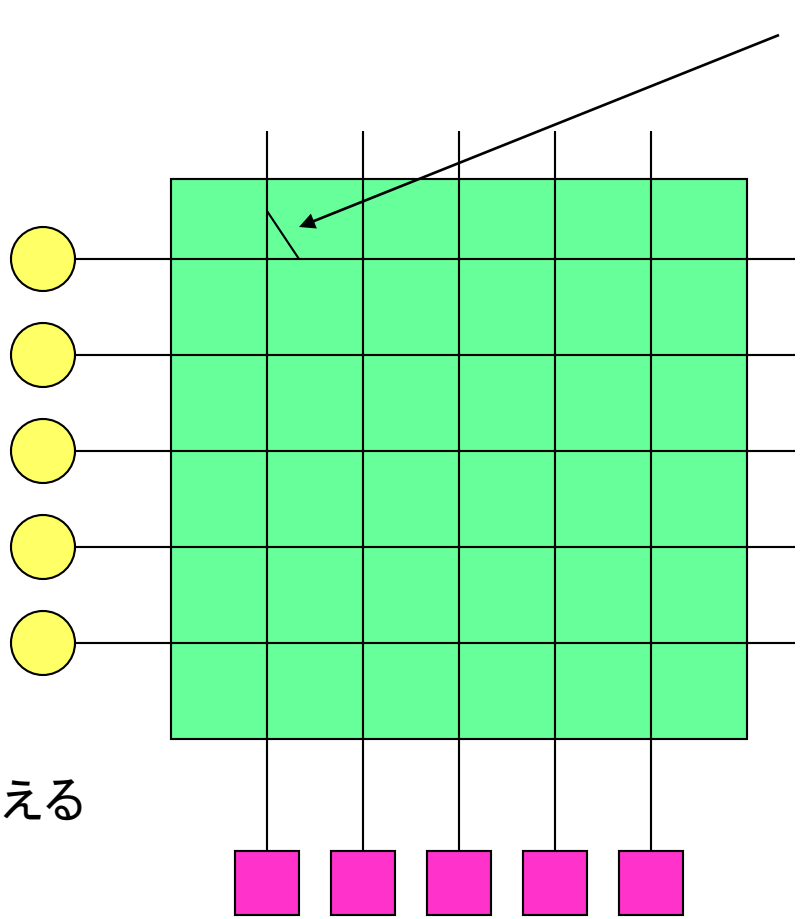
バスの基本は共通の信号線、しかしこれはもうあまり使われない。




バスの本質は、唯一のモジュールがデータを送信できること

色々な形の共有バスがあり得る

# クロスバスイッチ




バスの拡張として考える
ことができる

同時に複数のプロセッサ
メモリ間が交信できる

# 2. 同期の必要性

- あるプロセッサが共有メモリに書いても、別のプロセッサにはそのことが分からない。
- 同時に同じ共有変数に書き込みすると、結果がどうなるか分からない。
- そもそも共有メモリって結構危険な代物
- 多くのプロセッサが並列に動くには何かの制御機構が要る

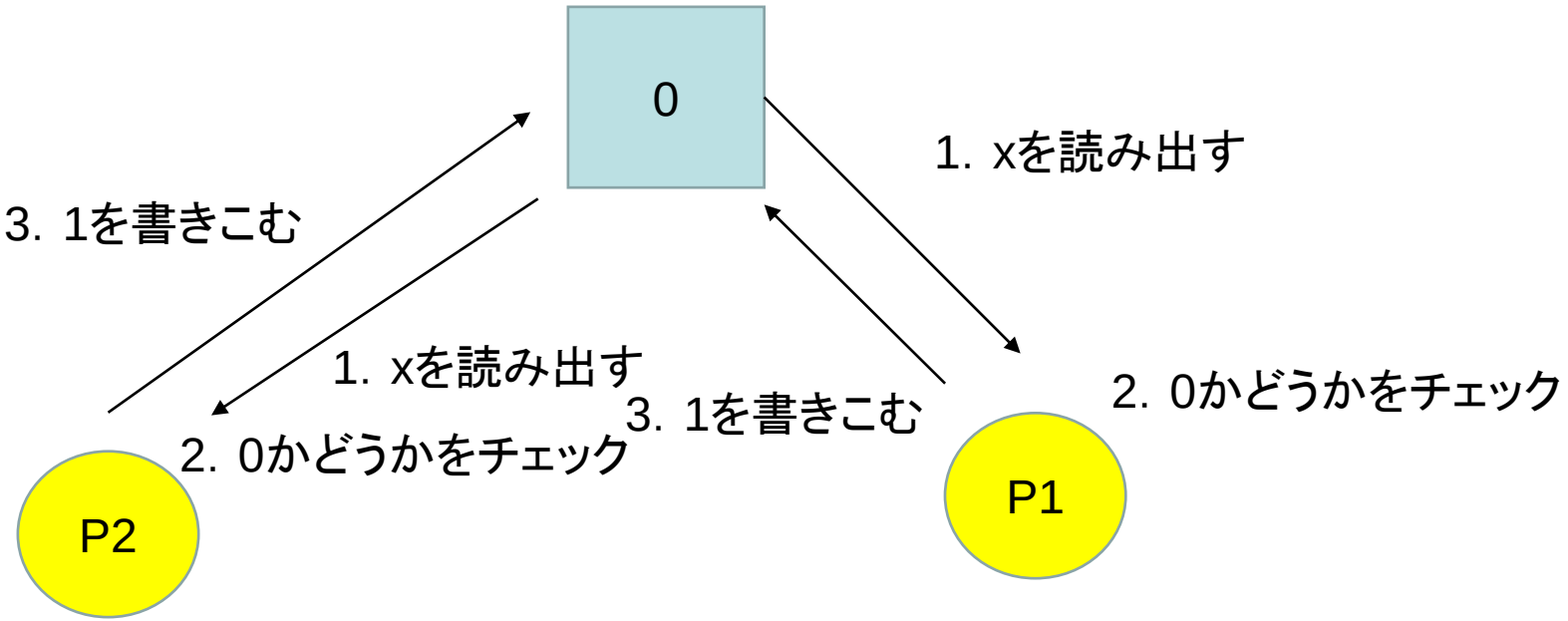
- 不可分命令、同期用メモリ、バリア同期機構

# 排他制御

- プリンタがあり、一度に一つのプロセッサしか使えない。同時に要求があった場合に一人を選びたい。
- アイディア:
  - 変数xを0に初期化しておく。
  - xを読んだプロセッサは、0ならば素早く1を書き込み、プリンタを利用。終わったら0を書き込む。
  - 1を読み込んだプロセッサは0を読み込めるまで繰り返し読み続ける（Busy Waiting)

→しかしこれはうまく行かない。なぜか？

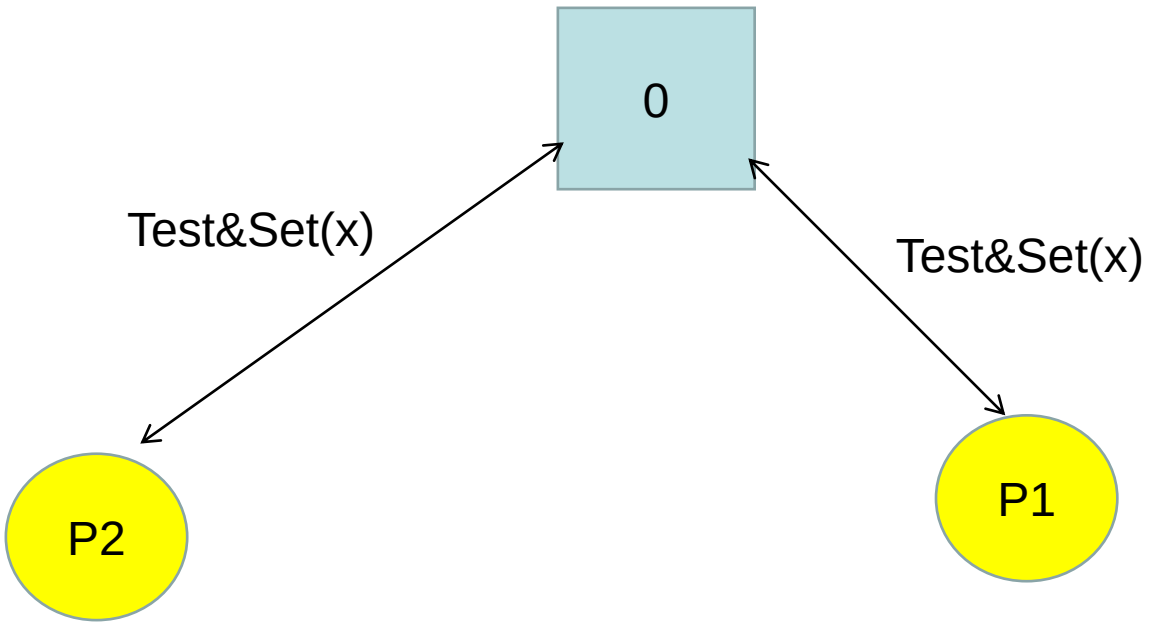
# P1とP2が同時に変数を読んだら？




P1がxを読んで0かどうかをチェックしている間に、P2がxを読み出すかもしれない→P1,P2共に0を取ることができる。

読む操作と書く操作を不可分（Atomic／Indivisible)に行う命令が必要
→不可分命令

# Test & Set (x)



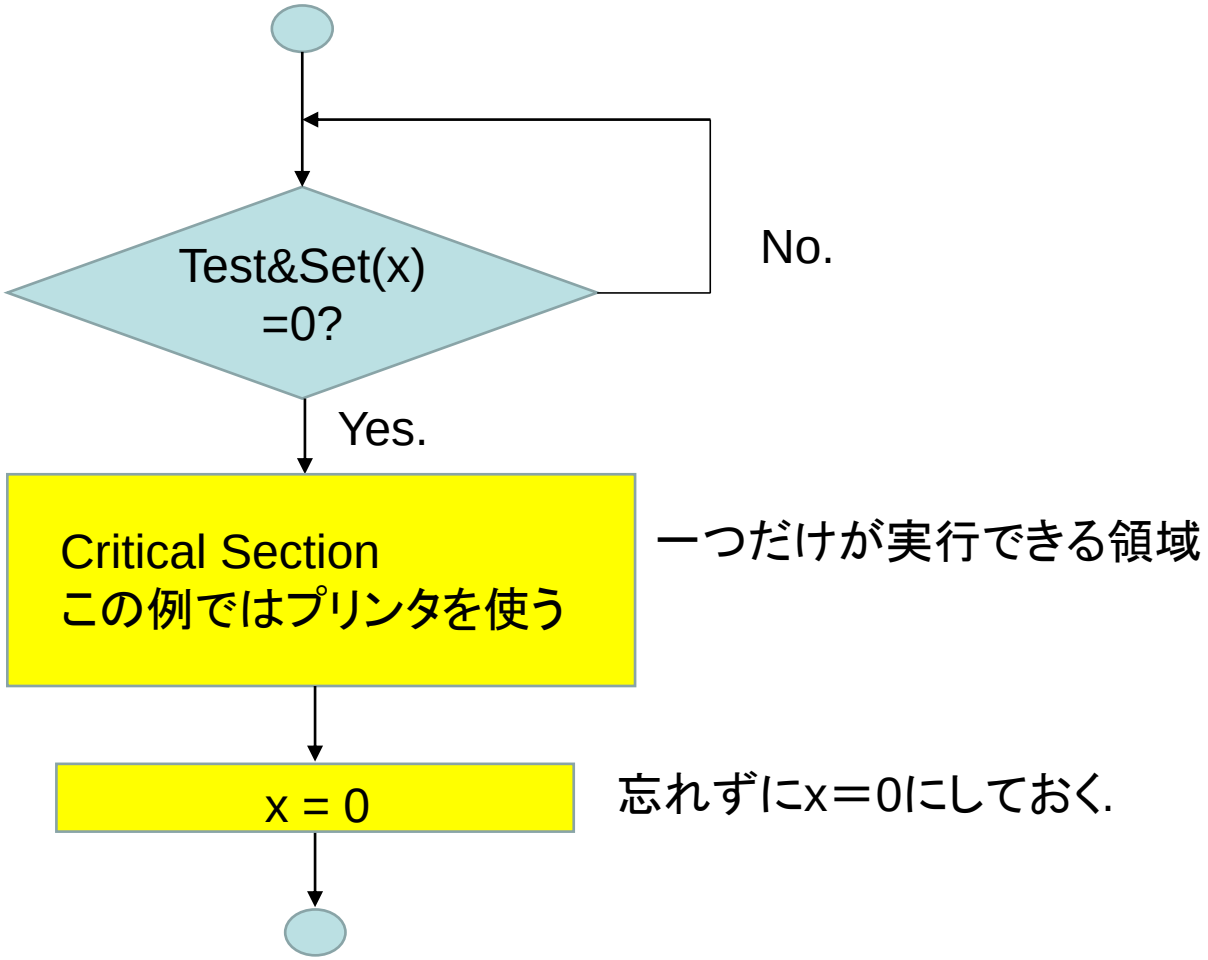

xを読み出す。
1を書きこむ
2つの操作を不可分に行う

この間共有メモリを占有

同時に命令が実行されても、必ず一つだけ0を読み、他は1にする。
→ハードウェアの支援が必要

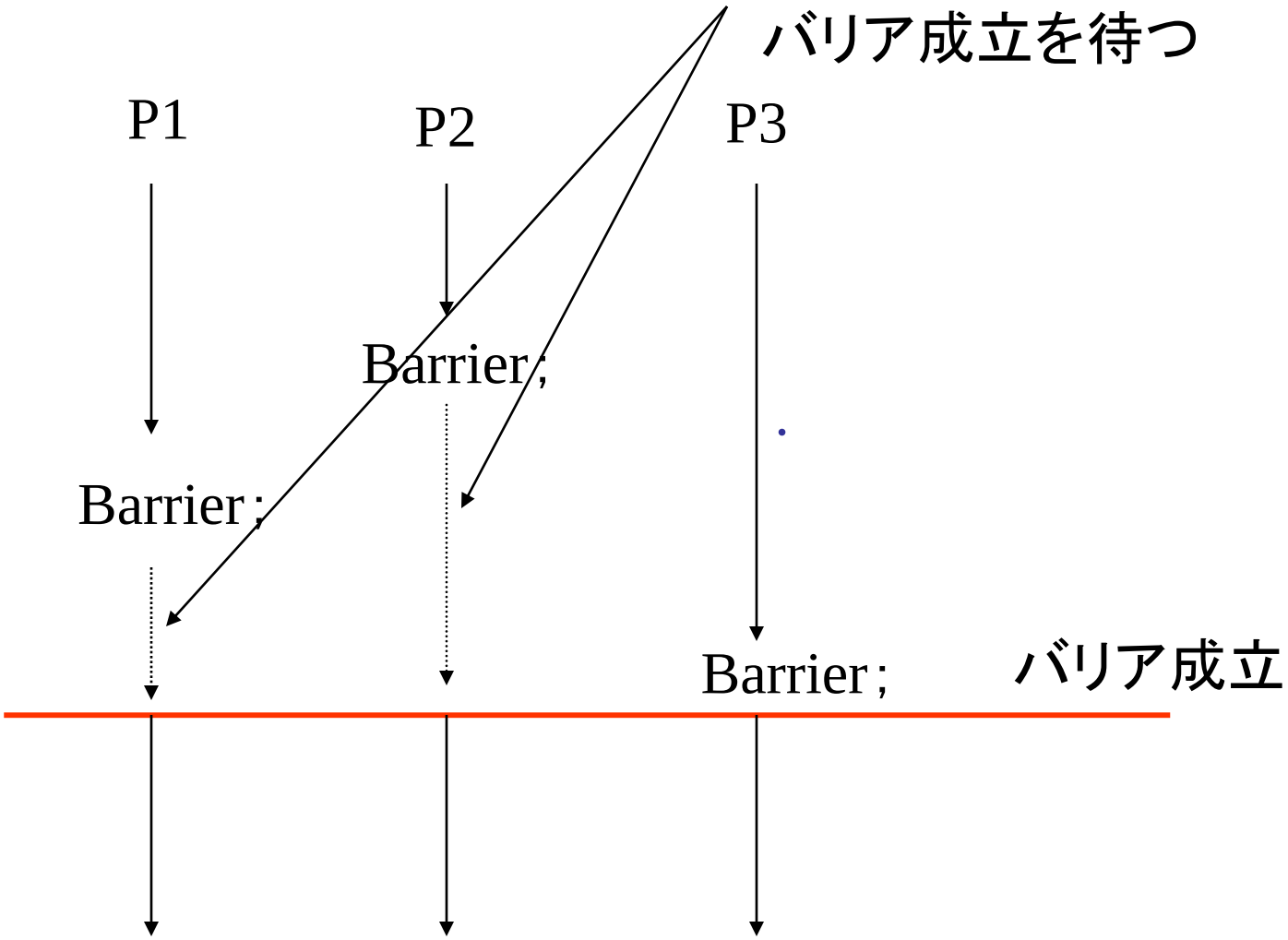
他にもSwap, Compare&Swap, Fetch&Dec, Fetch&Addなど色々あるが、原理は同じ

# Critical Sectionの実行



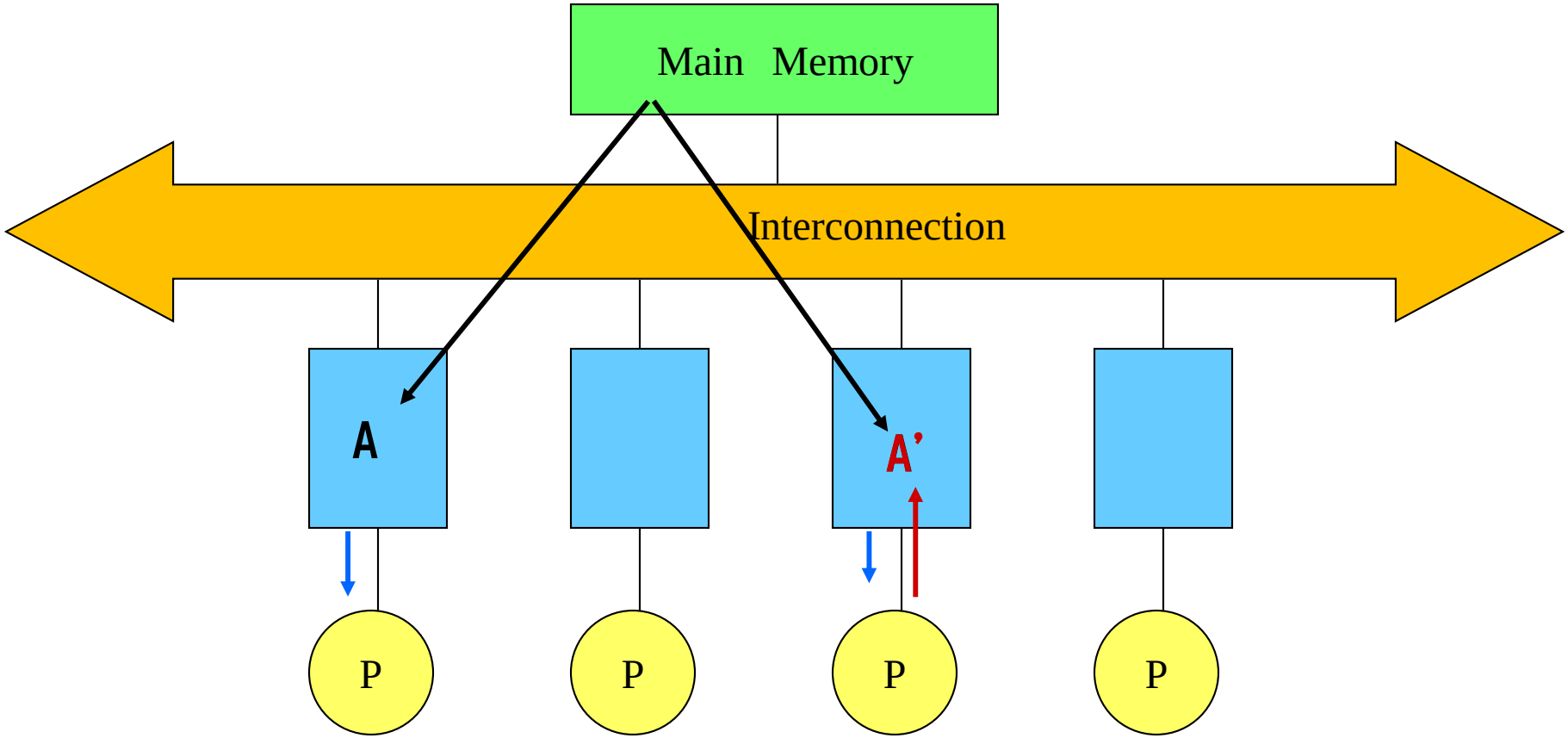

不可分命令があればCritical Sectionが作れる→なんでもできる！
でもちょっと使いにくい

# バリア同期




バリア同期は不可分命令があれば作れるが、専用のハードウェアを使う場合もある

# 3. キャッシュの一貫性問題

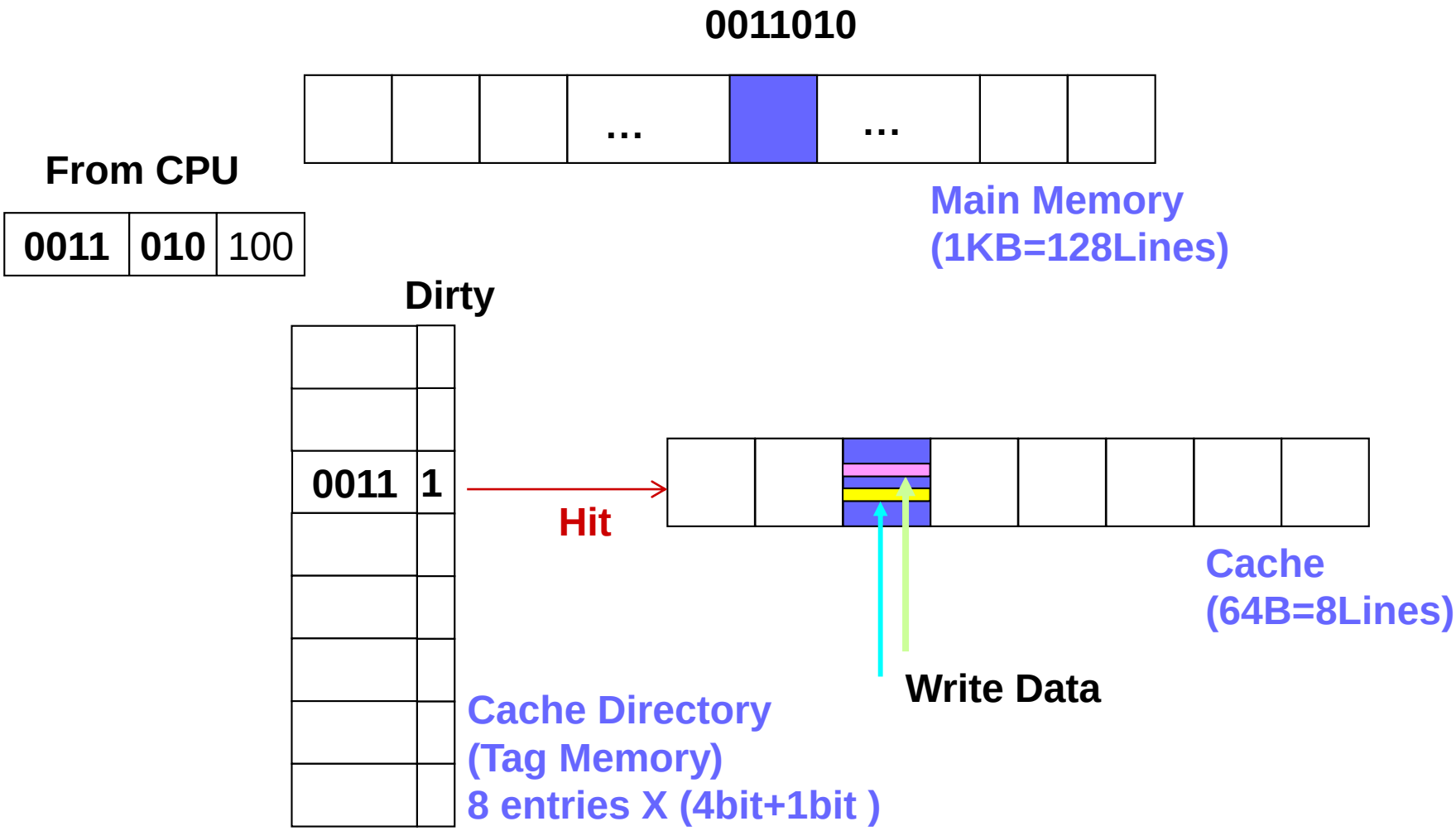



キャッシュを分散すれば、当然それぞれのキャッシュで
データの不一致が生じる

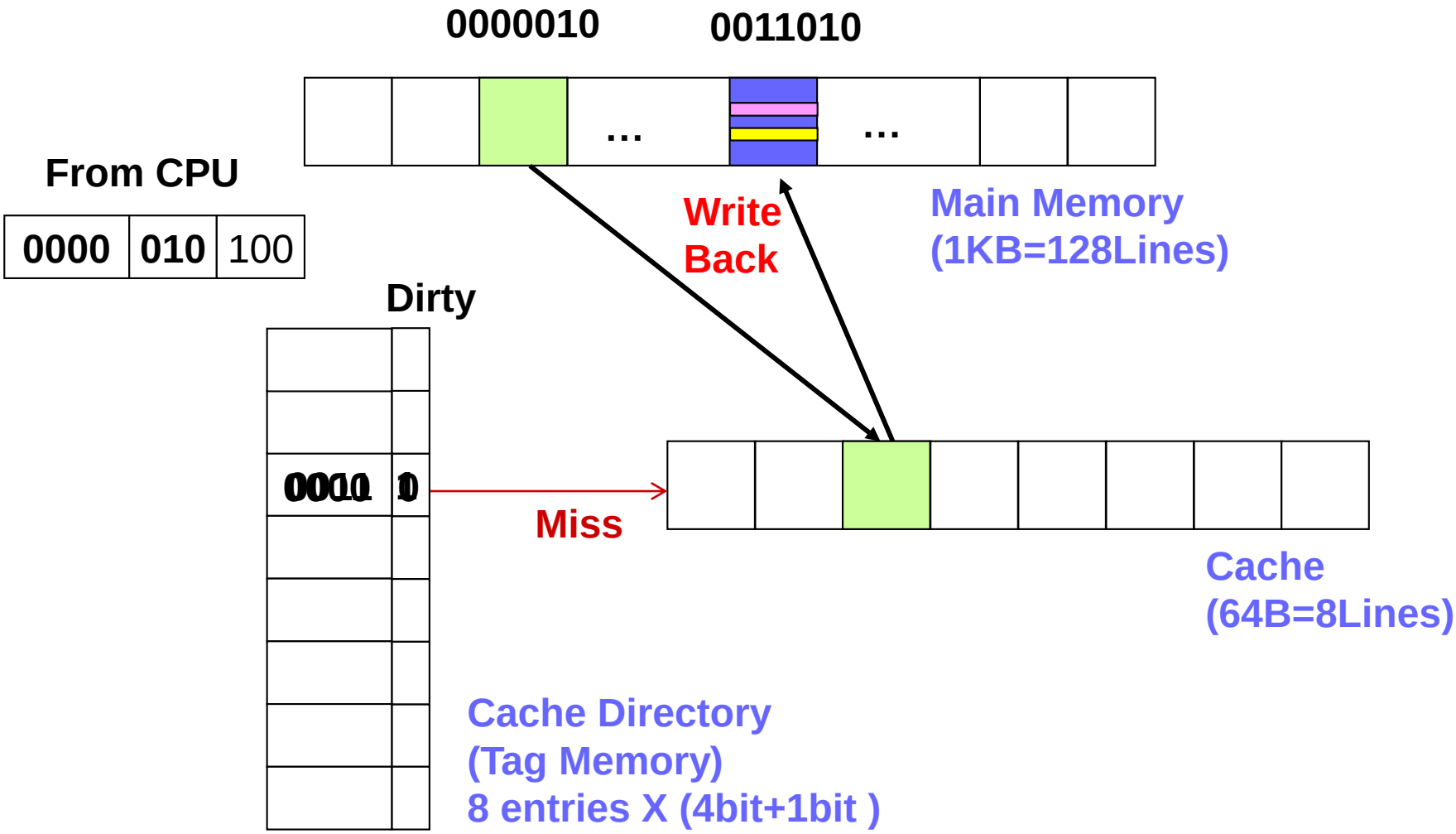
# キャッシュ一貫性問題の解決

- キャッシュを分散する限り不一致は起きる
- いつでも一致させる
  - コストは高いが共有メモリとして完全なモデルが実現できる→ Sequential Consistency
  - 共有バスなどの「皆が見れる通信路」があればSnoop Cache
  - 分散メモリ型ではDirectory方式が使われる
- 同期の時だけ一致を取る：緩いモデルも使われる。

# 復習：Write Back （Hit）



# 復習：Write Back （Replace）

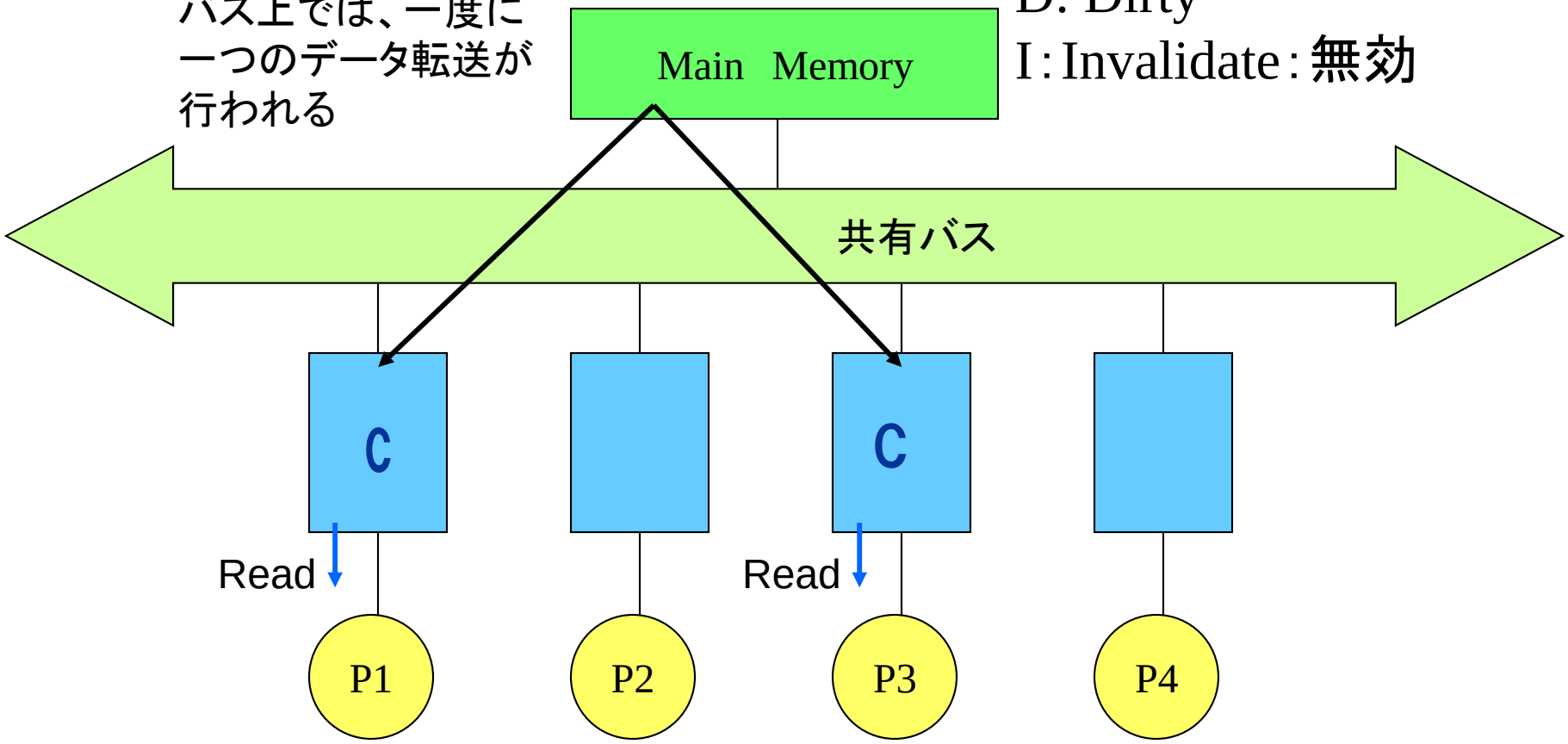


# 基本プロトコル

キャッシュの各ブロックの状態

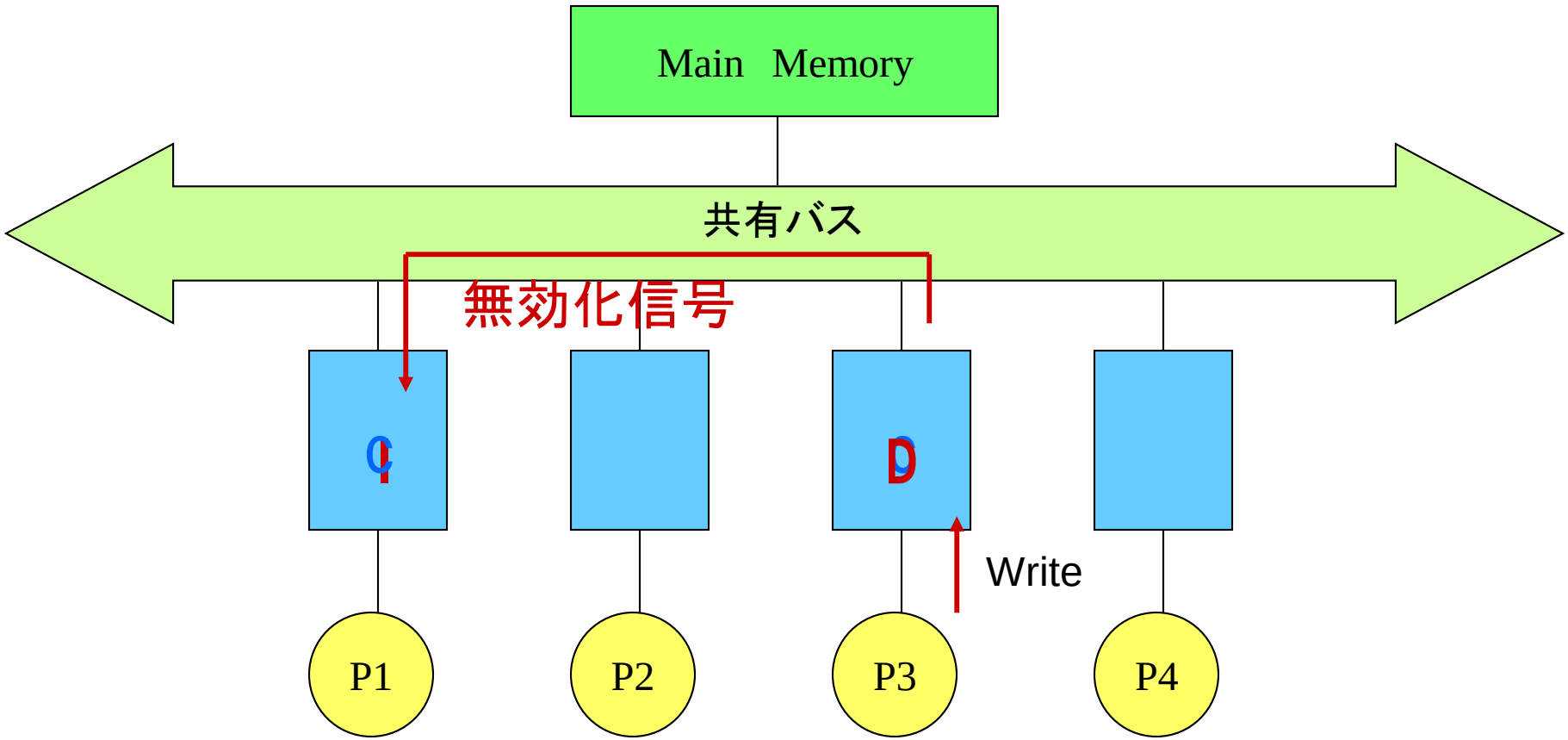
C : Clean (主記憶と一致)
D: Dirty
I : Invalidate : 無効

バス上では、一度に一つのデータ転送が行われる




同じキャッシュブロックを読み出すと、両方共Cleanになる。

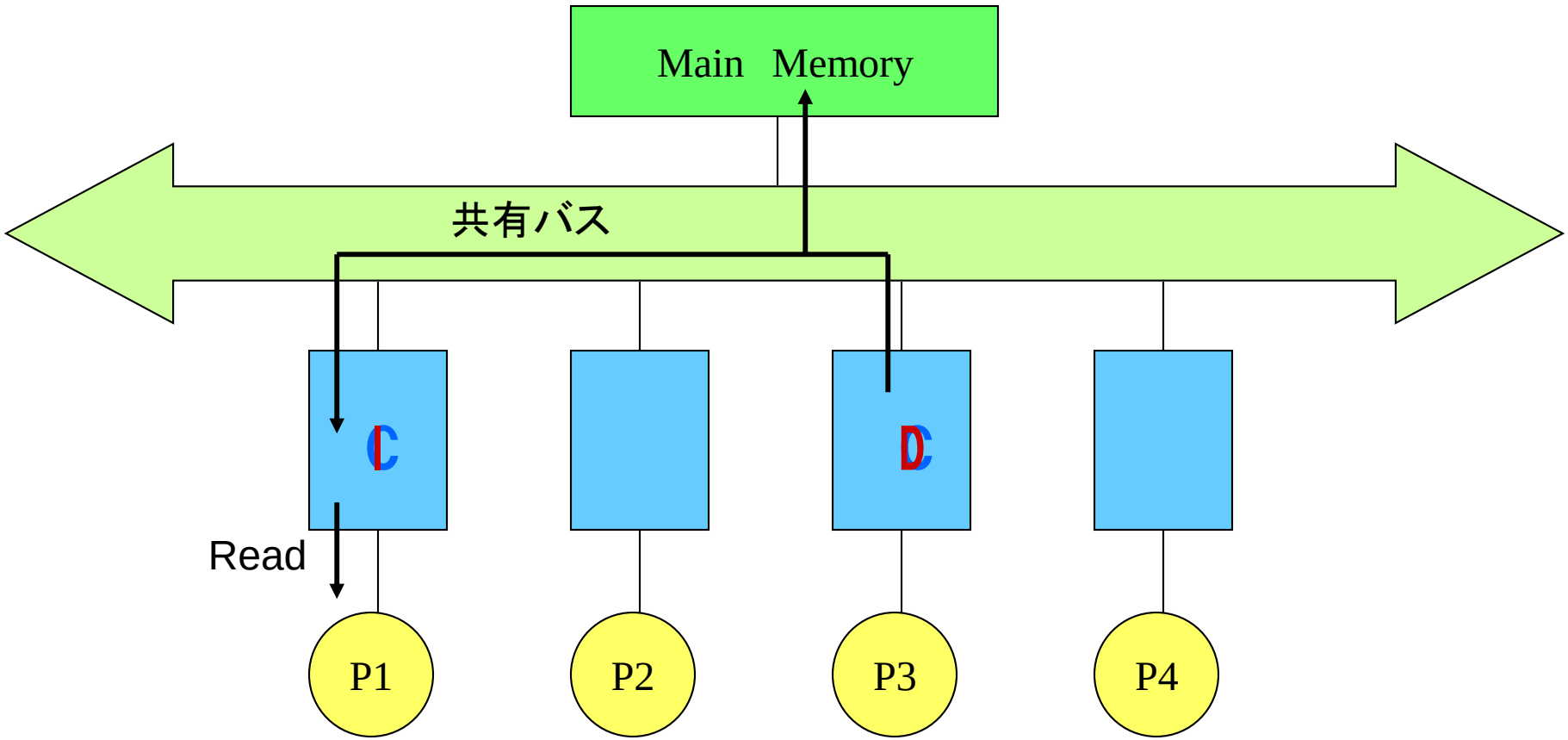
# P3が書き込み無効化




全てのキャッシュがバスを
見ており(スヌープ)、アドレスが
一致すると無効化

書き込みを行う
Clean→Dirtyに変化
共有バス上に書いたアドレスを送り
コピーを無効化

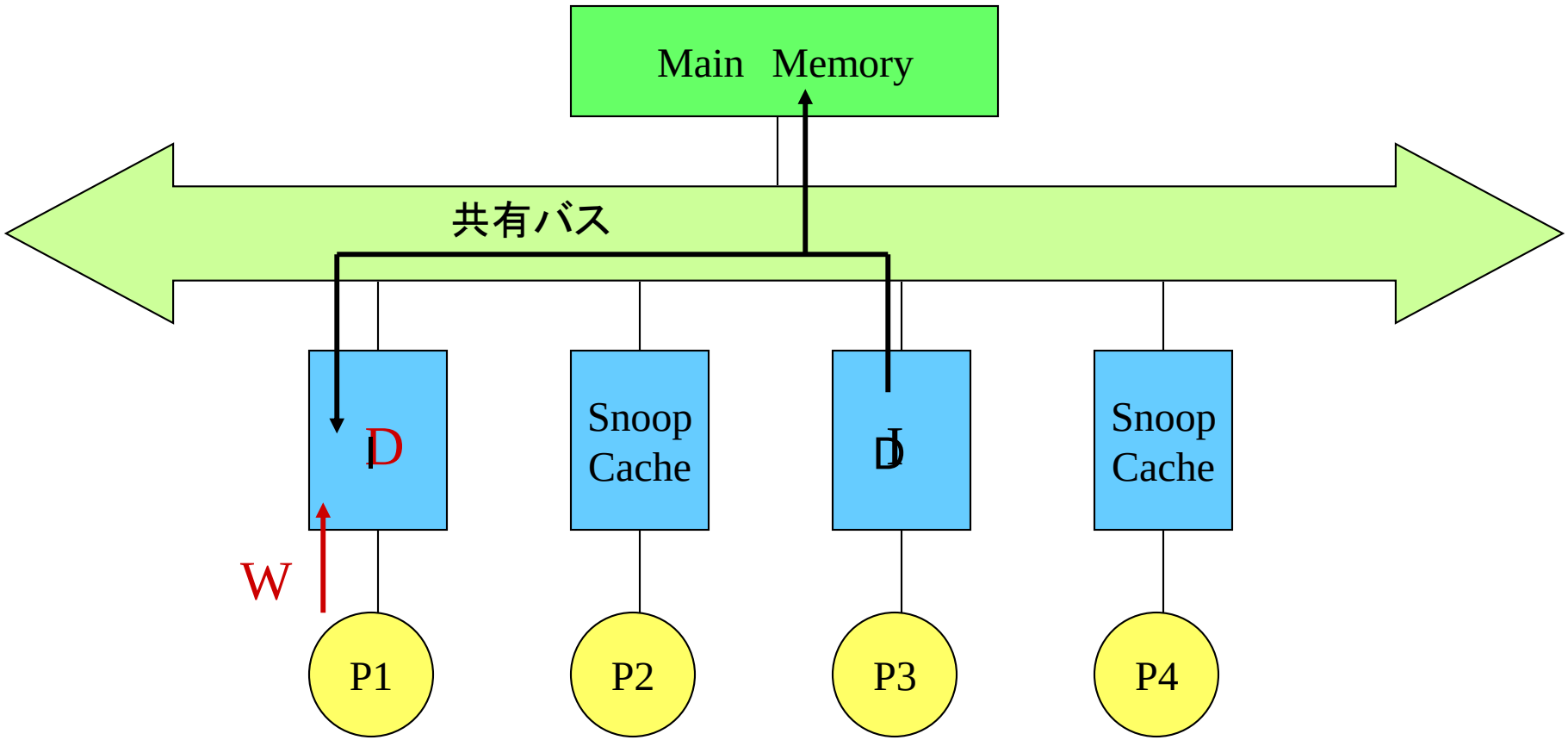
# P1が読み出しをした場合




P1が読み出すとミスが起き、
主記憶に共有バスを通して取りに行く
Cleanになる

共有バス上のアドレスを見て、アドレスが
一致してDのブロックへの読み出し要求を
検出→共有メモリに書き戻してからデータを
要求元に転送→Cleanになる

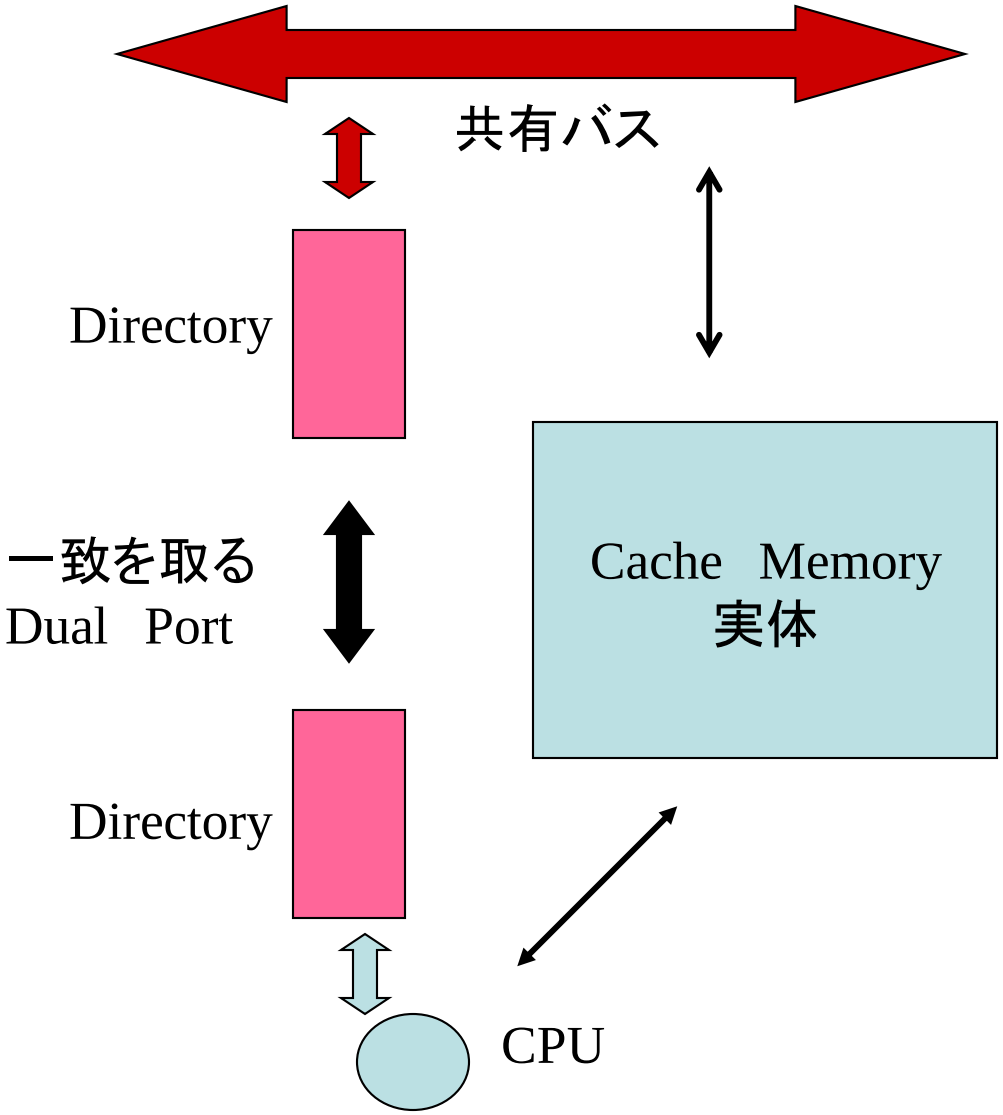
# P1が書き込みをした場合




P1が読み出すとミスが起き、
主記憶に共有バスを通して取りに行く
書き込みを行ってDirtyになる

共有バス上のアドレスを見て、アドレスが
一致してDのブロックへの書き込み要求を
検出→共有メモリに書き戻してからデータを
要求元に転送→I（無効）になる

# スヌープキャッシュの構造



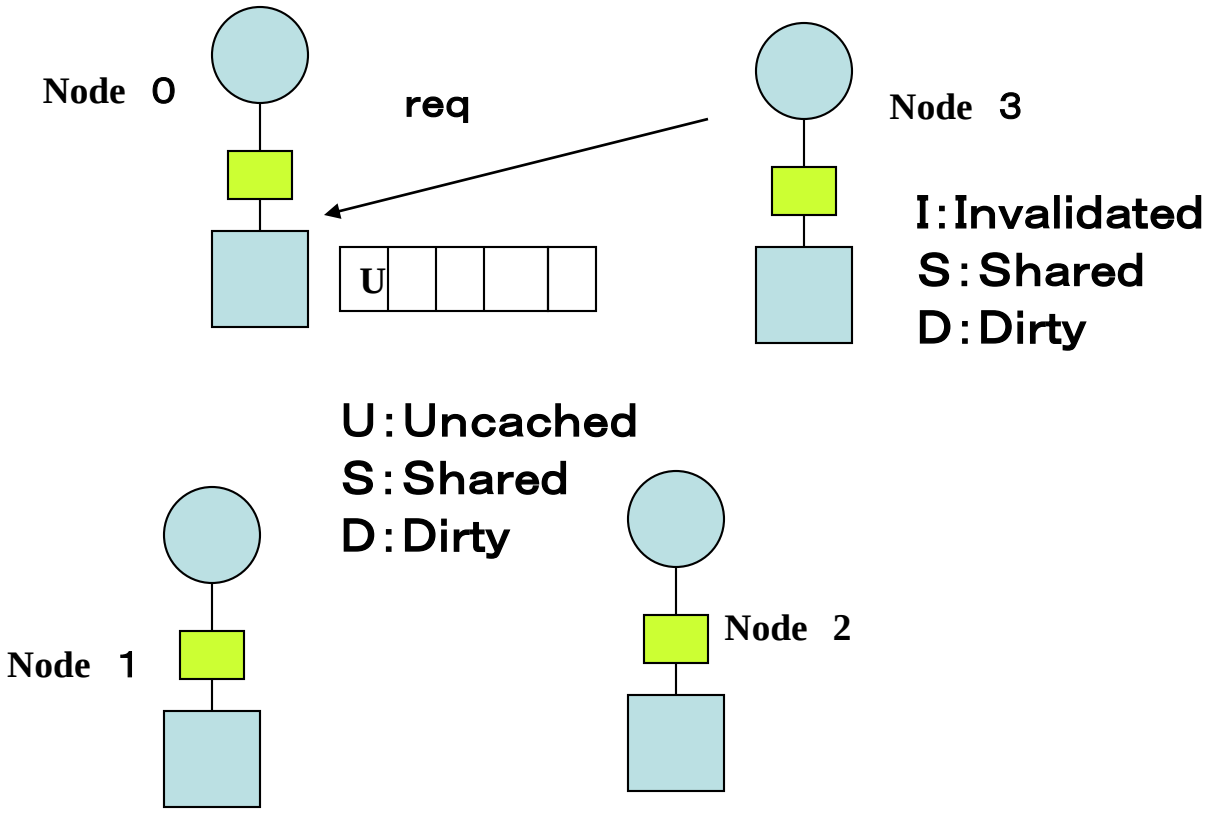

Directoryは、
両側からアクセス
可能

CPUとは関係なく
バスのTransaction
をチェックすること
ができる

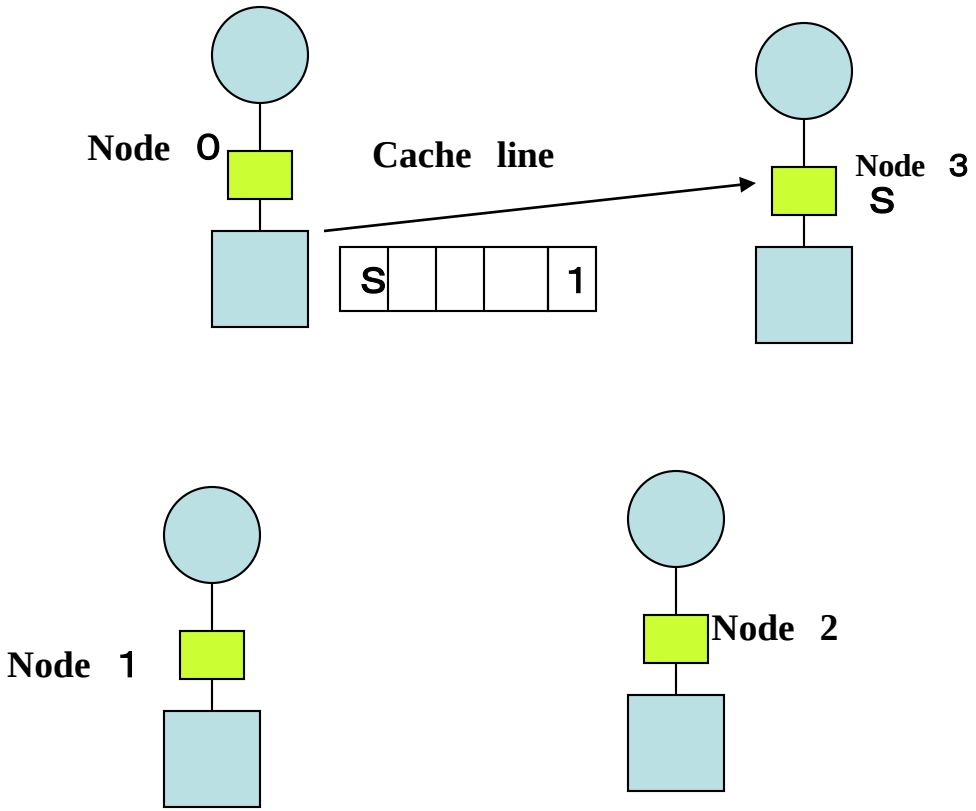
# ディレクトリ方式のキャッシュ

- ホームメモリが共有情報をディレクトリで管理
- ノード間のメッセージ交換でキャッシュの一致を制御
- プロトコル自体はスヌープ方式と似ている
- バスがないので常にディレクトリをメッセージでアクセスする

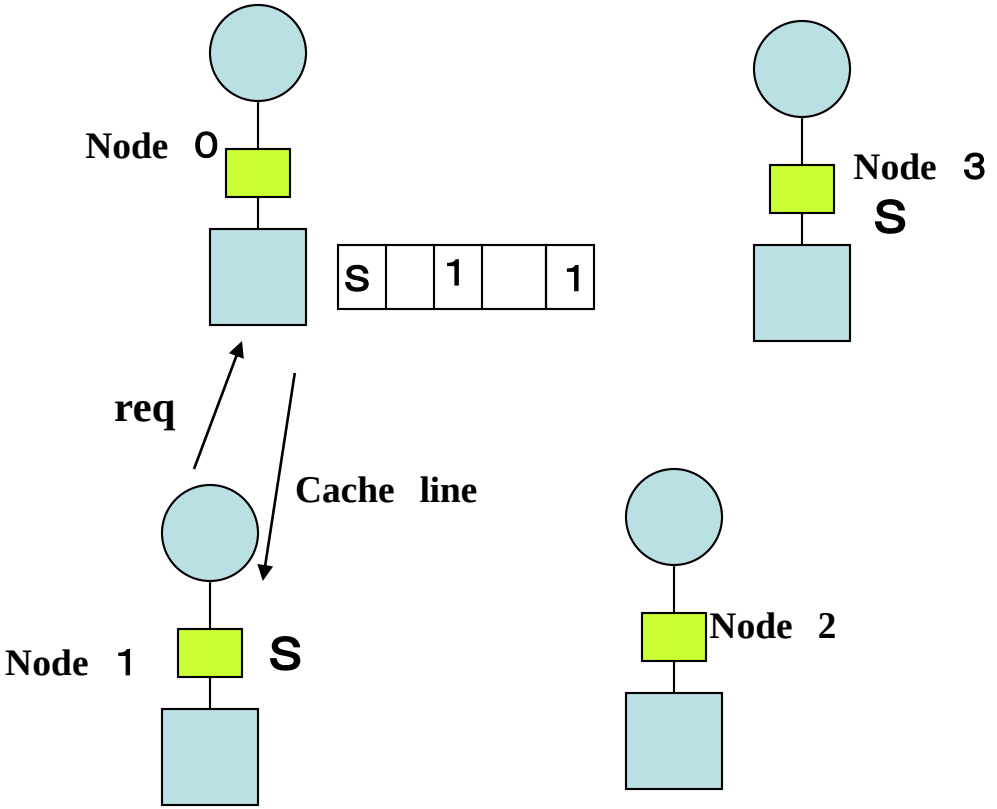
# キャッシュの制御（Node 3読み出し）



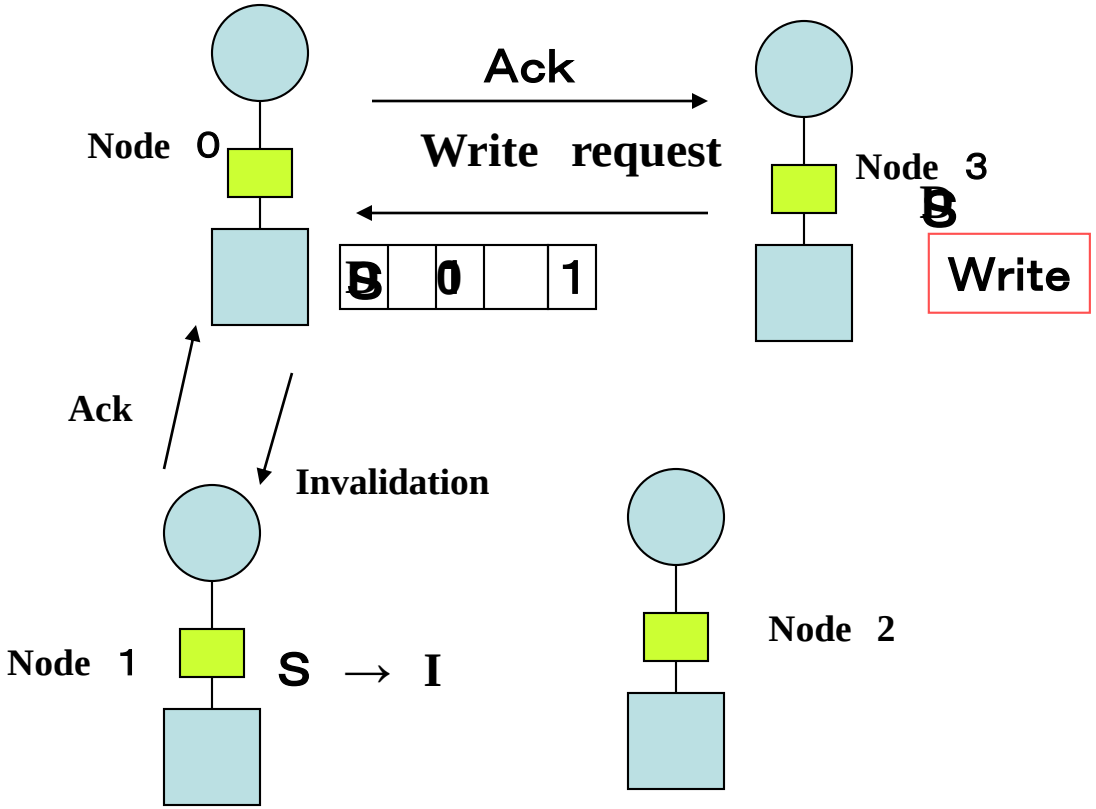
# キャッシュの制御（Node 3読み出し）



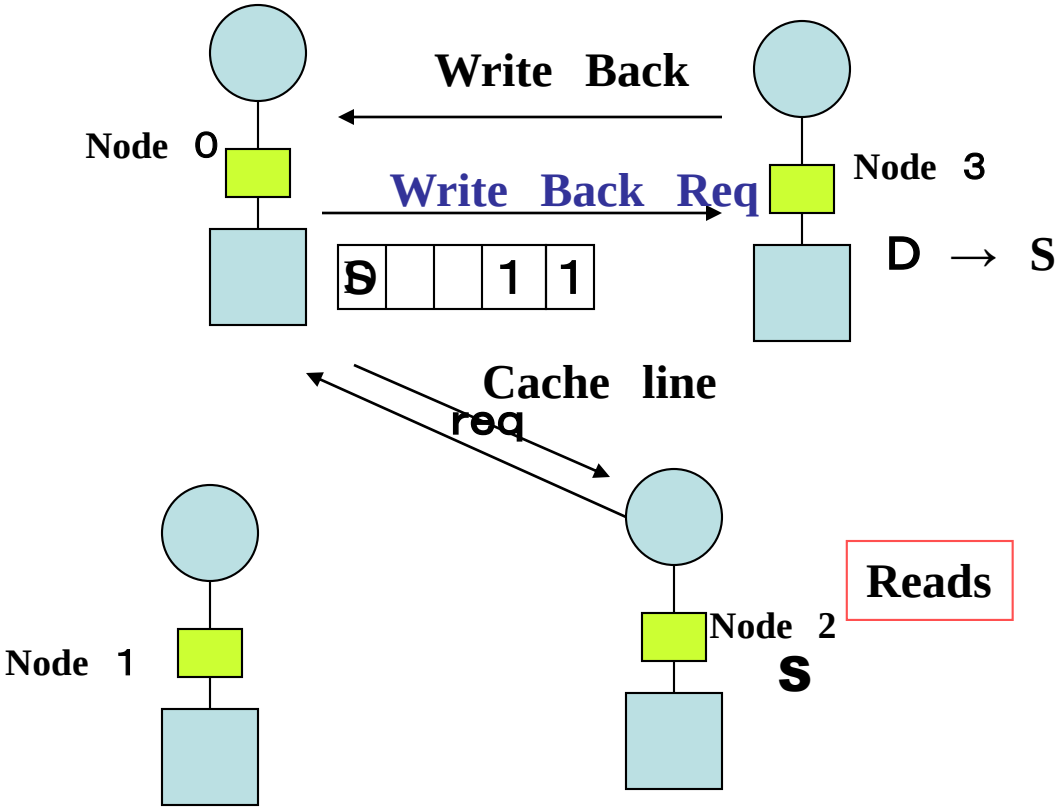
# キャッシュの制御（Node 1読み出し）



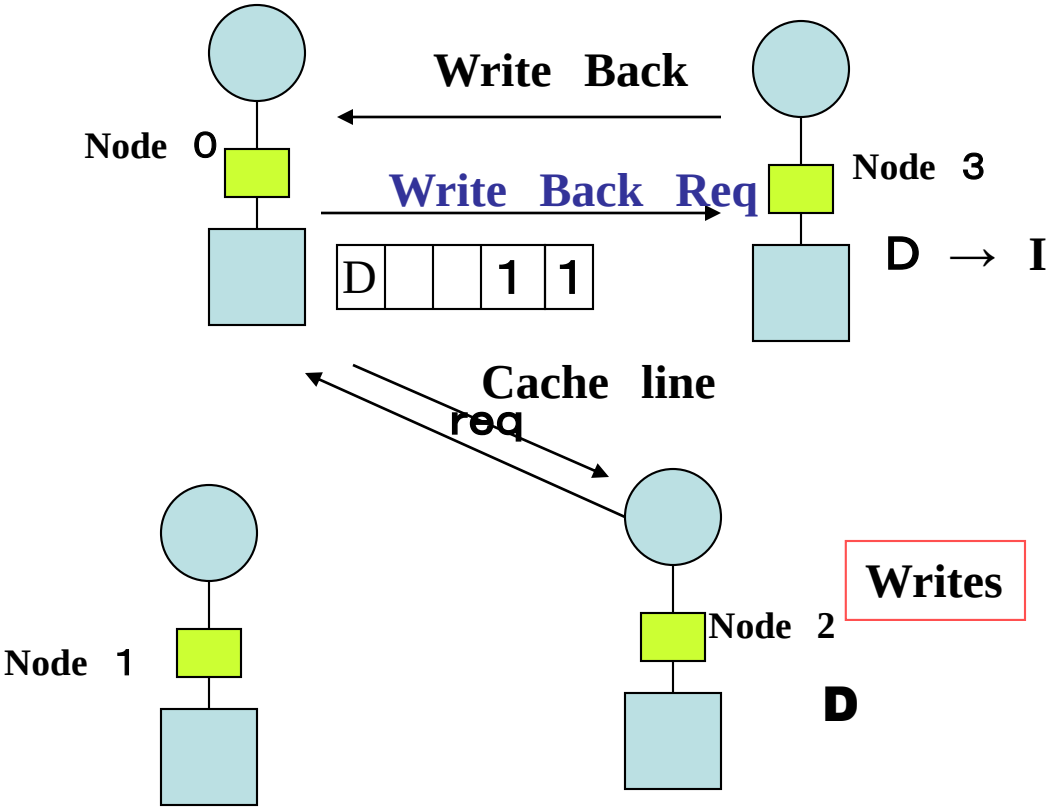
# キャッシュの制御（Node 3書込み）



# キャッシュの制御（Node 2読み出し）



# キャッシュの制御（Node 2書込み）



# スヌープキャッシュとディレクトリキャッシュ

- スヌープキャッシュは共有バスのように皆が見る(スヌープ)ことのできる通信路が必要
  - 転送、キャッシュブロックの状態制御は分散的に行われる
  - 集中メモリシステムに向いている
- ディレクトリキャッシュは、ホームメモリのディレクトリが共有バスの代わりに管理センターの役割を果たす
  - メッセージを交換してブロックの状態制御を行う
  - 汎用性が高いが、メッセージ交換のコストは大きい

# まとめ

- スマフォ、ノートPCの全て、サーバー、スーパーコンピュータの多くが共有メモリ型計算機
- ポイント
  - 高速化のためには並列化が必要
    - プログラマによる並列化→来週OpenMPの演習を行う
    - コンパイラによる自動並列化
  - 共有メモリを使ったデータ交換には同期が必要
    - 不可分命令
    - バリア同期
  - 分散したキャッシュの一貫性制御
    - スヌープキャッシュ
    - ディレクトリキャッシュ

# 演習

- P1,P2,P3が同じキャッシュブロックについて以下の操作を行った際の各キャッシュの状態を求めよ。無効化信号がバス上に流れるのはどのタイミングか？初期状態は全てIとする。

1. P1が読み出し
2. P2が読み出し
3. P1が書き込み
4. P3が読み出し
5. P2が書き込み
6. P1が読み出し