Cours 4: Gestion de la mémoire

Christophe Morvan

Université Paris-est, Marne-la-Vallée

20 octobre 2015

Gestion mémoire ?

Observation

Nous souhaitons atteindre plusieurs objectifs :

- Faire fonctionner plusieurs processus simultanément
- Ne pas être limité par la mémoire physique de la machine
- Avoir d'excellentes performances

Gestion mémoire ?

Observation

Nous souhaitons atteindre plusieurs objectifs :

- Faire fonctionner plusieurs processus simultanément
- Ne pas être limité par la mémoire physique de la machine
- Avoir d'excellentes performances

Problèmes

Comment faire :

- lorsque un processus souhaite plus de mémoire ?
- en cas d'erreur d'adresse dans un programme ?
- lorsqu'un processus n'utilise qu'une petite partie de la mémoire allouée ?
- lorsqu'un processus nécessite plus de mémoire que ce qui est installé ?

Problèmes du partage de la mémoire physique

Protection

- Un bug de A perturbe B
- A peut observer B

Problèmes du partage de la mémoire physique

Protection

- Un bug de A perturbe B
- A peut observer B

Équité

- Espace contigus
- Choix d'adresses

Problèmes du partage de la mémoire physique

Protection

- Un bug de A perturbe B
- A peut observer B

Équité

- Espace contigus
- Choix d'adresses

Épuisement

- Assez de mémoire ?
- Somme des processus $\geqslant$ mémoire physique

Problèmes du partage de la mémoire physique

Protection

- Un bug de A perturbe B
- A peut observer B

Équité

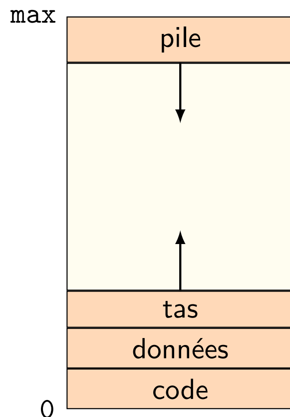
- Espace contigus
- Choix d'adresses

Épuisement

- Assez de mémoire ?
- Somme des processus $\geqslant$ mémoire physique

Une solution :

Mémoire virtuelle



Plan

1. Mémoire virtuelle (premiers pas)

2. Mémoire virtuelle (moderne)
   - Pagination
   - Algorithmes d'échange

3. Appels systèmes

Progression

1. Mémoire virtuelle (premiers pas)
2. Mémoire virtuelle (moderne)
   - Pagination
   - Algorithmes d'échange
3. Appels systèmes

Historique

OS & 1 programme

- Modèle simple
- Systèmes de protection pour l'OS
- Possibilité d'OS en ROM

Historique

OS & 1 programme

- Modèle simple
- Systèmes de protection pour l'OS
- Possibilité d'OS en ROM

IBM 360 : Bits de protection (1966)

- 4 bits servent de clé sur les adresses
  (stockés dans des registres dédiés du processeur)
- **Problème** : les adresses contenues dans le programme ne contiennent pas ces bits de protection

Historique

OS & 1 programme

- Modèle simple
- Systèmes de protection pour l'OS
- Possibilité d'OS en ROM

IBM 360 : Bits de protection (1966)

- 4 bits servent de clé sur les adresses
  (stockés dans des registres dédiés du processeur)
- **Problème** : les adresses contenues dans le programme ne contiennent pas ces bits de protection
- **Solution** : le chargeur effectue une traduction de toutes les adresses au chargement
  Attention aux constantes qui ne sont pas des adresses

Historique

OS & 1 programme

- Modèle simple
- Systèmes de protection pour l'OS
- Possibilité d'OS en ROM

IBM 360 : Bits de protection (1966)

- 4 bits servent de clé sur les adresses
  (stockés dans des registres dédiés du processeur)
- **Problème** : les adresses contenues dans le programme ne contiennent pas ces bits de protection
- **Solution** : le chargeur effectue une traduction de toutes les adresses au chargement
  Attention aux constantes qui ne sont pas des adresses

Existe encore (2015) sur des matériels "simples"

Espaces d'adressage

Une solution simple pour généraliser l'approche IBM 360 : les espaces d'adressages

Espaces d'adressage

Une solution simple pour généraliser l'approche IBM 360 : les espaces d'adressages

Registre base et limite

- Chaque processus possède une adresse de base et une adresse limite
- Chacune est stockée dans un registre dédié
- La traduction est simple : toute référence $\rightarrow$ addition du registre de base
- Attention : addition + comparaison à chaque référence

Espaces d'adressage

Une solution simple pour généraliser l'approche IBM 360 : les espaces d'adressages

Registre base et limite

- Chaque processus possède une adresse de base et une adresse limite
- Chacune est stockée dans un registre dédié
- La traduction est simple : toute référence $\rightarrow$ addition du registre de base
- Attention : addition + comparaison à chaque référence

Note : Le 8088 (ancêtre du x86) possédait plusieurs registres de base (pourquoi ?), mais aucun registre limite (problème ?)

Échange (va-et-vient)

Il est souvent souhaitable de disposer de plus de mémoire que ce qui est installé sur la machine

Échange (va-et-vient)

Il est souvent souhaitable de disposer de plus de mémoire que ce qui est installé sur la machine

Principe (va-et-vient)

- L'intégralité de la mémoire utilisée par un processus est placé sur disque lorsque celui-ci n'est plus actif et que la mémoire est requise
- Chaque processus possède un bloc contigu de mémoire
- De la mémoire est prévue pour l'éventuel accroissement
- Lors du stockage sur le disque la mémoire non-utilisée n'est pas enregistrée

Échange (va-et-vient)

Il est souvent souhaitable de disposer de plus de mémoire que ce qui est installé sur la machine

Principe (va-et-vient)

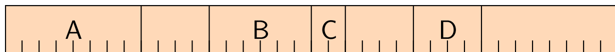
- L'intégralité de la mémoire utilisée par un processus est placé sur disque lorsque celui-ci n'est plus actif et que la mémoire est requise
- Chaque processus possède un bloc contigu de mémoire
- De la mémoire est prévue pour l'éventuel accroissement
- Lors du stockage sur le disque la mémoire non-utilisée n'est pas enregistrée

Système de compactage de la mémoire : chaque processus est dans une zone contiguë

Mémoire libre

Table de bits

- Unités d'allocation (mots → Ko)
- Un tableau bit → unité d'allocation (0/1)
- Pb : rechercher un espace libre de n Ko



1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1
			...				

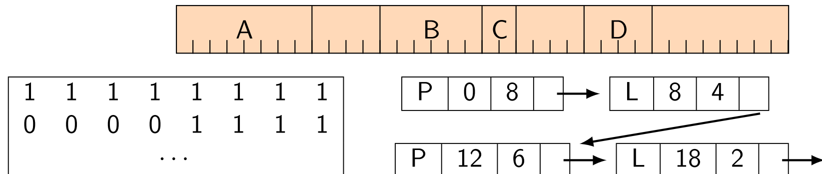
Mémoire libre

Table de bits

- Unités d'allocation (mots → Ko)
- Un tableau bit → unité d'allocation (0/1)
- Pb : rechercher un espace libre de n Ko

Liste chaînée

- Liste des occupations : processus (P)/libre (L)
- Chaque maillon : adresse début et taille
- Plusieurs choix d'algo pour l'allocation



Allocation mémoire

Lorsque la mémoire libre est stockée dans une liste il y a plusieurs choix d'algorithme pour allouer la mémoire

Allocation mémoire

Lorsque la mémoire libre est stockée dans une liste il y a plusieurs choix d'algorithme pour allouer la mémoire

Quelques algorithmes

- *first fit* : premier espace de taille suffisante
- *next fit* : idem mais *poursuit* sa recherche pour la recherche suivante
- *best fit* : recherche le meilleur ajustement
  Beaucoup plus lent – Optimise moins la mémoire
- *worst fit* : cherche l'espace libre le plus grand possible (pour éviter la fragmentation)

Allocation mémoire

Lorsque la mémoire libre est stockée dans une liste il y a plusieurs choix d'algorithme pour allouer la mémoire

Quelques algorithmes

- *first fit* : premier espace de taille suffisante
- *next fit* : idem mais *poursuit* sa recherche pour la recherche suivante
- *best fit* : recherche le meilleur ajustement
  Beaucoup plus lent – Optimise moins la mémoire
- *worst fit* : cherche l'espace libre le plus grand possible (pour éviter la fragmentation)

Il est possible d'améliorer ces algorithmes :

- Listes séparées (dans ce cas la liste *libre* peut être dans l'espace libre) + espaces trillés par taille
- *quick fit* plusieurs listes chacune a taille d'espace fixe

Progression

1 Mémoire virtuelle (premiers pas)

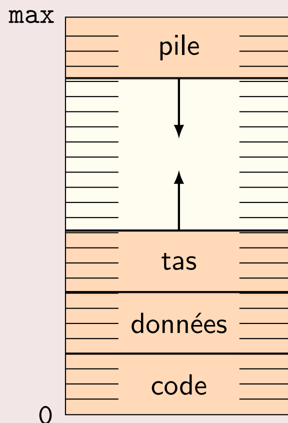
2 Mémoire virtuelle (moderne)
  Pagination
  Algorithmes d'échange

3 Appels systèmes

La pagination

Principe

- Chaque processus possède son espace d'adressage
- Cet espace d'adressage est découpé en pages (suite d'adresses contiguës qui seront placés de façon contiguë en mémoire)
- Le processus peut s'exécuter alors que certaines pages ne sont pas en mémoire
- L'emplacement d'une page en mémoire s'appelle cadre de page (*page frame*)



Translation : MMU

La MMU (Memory Management Unit)

Il s'agit de l'unité hardware qui effectue la correspondance entre les adresses virtuelles et les adresses physiques

Translation : MMU

La MMU (Memory Management Unit)

Il s'agit de l'unité hardware qui effectue la correspondance entre les adresses virtuelles et les adresses physiques

Exemple

Programme de 64Ko (adresses sur 16 bits) sur une mémoire de 32Ko, avec des pages de 4Ko

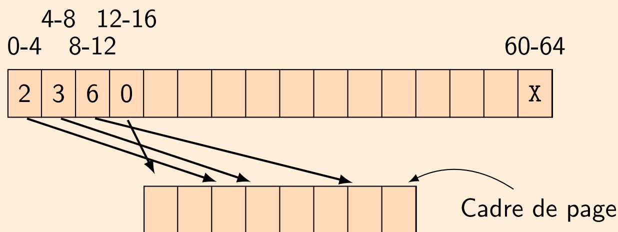


Table des pages

Pourquoi utiliser des tailles de pages de la forme 2^{n-k} ?

Table des pages

Pourquoi utiliser des tailles de pages de la forme 2^{n-k} ?

Codage de la table

$2^n = 2^k * 2^{n-k}$

k bits : numéro du cadre $\Rightarrow$ k premier bits de l'adresse virtuelle
$=$ emplacement dans la table des pages
La table des pages contient $k - \ell$ bits qui vont remplacer ceux de l'adresse virtuelle pour définir l'adresse physique

Table des pages

Pourquoi utiliser des tailles de pages de la forme 2^{n-k} ?

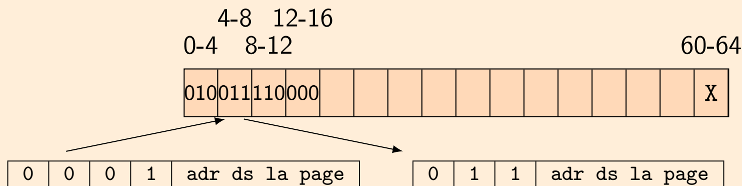
Codage de la table

$2^n = 2^k * 2^{n-k}$

k bits : numéro du cadre $\Rightarrow$ k premier bits de l'adresse virtuelle
$=$ emplacement dans la table des pages

La table des pages contient $k - \ell$ bits qui vont remplacer ceux de l'adresse virtuelle pour définir l'adresse physique

Suite de l'exemple



Une entrée

Détail d'une entrée

– Arch –	Inh.	Ref.	Mod.	Prot.	Pres.	– Numéro de cadre –

- Arch : Bits spécifiques à l'architecture
- Inh : Inhibe l'utilisation du cache
- Ref : À 1 dès que la page est consultée
- Mod : À 1 dès qu'elle est modifiée (*dirty bit*)
- Prot : 1 $\rightarrow$ lecture seule (parfois 3 bits)
- Pres. : Présente (mémoire)
- Cadre de page : Emplacement

Accélération

Table des pages : obligations

- Correspondance rapide
- Extrêmement grande

Accélération

Table des pages : obligations

- Correspondance rapide
- Extrêmement grande

Exemple

Espace d'adressage : 32 bits Taille de page : 4Ko (2^{12})

$\Rightarrow$ 1 million de pages

Accélération

Table des pages : obligations

- Correspondance rapide
- Extrêmement grande

Exemple

Espace d'adressage : 32 bits Taille de page : 4Ko (2^{12})

$\Rightarrow$ 1 million de pages par processus

Accélération

Table des pages : obligations

- Correspondance rapide
- Extrêmement grande

Exemple

Espace d'adressage : 32 bits Taille de page : 4Ko (2^{12})

$\Rightarrow$ 1 million de pages par processus

En général la table de traduction est en mémoire
1 registre est utilisé pour mémoriser l'adresse initiale de la table
Ralentissement des performances : 1 accès mémoire = 2 accès minimum pour effectuer l'accès réel

TLB : Mémoire associative

Principe

En règle général un programme accède régulièrement quelques zones mémoire

Pour améliorer les performances de la MMU il faut lui adjoindre un cache spécialisé

TLB : Mémoire associative

Principe

En règle général un programme accède régulièrement quelques zones mémoire
Pour améliorer les performances de la MMU il faut lui adjoindre un cache spécialisé

Mémoire associative

La mémoire associative ou TLB (*Translation Lookaside Buffer*) assiste la MMU pour trouver les enregistrements
Elle contient des enregistrements $(8 - 64)$ de la table de traduction
Lors d'une requête de page, elle vérifie d'abord son contenu
Si la page est absente, elle la mémorise en remplaçant une page antérieure.
Met à jour la table.

Échange

Lorsque la mémoire est saturée, la mémoire virtuelle est amenée à stocker des pages sur disque

Échange

Lorsque la mémoire est saturée, la mémoire virtuelle est amenée à stocker des pages sur disque

Libération d'une page mémoire

1) Déterminer la page à remplacer
2) Le mettre à jour sur disque si nécessaire

Les algorithmes de sélection de la page à remplacer peuvent s'appliquer à d'autres domaines (en particulier au *buffer cache* du disque)

Échange

Lorsque la mémoire est saturée, la mémoire virtuelle est amenée à stocker des pages sur disque

Libération d'une page mémoire

1) Déterminer la page à remplacer
2) Le mettre à jour sur disque si nécessaire

Les algorithmes de sélection de la page à remplacer peuvent s'appliquer à d'autres domaines (en particulier au *buffer cache* du disque)

De façon orthogonal au choix de la page à remplacer il faut déterminer si on remplace une page

- de n'importe quel processus
- du processus courant

Algorithme optimal

Référence

De façon à pouvoir évaluer la performance d'un algorithme, on effectue la comparaison par rapport à un algorithme qui aurait connaissance des requêtes à venir

Algorithme optimal

Référence

De façon à pouvoir évaluer la performance d'un algorithme, on effectue la comparaison par rapport à un algorithme qui aurait connaissance des requêtes à venir

Impossible à implémenter !

Algorithme optimal

Référence

De façon à pouvoir évaluer la performance d'un algorithme, on effectue la comparaison par rapport à un algorithme qui aurait connaissance des requêtes à venir

Impossible à implémenter !

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

Algorithme optimal

Référence

De façon à pouvoir évaluer la performance d'un algorithme, on effectue la comparaison par rapport à un algorithme qui aurait connaissance des requêtes à venir
Impossible à implémenter !

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

1
2
3
4

4 défauts de page

Algorithme optimal

Référence

De façon à pouvoir évaluer la performance d'un algorithme, on effectue la comparaison par rapport à un algorithme qui aurait connaissance des requêtes à venir

Impossible à implémenter !

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

1
2
3
5

5 défauts de page

Algorithme optimal

Référence

De façon à pouvoir évaluer la performance d'un algorithme, on effectue la comparaison par rapport à un algorithme qui aurait connaissance des requêtes à venir
Impossible à implémenter !

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

4
2
3
5

6 défauts de page

Algorithme optimal

Référence

De façon à pouvoir évaluer la performance d'un algorithme, on effectue la comparaison par rapport à un algorithme qui aurait connaissance des requêtes à venir

Impossible à implémenter !

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

4
2
3
5

6 défauts de page au mieux

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

1
2
3
4

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

5
2
3
4

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

5
1
3
4

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

5
1
2
4

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

5
1
2
3

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

4
1
2
3

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

4
5
2
3

Allocation FIFO

Algorithme FIFO

Les pages sont stockés dans une file. Lorsqu'on a besoin d'espace pour une nouvelle page, on élimine celle qui est en tête de file, et place la nouvelle en queue de file

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

4
5
2
3

10 défauts de page

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
3 pages mémoire

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 pages mémoire

1
2
3

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 pages mémoire

4
2
3

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 pages mémoire

4
1
3

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 pages mémoire

4
1
2

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 pages mémoire

5
1
2

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
3 pages mémoire

5
3
2

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 pages mémoire

5
3
4

Anomalie de Belady (1969)

En 1969 Lásló Bélády a démontré que plus de mémoire pouvait induire plus de défauts de page en FIFO

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 pages mémoire

5
3
4

9 défauts de page

Algorithme : Non récemment utilisé

Préambule

Dans la table de traduction chaque page possède deux bits R et M qui représentent le fait d'être accédé (lu/écrit) ou modifié
(Mis à jour à chaque référence $\Rightarrow$ matériel)
Remise à zéro périodique de R quoi qu'il arrive

Algorithme NRU

1) Les pages sont triées en 4 groupes :
   - 0 – Non-accédée Non-modifiée
   - 1 – Non-accédée Modifiée
   - 2 – Accédée Non-modifiée
   - 3 – Accédée Modifiée
2) Tirage au sort de la page à supprimer dans le groupe le plus petit

Algorithme : deuxième chance

Fondé sur FIFO

1) À partir de la tête de la file
2) Vérifie le bit R de la page :
    - S'il vaut 0, la page est remplacée
    - S'il vaut 1, le mettre à zéro, passer la page en question en fin de file

Algorithme : deuxième chance

Fondé sur FIFO

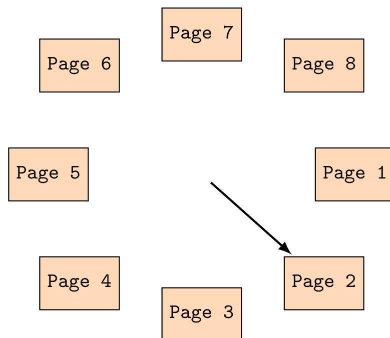
1) À partir de la tête de la file
2) Vérifie le bit R de la page :
   - S'il vaut 0, la page est remplacée
   - S'il vaut 1, le mettre à zéro, passer la page en question en fin de file

Nécessite des modifications fréquentes de la file
Performances médiocre

Algorithme : horloge

Fondé sur deuxième chance

Même fonctionnement que deuxième chance, mais la file est remplacée par une liste circulaire (le dernier élément pointe sur le premier)



Algorithme : LRU

Algorithme LRU (*Least Recently Used*

À chaque défaut de page on recherche la page qui a été utilisé le moins récemment

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

Algorithme : LRU

Algorithme LRU (*Least Recently Used*

À chaque défaut de page on recherche la page qui a été utilisé le moins récemment

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

1
2
3
4

Algorithme : LRU

Algorithme LRU (*Least Recently Used*

À chaque défaut de page on recherche la page qui a été utilisé le moins récemment

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

1
2
5
4

Algorithme : LRU

Algorithme LRU (*Least Recently Used*

À chaque défaut de page on recherche la page qui a été utilisé le moins récemment

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 pages mémoire

1
2
5
3

Algorithme : LRU

Algorithme LRU (*Least Recently Used*

À chaque défaut de page on recherche la page qui a été utilisé le moins récemment

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

1
2
4
3

Algorithme : LRU

Algorithme LRU (*Least Recently Used*

À chaque défaut de page on recherche la page qui a été utilisé le moins récemment

Exemple

Suite de références : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
4 pages mémoire

5
2
4
3

8 défauts de page

Implémentation LRU

Constat

- LRU coûteux
- Complexe
- Performant

Implémentation LRU

Constat

- LRU coûteux
- Complexe
- Performant

Implémentation

Nombreux choix d'implémentation

- Compteur d'instructions CPU
- Liste doublement chaînée
- Matrice $n \times n$ (pour n cadres de page) : Chaque accès à une page $\Rightarrow$ mise à 1 de sa ligne, à 0 de sa colonne
  Ligne la plus petite = page la moins récemment utilisée

Échange effectif

Approche naïve

Lors du remplacement d'une page :

- Sauvegarder la page éliminée sur le disque
- Charger la nouvelle page en mémoire

2 accès disque par défaut de page

Échange effectif

Approche naïve

Lors du remplacement d'une page :

- Sauvegarder la page éliminée sur le disque
- Charger la nouvelle page en mémoire

2 accès disque par défaut de page

Meilleur approche

Conserver en permanence un petit nombre de cadres de pages vides

Lors du remplacement d'une page :

- Charger la nouvelle page en mémoire dans un cadre vide
- Reprendre l'exécution
- Ajouter la page au stock de pages libres
- Occasionnellement on pourra même récupérer une page directement depuis les pages libres

Thrashing (Écroulement/effondrement)

Observation

Il arrive que la mémoire virtuelle devienne inutile

- On aimerait une mémoire vive de la taille du disque
- On possède une mémoire vive à la vitesse du disque !

Thrashing (Écroulement/effondrement)

Observation

Il arrive que la mémoire virtuelle devienne inutile

- On aimerait une mémoire vive de la taille du disque
- On possède une mémoire vive à la vitesse du disque !

Quand ?

- Jamais la même page utilisée
- 1 processus colossal (taille >> mémoire)
- Grand nombre de processus dont la taille totale >> mémoire

Thrashing (Écroulement/effondrement)

Observation

Il arrive que la mémoire virtuelle devienne inutile

- On aimerait une mémoire vive de la taille du disque
- On possède une mémoire vive à la vitesse du disque !

Quand ?

- Jamais la même page utilisée
- 1 processus colossal (taille $>>$ mémoire)
- Grand nombre de processus dont la taille totale $>>$ mémoire

Il est possible d'apporter une solution au dernier problème

Ensemble de travail

Principe

Considérer que pour sa bonne exécution un processus nécessite un nombre déterminé de pages

Associer à chaque processus un ensemble de pages (k) qui définit son ensemble de travail

Effectuer le chargement préalable de son ensemble de travail lorsqu'un processus obtient la main (au lieu des défauts de pages)

L'ensemble de travail est constitué des k dernières pages chargés accédées

Ensemble de travail

Principe

Considérer que pour sa bonne exécution un processus nécessite un nombre déterminé de pages

Associer à chaque processus un ensemble de pages (k) qui définit son ensemble de travail

Effectuer le chargement préalable de son ensemble de travail lorsqu'un processus obtient la main (au lieu des défauts de pages)

L'ensemble de travail est constitué des k dernières pages chargés accédées

Schéma d'algorithme d'échange

Lorsqu'il est nécessaire de remplacer une page, on élimine une page qui n'est pas dans l'ensemble de travail du processus courant

Ensemble de travail

Principe

Considérer que pour sa bonne exécution un processus nécessite un nombre déterminé de pages
Associer à chaque processus un ensemble de pages (k) qui définit son ensemble de travail
Effectuer le chargement préalable de son ensemble de travail lorsqu'un processus obtient la main (au lieu des défauts de pages)
L'ensemble de travail est constitué des k dernières pages chargés accédées

Schéma d'algorithme d'échange

Lorsqu'il est nécessaire de remplacer une page, on élimine une page qui n'est pas dans l'ensemble de travail du processus courant

Problème : Il est difficile d'implémenter précisément les ensembles de travail $\Rightarrow$ les pages accédées lors des k dernières ms

Algorithme : WSClock

Algorithme

Les pages sont placées dans une liste circulaire

- Examiner la page
- Si elle est dans l'ensemble de travail on passe (comparer le label de dernière modification avec le temps actuel moins k)
- Si elle n'y est pas et qu'elle n'a pas été modifiée la remplacer
- Si elle a été modifiée la placer dans la liste des pages à écrire (ordonanceur) passer à la suivante

On définit des limites au nombre de pages à écrire

Synthèse

Algorithmes d'éviction

Algorithme	Caractéristique
FIFO	Peu performant
NRU	Simple
Seconde chance	Simple peu efficace
Horloge	Simple efficace
LRU	Complexe, proche optimal
WSClock	Sophistiqué, efficace

Synthèse

Algorithmes d'éviction

Algorithme	Caractéristique
FIFO	Peu performant
NRU	Simple
Seconde chance	Simple peu efficace
Horloge	Simple efficace
LRU	Complexe, proche optimal
WSClock	Sophistiqué, efficace

Horloge et WSClock sont utilisés en pratique

NRU simple a aussi de bonne performance pour une implémentation très simple. Il n'est pas victime de l'effet Bélády

Segmentation

Autre approche de la gestion mémoire

De façon orthogonale à l'utilisation de page mémoire, il est possible de définir des segments de mémoire dans les processus

Exemple : permet une séparation fine des données et du code

Mis en œuvre par le matériel cela permet aussi de sécuriser les accès

Segmentation

Autre approche de la gestion mémoire

De façon orthogonale à l'utilisation de page mémoire, il est possible de définir des segments de mémoire dans les processus
Exemple : permet une séparation fine des données et du code
Mis en œuvre par le matériel cela permet aussi de sécuriser les accès

Les processeurs modernes (e.g., x86) mettent en œuvre à la fois la pagination et la segmentation

Optimisation : Copy-on-write

Copy-on-write

Dans la gestion de la mémoire on est souvent amené à réaliser des copies

Le mécanisme copy-on-write consiste à n'effectuer réellement la copie que lorsqu'elle est modifiée

Optimisation : Copy-on-write

Copy-on-write

Dans la gestion de la mémoire on est souvent amené à réaliser des copies

Le mécanisme copy-on-write consiste à n'effectuer réellement la copie que lorsqu'elle est modifiée

Exemple

Lors d'un appel à `fork`, seul un nouvel enregistrement est fait dans la table des processus (avec un nouveau PCB)

La mémoire utilisée initialement par le processus n'est pas dupliqué

Optimisation : Copy-on-write

Copy-on-write

Dans la gestion de la mémoire on est souvent amené à réaliser des copies
Le mécanisme copy-on-write consiste à n'effectuer réellement la copie que lorsqu'elle est modifiée

Exemple

Lors d'un appel à `fork`, seul un nouvel enregistrement est fait dans la table des processus (avec un nouveau PCB)
La mémoire utilisée initialement par le processus n'est pas dupliqué

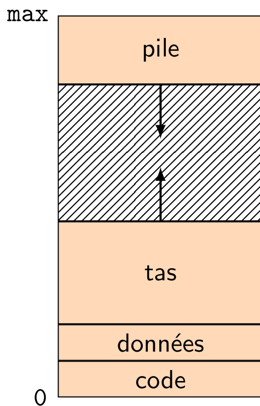
Systèmes de fichiers

Le système de fichier `btrfs` (prononcé *ButterFS*) repose sur le principe de copy-on-write

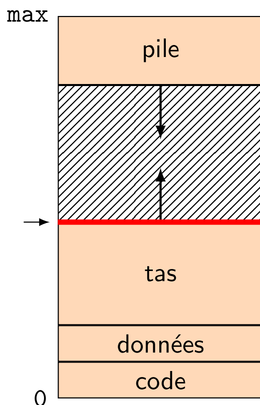
Progression

1. Mémoire virtuelle (premiers pas)
2. Mémoire virtuelle (moderne)
   - Pagination
   - Algorithmes d'échange
3. Appels systèmes

Structure mémoire ordinaire



Structure mémoire ordinaire



Breakpoint

Limite supérieure du tas

Les adresses entre breakpoint et la piles sont invalides

Appels vers `break point`

Deux appels systèmes

- `int brk(void *addr);`
  `addr` : nouvelle adresse du breakpoint
  Résultat : 0 en cas de succès
  -1 en cas d'erreur (positionne `errno`)

Appels vers `break point`

Deux appels systèmes

- `int brk(void *addr);`
  `addr` : nouvelle adresse du breakpoint
  Résultat : 0 en cas de succès
  −1 en cas d'erreur (positionne `errno`)
- `void *sbrk(intptr_t increment);`
  `increment` : nombre d'octets d'augmentation
  Résultat : Valeur antérieure du breakpoint

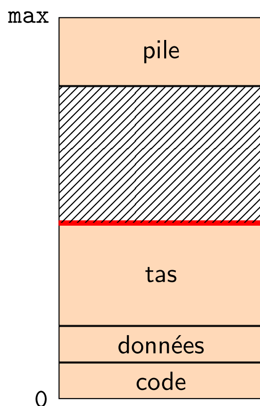
Appels vers `break point`

Deux appels systèmes

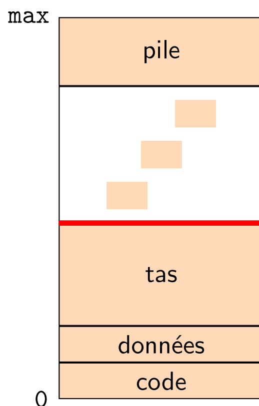
- `int brk(void *addr);`
  `addr` : nouvelle adresse du breakpoint
  Résultat : 0 en cas de succès
  -1 en cas d'erreur (positionne `errno`)
- `void *sbrk(intptr_t increment);`
  `increment` : nombre d'octets d'augmentation
  Résultat : Valeur antérieure du breakpoint

Il est possible d'écrire `malloc`, **très simplement**, en utilisant ces fonctions
`free` est plus difficile à coder (on doit gérer l'apparition de "trous")

Espace entre la pile et le tas



Espace entre la pile et le tas



Il est possible d'obtenir des adresses au delà du breakpoint
Il s'agit de fichiers placés en mémoire

Appel mmap

mmap

```
void *mmap(void *addr, size_t length, int prot,
    int flags, int fd, off_t offset);
```

- `addr` Choix d'adresse – `NULL` ⇒ au choix
- `length` Taille de l'espace
- `prot` Protection (OR) : `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`
- `flags` `MAP_SHARED`, `MAP_PRIVATE`, `MAP_ANONYMOUS`
- `fd` descripteur vers un fichier
- `offset` Point de départ dans le fichier (multiple taille de page)

Dans le cas anonyme : `fd = -1`, `offset` est ignoré, l'espace initial est rempli de 0

Mémoire partagée

- `int shm_open(const char *name, int oflag, mode_t mode);` Créer et ouvrir un nouvel objet en mémoire, ou ouvrir un objet existant. Elle est analogue à open(2). La fonction renvoie un descripteur de fichiers qui pourra être utilisé par les appels `mmap` et `ftruncate`.

Mémoire partagée

- `int shm_open(const char *name, int oflag, mode_t mode);`
  Créer et ouvrir un nouvel objet en mémoire, ou ouvrir un objet existant. Elle est analogue à open(2). La fonction renvoie un descripteur de fichiers qui pourra être utilisé par les appels `mmap` et `ftruncate`.
- `int ftruncate(int fd, off_t length);`
  Définir la taille de l'objet en mémoire partagée. (Un objet nouvellement créé en mémoire partagée a une taille nulle.)

Mémoire partagée

- `int shm_open(const char *name, int oflag, mode_t mode);`
  Créer et ouvrir un nouvel objet en mémoire, ou ouvrir un objet existant. Elle est analogue à open(2). La fonction renvoie un descripteur de fichiers qui pourra être utilisé par les appels `mmap` et `ftruncate`.
- `int ftruncate(int fd, off_t length);`
  Définir la taille de l'objet en mémoire partagée. (Un objet nouvellement créé en mémoire partagée a une taille nulle.)
- `int shm_unlink(const char *name);`
  Supprimer le nom d'un objet en mémoire partagée.

Autres appels

- `int msync(void *addr, size_t len, int flags);`
  Applique les changements effectués dans la mémoire à l'original (selon les indications `addr` et `len`)

Autres appels

- `int msync(void *addr, size_t len, int flags);`
  Applique les changements effectués dans la mémoire à l'original (selon les indications `addr` et `len`)
- `int munmap(void *addr, size_t len);`
  Supprime les pages dans la plage indiquée (`addr` doit être un multiple de la taille de page)

Autres appels

- `int msync(void *addr, size_t len, int flags);`
  Applique les changements effectués dans la mémoire à l'original (selon les indications `addr` et `len`)
- `int munmap(void *addr, size_t len);`
  Supprime les pages dans la plage indiquée (`addr` doit être un multiple de la taille de page)
- `int mprotect(void *addr, size_t len, int prot);`
  Modifie la protection vers `prot`

Autres appels

- `int msync(void *addr, size_t len, int flags);`
  Applique les changements effectués dans la mémoire à l'original (selon les indications `addr` et `len`)
- `int munmap(void *addr, size_t len);`
  Supprime les pages dans la plage indiquée (`addr` doit être un multiple de la taille de page)
- `int mprotect(void *addr, size_t len, int prot);`
  Modifie la protection vers `prot`
- `int mincore(void *addr, size_t len, char *vec);`
  Positionne dans `vec` les pages du processus résidente à l'instant, à partir de l'adresse `addr`
  Taille de `vec` :`(len+PAGE_SIZE-1) / PAGE_SIZE`
  Le bit de poid faible de chaque octet de `vect` correspond à la présence de la page