# Optimizaciones para paginamiento en demanda

- Grabar en disco solo páginas que se hayan modificado
  - Al recuperar una página *q* de disco, colocar *bit dirty* en 0
  - La MMU coloca automáticamente bit *dirty* en 1 cuando se escribe en *q*
  - Al reemplazar *q*, grabar solo si bit dirty ≡ 1
- Si el disco de paginamiento está desocupado, elegir cualquier página *q* con bitWS(q) ≡ 0 y bitR(q) ≡ 0:
  - grabar *q* en disco
  - colocar bit dirty en 0
- Tratar de que el 20 % de las páginas reales disponibles para procesos no pertenezca al working set de ningún proceso
  - Si es menor que 20 % recurrir a swapping
  - Si es mayor que 20 % recuperar procesos de disco
- Tratar de que la tasa de page faults de cada proceso sea por ejemplo 10 page faults por segundo:
  - Un disco moderno permite atender unos 100 page faults por segundo, cuando no hay que escribir la página de reemplazo
  - 10 page faults significarían aproximadamente un 10 % de sobrecosto en tiempo de ejecución
  - Si es mayor que 10 page faults : agrandar Δt para ese proceso
  - Si es menor que 10 page fauls: disminuir Δt para ese proceso

# Localidad de los accesos

- Localidad de los accesos a variables locales (en la pila): excelente
- Localidad de los accesos al código: buena
- Localidad de los acceso a datos: depende de los algoritmos y las estructuras de datos
  - acceso secuencial a arreglos de gran tamaño: mala
  - quicksort: mala localidad al comienzo cuando hay que particionar el arreglo completo, pero excelente localidad a medida que se ordenan arreglos cada vez más pequeños
  - Listas enlazadas, árboles binarios y objetos en general: mala localidad especialmente cuando el heap se encuentra muy fragmentado; basta referenciar un solo objeto de una página para que toda esa página deba residir en memoria
- Además hay que considerar los desaciertos en la TLB
  - considere por ejemplo sumar los elementos de una matriz de 512 x 512 completamente residente en memoria almacenada por filas en la memoria del computador, ¿cuál recorrido de la matriz es más eficiente? ¿Por filas o por columnas?

```
double m[512][512];
…
double s= 0;
for (int i= 0; i<512; i++)
   for (int j= 0; j<512; j++)
     s += m[i][j];
```

```
double m[512][512];
…
double s= 0;
for (int j= 0; j<512; j++)
   for (int i= 0; i<512; i++)
     s += m[i][j];
```

# Implementación de la estrategia del reloj para un núcleo clásico monocore

```
// Se invoca cuando ocurre un pagefault,
// es decir bit V==0 o el acceso fue una escritura y bit W==0

void pagefault(int page) {
  Process *p= current_process; // propietario de la página
  int *ptab= p->pageTable;
  if (bitS(ptab[page]))        // ¿Está la página en disco?
    pageIn(p, page, findRealPage()); // sí, leerla de disco
  else
    segfault(page);            // no
}

// Recupera de disco la página page
// del proceso p colocándola en realPage

void pageIn(Process *p, int page, int realPage) {
  int *ptab= p->pageTable;
  setRealPage(&ptab[page], realPage);
  setBitV(&ptab[page], 1);
  loadPage(p, page);   // retoma otro proceso
  setBitS(&ptab[page], 0);
  purgeTlb();          // invalida la TLB
  purgeL1();           // invalida cache L1
}
```

*// Graba en disco la página `page` del proceso `q`*

```
int pageOut(Process *q, int page) {
  int *qtab= q->pageTable;
  int realPage= getRealPage(qtab[page]);
  savePage(q, page);   // retoma otro proceso
  setBitV(&qtab[page], 0);
  setBitS(&qtab[page], 1);
  return realPage; // Retorna la página real en donde se ubicaba
}
```

*// Variables globales*

```
Iterator *it; // = processIterator();
Process *cursor_process= NULL;
int cursor_page;
```

```
int findRealPage() {
  // recorre las páginas residentes en memoria de todos los procesos buscando una
  // página que no haya sido referenciada en toda una vuelta del reloj
  int realPage= getAvailableRealPage();
  if (realPage>=0)        // ¿Quedan páginas reales disponibles?
    return realPage;      // Sí, retornamos esa página
  // no, hay que hacer un reemplazo
  for (;;) {
    if (cursor_process==NULL) { // ¿Quedan páginas en proceso actual?
      // no
      if (!hasNext(it))     // ¿Quedan procesos por recorrer?
        resetIterator(it); // partiremos con el primer proceso nuevamente
      cursor_process= nextProcess(it);   // pasamos al próximo
      cursor_page= cursor_process->firstPage; // primera pág.
    }
    // Estamos visitando la página cursor_page del proceso cursor_process
    int *qtab= cursor_process->pageTable;
    // mientras queden páginas por revisar en cursor_process
    while (cursor_page<=cursor_process->lastPage) {
      if (bitV(qtab[cursor_page])) {    // ¿Es válida?
        if (bitR(qtab[cursor_page]))    // no fue referenciada
          setBitR(&qtab[cursor_page], 0);
        else // sí, se reemplaza la página cursor_page de cursor_process
          return pageOut(cursor_process, cursor_page++);
      }
      cursor_page++;
    }
    // Se acabaron las páginas de cursor_process,
    // hay que buscar en el próximo proceso
    cursor_process= NULL;
  }
}
```

# Ejercicio: Reimplemente la misma estrategia del reloj pero considerando una MMU que no implementa el bit R

```
// Se invoca cuando ocurre un pagefault,
// es decir bit V==0 o el acceso fue una escritura y bit W==0
void pagefault(int page) {
  Process *p= current_process; // propietario de la página
  int *ptab= p->pageTable;
  if (bitV2(ptab[page])) {
    setBitV(&ptab[page], 1);
    purgeTlb();
  }
  else if (bitS(ptab[page])     // ¿Está la página en disco?
    pageIn(p, page, findRealPage()); // sí, leerla de disco
  else
    segfault(page);              // no
}

// Recupera de disco la página page
// del proceso p colocándola en realPage
void pageIn(Process *p, int page, int realPage){
  int *ptab= p->pageTable;
  setRealPage(&ptab[page], realPage);
  setBitV(&ptab[page], 1);
  setBitV2(&ptab[page], 1);
  loadPage(p, page);   // retoma otro proceso
  setBitS(&ptab[page], 0);
  purgeTlb();          // invalida la TLB
  purgeL1();           // invalida cache L1
}
```

*// Graba en disco la página `page` del proceso `q`*

```
int pageOut(Process *q, int page) {
  int *qtab= q->pageTable;
  int realPage= getRealPage(qtab[page]);
  savePage(q, page);   // retoma otro proceso
  setBitV(&qtab[page], 0);
  setBitV2(&qtab[page], 0);
  setBitS(&qtab[page], 1);
  return realPage; // Retorna la página real en donde se ubicaba
}
```

*// Variables globales*

```
Iterator *it; // = processIterator();
Process *cursor_process= NULL;
int cursor_page;
```

```
int findRealPage() {
  // recorre las páginas residentes en memoria de todos los procesos buscando una
  // página que no haya sido referenciada en toda una vuelta del reloj
  int realPage= getAvailableRealPage();
  if (realPage>=0)        // ¿Quedan páginas reales disponibles?
    return realPage;      // Sí, retornamos esa página
  // no, hay que hacer un reemplazo
  for (;;) {
    if (cursor_process==NULL) { // ¿Quedan páginas en proceso actual?
      // no
      if (!hasNext(it))     // ¿Quedan procesos por recorrer?
        resetIterator(it); // partiremos con el primer proceso nuevamente
      cursor_process= nextProcess(it);   // pasamos al próximo
      cursor_page= cursor_process->firstPage; // primera pág.
    }
    // Estamos visitando la página cursor_page del proceso cursor_process
    int *qtab= cursor_process->pageTable;
    // mientras queden páginas por revisar en cursor_process
    while (cursor_page<=cursor_process->lastPage) {
      if (bitV2(qtab[cursor_page]))    // ¿Es válida?
        if (bitV(qtab[cursor_page]))    // no fue referenciada
          setBitV(&qtab[cursor_page], 0);
        else // sí, se reemplaza la página cursor_page de cursor_process
          return pageOut(cursor_process, cursor_page++);
      }
      cursor_page++;
    }
    // Se acabaron las páginas de cursor_process,
    // hay que buscar en el próximo proceso
    cursor_process= NULL;
  }
}
```

# Problema: tamaño de la tabla de páginas

- En x86 un proceso puede direccionar hasta 4 GB: 2^20 páginas
- ¡La tabla de páginas debería ocupar 4 MB!
- Incluso para procesos que ocupan poca memoria
- Con direcciones de 64 bits la tabla debería ser aún más gigantesca
- Alternativamente se pueden usar tablas de páginas más pequeñas pero reduciendo el tamaño máximo de memoria direccionable por los procesos
- Solución: *tablas de página de múltiples niveles*

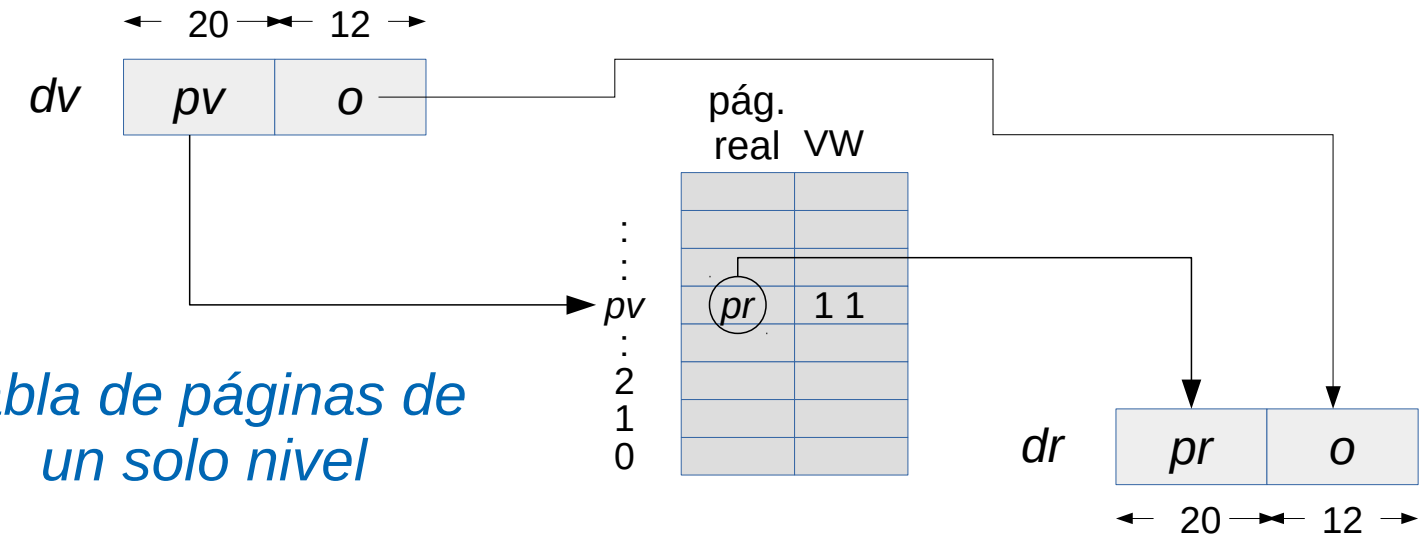
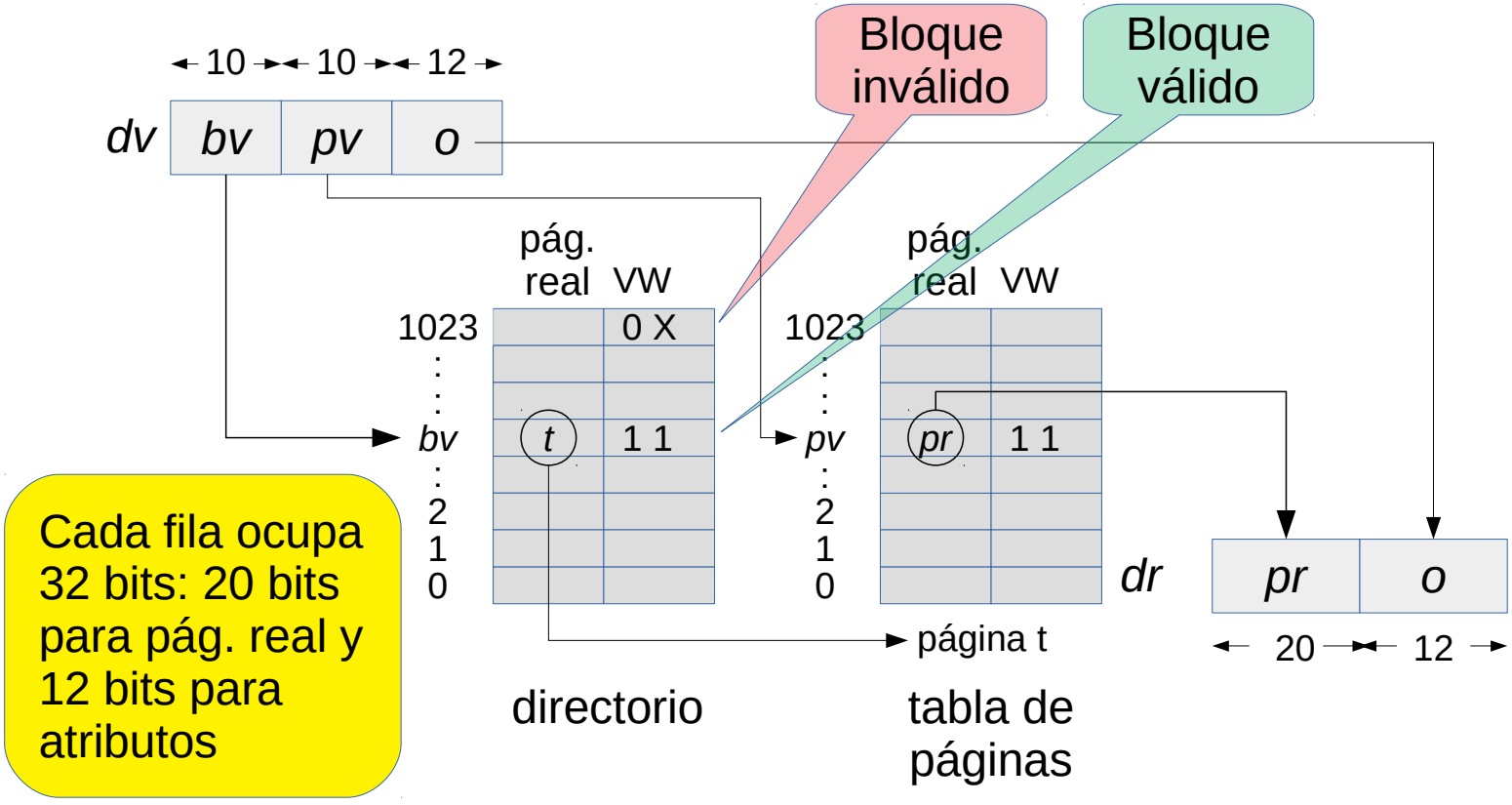



*Tabla de páginas de un solo nivel*

# Tablas de páginas de 2 niveles en x86

- El espacio de direcciones virtuales de un proceso se descompone en 1024 bloques, cada uno de 4 MB (1024 páginas de 4 KB)
- Para traducir las direcciones de un solo bloque se necesita una tabla de páginas que ocupa exactamente una página de la memoria (4 KB)
- Además se agrega una nueva tabla: el directorio, que ocupa una página (4 KB) y que contiene los números de página de las tablas de páginas para cada bloque (hasta 1024 tablas)
- Un proceso pequeño que entre código y datos requiere menos que 4 MB necesita una tabla de páginas para código y datos, otra tabla para la pila y finalmente el directorio: sobrecosto total es de 3 pág.



# Tablas de páginas de *n* niveles en x86-64

- Las direcciones virtuales y reales son de 64 bits y por lo tanto las tablas solo pueden almacenar 512 filas
- Cada proceso tiene *n* niveles de tablas de páginas
- El primer nivel es el directorio y tiene 512 filas de 64 bits con páginas reales que almacenan tablas de segundo nivel
- En el nivel *k* cada tabla tiene 512 filas de 64 bits con páginas reales que almacenan tablas de nivel *k-1*, con $1 < k < n$
- En el nivel *n* cada tabla tiene 512 filas de 64 bits con las páginas reales atribuidas a un bloque de 2 MB del proceso
- Amd64 define 4 niveles con direcciones de 48 bits (los restantes son 0)
- Puede direccionar hasta 4 KB·$512^4$ = 256 TB