Commencé le	mardi 25 avril 2023, 13:30
État	Terminé
Terminé le	mardi 25 avril 2023, 15:59
Temps mis	2 heures 29 min
Note	**15,58** sur 20,00 (**77,9**%)

Question **1**

Non répondue

Non noté

Directives :

1. Cet examen est composé de 11 questions au total (Q1 à Q11) pour une durée totale de 2 heures 30 minutes.
2. Pondération 45%.
3. En guise de documentation, vous aurez accès à votre compte Moodle et, par conséquent, à la totalité du site du cours INF2610, incluant les diapos, etc. Aucune autre documentation ne sera permise. Les ordinateurs personnels, tablettes, calculatrices et cellulaires ne sont pas permis.
4. Aucune réponse aux questions durant l'examen.
5. Tous les appels système utilisés dans les questions sont supposés exempts d'erreurs et considèrent leurs options par défaut.
6. Il n'est pas demandé de traiter les cas d'erreurs, ni d'inclure les directives d'inclusion dans les codes à compléter.
7. Pour les questions à choix multiples, **vous devez sélectionner une seule réponse**.
8. Il n'est pas possible de joindre des fichiers à vos réponses.
9. Lisez au complet chaque question avant d'y répondre.
10. Vous pouvez inclure vos commentaires au responsable du cours dans l'espace ci-après.

Bonne fin de session à tous!

Question **2**

Terminé

Non noté

Sur mon honneur, j'affirme que je compléterai cet examen en vertu du code de conduite de l'étudiant de Polytechnique Montréal et de sa politique sur le plagiat. J'affirme également que je compléterai cet examen par moi-même, sans communication avec personne, et selon les directives diffusées sur les canaux de communication.

Écrivez votre nom complet ainsi que votre matricule en guise d'approbation dans la zone de texte ci-dessous.

[redacted]

Question **3**

Terminé

Note de 1,33 sur 2,00

[Q1]

1-Donnez l'arborescence des processus créés par le code ci-dessous (supposez que les appels système ne retournent pas d'erreur). **Indiquez** la séquence d'instructions exécutées par chacun des processus.

2- Donnez aussi tous les ordres possibles d'affichages des chiffres (6, 7, 8 et 9).

3- Quelle(s) est (sont) la (les) valeur(s) de *x* affichée(s) à l'écran ?
Justifiez vos réponses.

```
int x=0;
```

```
int main() {
  x=x+1;
  if (fork() == 0) {
    x=x+3;
    if (fork() == 0) { x=x+1; write(1,"9",1); exit(0);}
    while(wait(NULL)>0);
    write(1,"6",1);
    exit(0);
  } else {
    x=x+2;
    if (fork() == 0) { x=x+1; write(1,"7",1); exit(0); }
    while(wait(NULL)>0);
    write(1,"8",1);
  }
  printf("\n x=%d\n",x);
  exit(0);
}
```

---

1. Le processus principal crée deux enfants, et le premier enfant crée un enfant.

Séquence du grand-parent :

```
x = x + 1;
if (fork() == 0) {
} else {
x = x + 2;
if (fork() == 0)
while (wait(NULL)>0);
write(1, "8", 1);
}
printf("\n x=%d\n",x);
```

```
exit(0);
```

Séquence du premier enfant :

```
if (fork() == 0) {
x = x + 3;
if (fork() == 0)
while(wait(NULL)>0);
write(1,"6",1);
exit(0);
```

Séquence du deuxième enfant :

```
if (fork() == 0) { x=x+1; write(1,"7",1); exit(0); }
```

Séquence du petit-enfant :

```
if (fork() == 0) { x=x+1; write(1,"7",1); exit(0); }
```

2. Les ordres d'affichage possibles sont :

9678, 7896, 9768, 9786, 7968, 7986

Les seules contraintes sont que 9 s'affiche avant 6 et que 7 s'affiche avant 8.

3. Le seul processus qui va se rendre au printf qui affiche la valeur de x à l'écran est le processus original (le grand-parent). En sachant que chaque processus a sa propre copie de la variable globale x, on peut donc exécuter toutes les opérations qui modifient x dans la séquence du grand-parent pour voir quelle valeur sera affichée :

```
int x = 0;
x = x + 1; // x = 1
x = x + 2; // x = 3
```

La valeur de x affichée sera donc 3.

Commentaire :

2) Mauvaise réponse.

Réponse attendue :

• Le parent attend la fin de ses enfants, puis affiche 8. Donc, 8 doit sortir en dernier dans tous les cas.

• Le premier enfant du parent attend la fin de son enfant à lui (le petit-enfant), puis affiche 6. Le petit-enfant affiche 9. Donc, 9 doit toujours précéder 6.

Les combinaisons suivantes sont donc valides : 7968, 9768, 9678.

Question **4**

Terminé

Note de 1,00 sur 1,00

[Q2]

Donnez le nombre de processus créés par l'interpréteur de commandes (shell), lorsqu'on lui soumet la commande suivante : "*cat f1 f2 | sort > f3*", où *f1, f2* et *f3* sont des fichiers ordinaires. **Indiquez** aussi l'entrée standard, la sortie standard et la sortie erreur pour chacun des processus créés.

On demande le nombre de processus créés, donc on ne prend pas en compte le processus déjà existant du shell (P0).

Le shell va créer deux processus : un pour exécuter la commande cat (P1) et un pour exécuter la commande sort (P2).

Voici les entrées et sorties de chaque processus créé :

P1:

stdout : Un tube anonyme (T1)
stdin : Le terminal de l'utilisateur
stderr : Le terminal de l'utilisateur

P2:

stdout : Le fichier f3
stdin : Le tube anonyme T1
stderr : Le terminal de l'utilisateur

Quand je dis le terminal de l'utilisateur, cela signifie que la sortie s'affichera sous forme textuelle, et que les caractères tapés par l'utilisateur seront envoyés à l'entrée.

Commentaire :

Question **5**

Terminé

Note de 2,00 sur 2,00

[Q3] (2 pts)

Considérez les deux threads concurrents *A* et *B* suivants d'un même processus :

```
/*0*/ Sémaphore SA=0, SB=0;
A {                          B {
 while(1) {                   while(1) {
  /*1*/                        /*4*/
  a1();                       b1();
  /*2*/                        /*5*/
  a2();                       b2();
  /*3*/                        /*6*/
 }                            }
}                            }
```

Synchronisez, en utilisant les sémaphores *SA* et *SB*, les threads *A* et *B* pour forcer, à chaque cycle, (1) l'exécution de la fonction *a1* avant celle de *b2* et (2) l'exécution de *b2* avant celle de *a2*.

Attention :

- Aucune autre relation de précédence entre les fonctions ne doit être forcée.
- Vous devez utiliser le sémaphore *SA* pour bloquer/ débloquer le thread *A*.
- Vous devez utiliser le sémaphore *SB* pour bloquer/ débloquer le thread *B*.

Pour répondre à cette question, sélectionnez les instructions à insérer aux endroits appropriés. Sélectionnez "*rien*", s'il n'y a aucune instruction à insérer.

/*1*/	rien
/*2*/	V(SB); P(SA);
/*3*/	rien
/*4*/	rien
/*5*/	P(SB);
/*6*/	V(SA);

Votre réponse est correcte.

La réponse correcte est :

/*1*/ → rien,

/*2*/ → V(SB); P(SA);,

/*3*/ → rien,

/*4*/ → rien,

/*5*/ → P(SB);,

/*6*/ → V(SA);

Question **6**

Terminé

Note de 2,00 sur 2,00

[Q4] (2 pts)

Considérez le moniteur ProducteursConsommateurs vu en classe :

```
Moniteur ProducteursConsommateurs {
      const int N=100;  // taille du tampon
       int tampon[N];
       int compteur=0, ic=0, ip=0;
       boolc nplein, nvide;
       void deposer (int objet) { // section critique pour le dépôt
                while(compteur==N) wait(nplein);
                tampon[ip] = objet;
                ip = (ip+1)%N;
                compteur++;
                signal(nvide);
       }
      void retirer (int& objet) { //section critique pour le retrait
                  while(compteur==0) wait(nvide);
                  objet = tampon[ic];
                  ic = (ic+1)%N;
                  compteur--;
                  signal(nplein);
        }
}
```

Supposez que N=3 et qu'il y a exactement deux producteurs (P1 et P2) et un seul consommateur (C1) qui partagent une variable du type : Moniteur ProducteursConsommateurs.
Une famine est-elle possible pour l'un des deux producteurs (c-à-d une demande de production qui est retardée indéfiniment) ? Justifiez votre réponse.

Si on utilise des variables de condition "Signal-and-Continue", on peut avoir le scénario suivant :

- Le tampon est rempli à 4 / 5 (on suppose N = 5)
- P1 dépose un object (le tampon est donc plein)
- P2 essaye de déposer un objet, il est mis en attente passive à wait(nplein);
- C1 commence à consommer
- P1 essaye de déposer un objet, il est mis en attente du moniteur
- C1 termine de consommer, et envoie un signal à nplein. P2 est mis en attente du moniteur derrière P1
- P1 dépose son deuxieme objet (le tampon est donc plein)
- P2 reprend son exécution, mais le tampon est plein, il se met donc en attente passive à wait(nplein);
- Et ainsi de suite...

P2 ne pourra jamais déposer d'objet si P1 essaye de déposer un objet pendant que C1 est en train de consommer.

Si on utilise des variables de condition "Signal-and-Wait", on n'a plus ce problème, car P2 pourra s'exécuter dès que signal(nplein) est reçu, et donc avant P1.

1- P1 produit M[P2] compteur=1

2 - P2 produit M[P1] compteur=2

3 - P1 produit M[P2] compteur =3

4- P2 --> nplein[P2] M[C1,P1]

5- C1 consomme --> nplein[], M[P1,P2] compteur=2

6 - P1 produit M[P2] compteur=3 on retombe sur l'état atteint par l'étape 3. Il suffit de répéter 4, 5 et 6 pour obtenir un scénario où la demande production de P2 n'est jamais complétée.

Commentaire :

Question **7**

Terminé

Note de 1,50 sur 2,00

[Q5] (2.5 pts)

On vous présente le pseudo-code d'un moniteur qui définit un algorithme alternatif pour le problème des philosophes.

```
#define N 5
#define G ((i-1)%5)
#define D ((i+1)%5)
#define affame 0
#define enTrainDeManger 1
#define enTrainDePenser 2

Moniteur Philosophes {
  int etat[N];
  boolc philosophe[N];

  void prendreFourchettes(int i) {
    etat[i] = affame;
    test(i);
    if(etat[i] != enTrainDeManger) wait(philosophe[i]);
  }

  void deposerFourchettes(int i) {
    etat[i] = enTrainDePenser;
    test(D);
    test(G);
  }

  void test(int i) {
    if(etat[D] != enTrainDeManger && etat[G] != enTrainDeManger && etat[i] == affame) {
      etat[i] = enTrainDeManger;
      signal(philosophe[i]);
    }
  }

  void init() {
    for(int i=0; i<N; i++)
      etat[i] = enTrainDePenser;
  }
};
```

On vous demande de répondre aux questions suivantes:
1 - [0.5 pt] Quelle notion théorique en synchronisation est-elle représentée par le terme *boolc*?
2 - [0.5 pt] L'exclusion mutuelle est-elle assurée convenablement même si on ne trouve aucun mutex? Si oui, expliquez comment. Sinon, indiquez la manière la plus simple d'incorporer l'exclusion mutuelle dans ce pseudo-code.
3 - [0.5 pt] Quel est le but des appels à *test(D)* et *test(G)* dans la méthode *deposerFourchettes*? Expliquez en détail.
4 - [0.5 pt] La majorité de ce pseudo-code est composée d'instructions C++. Votre collègue vous propose de compléter cette traduction vers le C++ afin d'obtenir un code compilable et exécutable qui fournira la fonctionnalité recherchée. Est-ce une bonne idée? Expliquez. Votre réponse n'a pas à fournir de code.

1. boolc permet de représenter le type d'une "variable de condition", qui permet de passer un processus dans un moniteur en attente passive en utilisant wait pour éviter d'avoir un processus en attente active qui empêche les autres processus d'accéder au moniteur et ainsi libérer l'attente.

2. En ce moment, en supposant que l'exclusion mutuelle ne soit pas gérée par le langage de programmation directement, il n'y a rien qui empêche deux processus d'accéder aux méthodes du moniteur en même temps. Pour résoudre ce problème, on pourrait ajouter un mutex binaire aux attributs du moniteur, puis appeler P(mutex) au début et V(mutex) à la fin de chaque méthode du moniteur, pour s'assurer que seulement un processus à la fois puisse utiliser le moniteur.

3. Les appels à test(D) et test(G) permettent au philosophe qui a terminé de manger de laisser un de ses voisins manger s'il le voulait. Par exemple, si le philosophe 1 dépose ses fourchettes et que le philosophe 2 est affamé, alors le moniteur va passer l'état du philosophe 2 à "en train de manger" et va envoyer un signal pour que l'appel du philosophe 2 à "prendreFourchettes" puisse se terminer. Par contre, si on essaye de déposer les fourchettes d'un philosophe qui était déjà en train de penser, la condition dans test va échouer et on ne va pas laisser les philosophes voisins manger.

4. Je pense que compléter la traduction vers le C++ est une mauvaise idée pour plusieurs raisons :

- Le code actuel ne marche que pour un nombre de philosophes égal à 5, et la façon dont les #define ont été faits n'est pas très flexible
- Il faut implémenter la fonctionnalité de "boolc", qui n'existe pas directement en C++ (même si on pourrait assez facilement utiliser des sémaphores pour ça)
- Le code présenté semble ne pas fonctionner. Par exemple, si le philosophe 0 et le philosophe 2 sont en train de manger, et que le philosophe 1 est affamé, alors si le philosophe 0 appelle "deposerFourchettes", le philosophe 1 va passer à l'état "en train de manger", alors qu'il n'y a pas de couverts disponibles à sa droite.

Commentaire :

2) Justification incorrecte.

Justification attendue :

L'exclusion mutuelle est assurée par l’essence même de la notion de moniteur. Celui-ci garantit qu’un seul thread, tout au plus, puisse être présent dans l’objet à la fois.

Question **8**

Terminé

Note de 1,50 sur 1,50

[Q6] (1.5 pt)

Dans un système, 3 processus partagent un certain nombre ***r*** de ressources identiques. Chaque processus a besoin au maximum de 20 ressources. Quelle est la valeur minimale de *r* qui permette de garantir qu'il n'y aura pas d'interblocage? Justifiez votre réponse à la page suivante.

Réponse : 58

La réponse correcte est : 58

Question **9**

Terminé

Non noté

Justifiez votre choix de réponse (si votre justification est absente ou erronée vous aurez **0 point** peu importe votre réponse dans le QCM).

Le but est de trouver combien de ressources sont nécessaires dans le pire scénario possible pour éviter les interblocages.

Si on pense aux matrices d'allocations et de requêtes, on peut obtenir le scénario où les trois processus possèdent 19 ressources, et donc il leur en manque 1 chacun pour s'exécuter. On voit tout de suite que le nombre de ressources disponibles doit être d'au moins 57 (19*3) plus 1 pour pouvoir fournir la ressource au premier processus qui demandera celle qui lui manque, donc 58. Si deux processus demandent leur ressource en même temps, l'un des deux l'obtient, se termine, puis la donne au deuxième.

Si un des trois processus possède 20 ressources, alors on peut résoudre l'interblocage avec l'algorithme du banquier car la valeur dans "Req" sera de 0. Si un processus possède moins de ressources mais en demande plus, il y aura un nombre plus élevé dans A, donc on pourra là aussi résoudre l'interblocage.

Par exemple, si le premier processus possède 18 ressources, alors le nombre de ressources disponibles sera (58-19-19-18)=2, on aura donc assez de ressources pour utiliser l'algorithme du banquier. Si un autre processus possède lui aussi moins de ressources, c'est encore plus facile.

Question **10**

Terminé

Note de 2,00 sur 2,00

[Q7] (2 pts)

Dans un système, *4* processus *(P1, P2, P3 et P4)* partagent, en exclusion mutuelle, *3* types de ressources *(R1,R2 et R3)* en quantités respectives *(5, 4, 6)*. Supposez l'état courant du système :

Matrice Alloc des ressources allouées :

Alloc :

	R1	*R2*	*R3*
P1	*0*	*1*	*2*
P2	*1*	*0*	*0*
P3	*3*	*2*	*1*
P4	*0*	*0*	*0*

Matrice *Req* des ressources requises mais non encore acquises :

Req :

	R1	*R2*	*R3*
P1	*1*	*1*	*1*
P2	*2*	*1*	*3*
P3	*1*	*1*	*4*
P4	*2*	*0*	*5*

1- Est-ce que cet état est sûr ? Justifiez votre réponse en déroulant pas-à-pas l'algorithme du banquier.

Oui

2- Supposez que le système utilise l'algorithme du banquier pour éviter les interblocages. Le système reçoit de la part du processus P2, une demande de 2 ressources de type *R3*. Le système va-t-il accepter cette demande ? Justifiez votre réponse en expliquant clairement toutes les étapes de l'analyse de la demande.

Non

Utilisez la page suivante pour justifier vos réponses. Les réponses non justifiées ne seront pas considérées.

1- Est-ce que cet état est sûr ? Justifiez votre réponse en déroulant pas-à-pas l'algorithme du banquier.

Cet état est sûr. Au départ :

E	R1	R2	R3
	5	4	6
ALLOC	R1	R2	R3
P1	0	1	2
P2	1	0	0
P3	3	2	1
P4	0	0	0
REQ	R1	R2	R3

P1	1	1	1
P2	2	1	3
P3	1	1	4
P4	2	0	5
A	R1	R2	R3
	1	1	3

On prend P1, ce qui mène à :

E	R1	R2	R3
	5	4	6
ALLOC	R1	R2	R3
P2	1	0	0
P3	3	2	1
P4	0	0	0
REQ	R1	R2	R3
P2	2	1	3
P3	1	1	4
P4	2	0	5
A	R1	R2	R3
	1	2	5

On prend P3, ce qui mène à :

E	R1	R2	R3
	5	4	6
ALLOC	R1	R2	R3
P2	1	0	0
P4	0	0	0
REQ	R1	R2	R3
P2	2	1	3
P4	2	0	5
A	R1	R2	R3
	4	4	6

Par la suite, on peut indifféremment prendre P2 ou P4 et terminer.

2- Supposez que le système utilise l'algorithme du banquier pour éviter les interblocages. Le système reçoit de la part du processus P2, une demande de 2 ressources de type *R3*. Le système va-t-il accepter cette demande ? Justifiez votre réponse en expliquant clairement toutes les étapes de l'analyse de la demande.

Le système ne va pas accepter cette demande. Au départ :

E	R1	R2	R3
	5	4	6
ALLOC	R1	R2	R3
P1	0	1	2

P2	1	0	2
P3	3	2	1
P4	0	0	0
REQ	R1	R2	R3
P1	1	1	1
P2	2	1	1
P3	1	1	4
P4	2	0	5
A	R1	R2	R3
	1	1	1

On prend P1, ce qui mène à :

E	R1	R2	R3
	5	4	6
ALLOC	R1	R2	R3
P2	1	0	2
P3	3	2	1
P4	0	0	0
REQ	R1	R2	R3
P2	2	1	1
P3	1	1	4
P4	2	0	5
A	R1	R2	R3
	1	2	3

... et nous sommes bloqués.

Question **11**

Terminé

Non noté

Utilisez cette page pour justifier chacune de vos réponses (si votre justification est absente ou erronée vous aurez **0 point** pour cette question peu importe votre réponse dans le QCM).

1.

E = [5, 4, 6]
A = [1, 1, 3]

On prend le processus P1 car [1, 1, 1] <= [1, 1, 3]

A = [1, 2, 5]

On prend le processus P3 car [1, 1, 4] <= [1, 2, 5]

A = [4, 4, 6]

On prend le processus P4 car [2, 0, 5] <= [4, 4, 6]

A = [4, 4, 6]

On prend le processus P2 car [2, 1, 3] <= [4, 4, 6]

A = [5, 4, 6]

Tous les processus ont été marqués, l'état est sûr.

2. Supposons l'état où le processus P2 obtient les deux ressources de type R3 :

E = [5, 4, 6]
A' = [1, 1, 1]
Alloc' =

	R1	R2	R3
P1	0	1	2
P2	1	0	2
P3	3	2	1
P4	0	0	0

Req' =

	R1	R2	R3
P1	1	1	1
P2	2	1	1
P3	1	1	4
P4	2	0	5

On prend le processus P1 car [1, 1, 1] <= [1, 1, 1]

A' = [1, 2, 3]

Il n'y a aucun processus dont la ligne dans Req est <= à A', l'état n'est donc pas sûr, on refuse la demande.

Question **12**

Terminé

Note de 1,25 sur 3,00

[Q8] (3 pts)

Soit un système de gestion de la mémoire, à pagination pure, possédant les caractéristiques suivantes :

- une mémoire physique de 2 *MiO (= 2^{21} octets)*,
- une adresse virtuelle codée sur *24 bits*,
- des pages de 2^8 octets chacune, et
- des entrées de tables des pages (TDPs) de 16 (=2^4) octets chacune.

1 - [0.25 pt] Quelle est la taille maximale en nombre de pages de l'espace virtuel ?

2 - [0.25 pt] Quel est le nombre de cadres dans l'espace physique ?

3 - [0.25 pt] Quel est le nombre de bits nécessaires pour stocker le déplacement dans un cadre ?

4 - [0.25 pt] Quel est, en hexadécimal, le numéro de page référencé par l'adresse virtuelle *0x3F45B7*?

5 - [0.5 pt] Quelle est la taille en nombre de cadres de la TDP, dans le cas d'une TDP à un niveau ?

6 - [0.5 pt] Dans le cas d'une TDP à 2 niveaux, et à supposer que chaque table des pages du second niveau comporte 512 entrées, quelle est la taille en nombre de cadres i) de la table des pages du premier niveau, et ii) d'une table des pages du deuxième niveau?

7 - [0.5 pt] Dans le cas d'une TDP à 2 niveaux de la question précédente, quel est, en hexadécimal, le numéro de l'entrée, dans la table des pages du 1er niveau, référencée par l'adresse virtuelle *0x3F45B7* ?

8 - [0.5 pt] Toujours dans le cas d'une TDP à 2 niveaux de la question précédente, quelle est, en hexadécimal, l'adresse physique de l'adresse virtuelle *0x3F45B7*, si la page référencée est en mémoire dans le cadre 6 ?

Justifiez vos réponses.

1. On sait qu'il y a 2^(X-Y) si l'adressage est sur X bits et que chaque page fait 2^Y octets.

On a donc 2^(24-8) = 2^16 pages dans l'espace virtuel

2. Comme c'est un système à pagination pure, où la taille des pages est égale à la taille des cadres, il y a aussi 2^16 cadres dans l'espace physique.

3. Comme il y a 2^8 octets dans chaque cadre, il faut 8 bits pour pouvoir adresse chaque octet dans un cadre. Le nombre de bits nécessaires pour stocker le déplacement dans un cadre est donc de 8.

4. Si on utilise les 8 derniers bits pour le déplacement, alors on utilise les 24-8 = 16 premiers bits pour le numéro de la page. On prend donc les 4 premiers chiffres de l'adresse (16 bits = 2 octets = 4 chiffres hexadécimaux), ce qui nous donne le numéro de page 0x3F45.

5. Pour pouvoir adresse l'entièreté de la mémoire virtuelle, la table des pages doit pouvoir avoir une entrée pour chaque cadre, il faut donc que sa taille soit de 2^16 cadres.

6. Si chaque table du second niveau contient 512 entrées (2^9), alors il faut 2^(16-9) = 2^7 tables de deuxième niveau pour adresse tous les cadres. Il faut donc que la table de premier niveau contienne 2^7 cadres qui contiendront l'adresse des tables de deuxième niveau, et les tables de deuxième niveau contiendront 2^9 cadres pour contenir les 512 entrées.

7. Ce sont les 7 premiers bits qui contiendront le numéro de l'entrée dans la table de premier niveau. 0x3F = 0011 1111. Si on prend les 7 premiers bits, on obtient 0011 1111, soit 0x3F, qui est le numéro d'entrée dans la table de premier niveau.

8. Si la page est dans le cadre 6, alors on peut recréer l'adresse physique en ayant pour les 16 premiers bits la valeur "6", et pour l'alignement les 8 derniers bits de l'adresse virtuelle. On a donc 0x0006B7 pour l'adresse physique.

Commentaire :

2) Réponse incorrecte.

Réponse attendue :
2^(21-8) = 2^13 = 8 192 cadres

5) Réponse incorrecte.

Réponse attendue :
2^16 entrées * 16 octets = 1 048 576 octets = 4 096 cadres

6) i) Réponse incorrecte.

Réponse attendue :
On a 7 bits car la table de deuxième niveau doit en avoir 9 pour accommoder 512 entrées.
Donc : 2^7 entrées * 16 octets par entrée / 256 octets par cadre = 8 cadres

6) ii) Réponse incorrecte.

Réponse attendue :
2^9 entrées * 16 octets par entrée / 256 octets par cadre = 32 cadres

7) Réponse incorrecte.

Réponse attendue :
0x3F45B7 = 0b001111110100010110110111.
Les sept premiers bits sont 0b0011111 = 0x1F

Question **13**

Terminé

Note de 0,50 sur 0,50

[Q9] (0.5 PT)

Lequel des énoncés suivants est **vrai**?

- La segmentation est de moins en moins utilisée de nos jours sur les micro-ordinateurs, notamment parce que la pagination constitue un modèle de gestion de la mémoire considéré comme plus intéressant.
- Le va-et-vient consiste à retirer temporairement de la mémoire des portions de processus.
- On ne peut généralement pas démarrer des processus qui, collectivement, vont accaparer davantage de mémoire que la capacité mémoire physique de l'ordinateur.
- L'écroulement du système survient quand les espaces privés des processus ne sont plus protégés.
- On ne peut généralement pas faire fonctionner un ordinateur qui comporte moins de mémoire physique que ce qui est couvert par l'espace d'adressage.

Votre réponse est correcte.

La réponse correcte est :
La segmentation est de moins en moins utilisée de nos jours sur les micro-ordinateurs, notamment parce que la pagination constitue un modèle de gestion de la mémoire considéré comme plus intéressant.

Question **14**

Terminé

Note de 2,50 sur 2,50

[Q10] (2.5 pts)

Supposez un système monoprocesseur et les 5 processus (P1, P2, P3, P4 et P5) décrits dans le tableau suivant :

Processus	Date d'arrivée	Priorité	Temps d'exécution
P1	0	3	6 (2) 2
P2	1	2	4
P3	1	3	2 (2) 1
P4	2	3	3
P5	5	2	2

Ce système dispose d'un seul périphérique d'E/S qui gère les requêtes d'E/S selon la discipline FIFO. Le temps de commutation de contexte est nul. La priorité *2* est plus basse que la priorité *3*. Le temps d'exécution ***X (Y) Z*** d'un processus *Pi* signifie que l'exécution de *Pi* nécessite, dans l'ordre, *X* unités de temps CPU, *Y* unités de temps en E/S et *Z* unités de temps CPU.

Donnez le diagramme de Gantt montrant l'ordre d'exécution des *5* processus, dans le cas d'un ordonnancement préemptif à files multiples et priorités fixes. L'ordonnancement des processus de même priorité est circulaire avec un quantum égal à *3*. Pour répondre à cette question, **complétez le tableau suivant** :

0	P1
1	P1
2	P1
3	P3
4	P3
5	P4
6	P4
7	P4
8	P1
9	P1
10	P1
11	P3
12	P2
13	P1
14	P1
15	P5
16	P5
17	P2
18	P2

19	P2
20	rien
21	rien

Votre réponse est correcte.

La réponse correcte est :
0 → P1,

1 → P1,

2 → P1,

3 → P3,

4 → P3,

5 → P4,

6 → P4,

7 → P4,

8 → P1,

9 → P1,

10 → P1,

11 → P3,

12 → P2,

13 → P1,

14 → P1,

15 → P5,

16 → P5,

17 → P2,

18 → P2,

19 → P2, 20 → rien,

21 → rien

Question **15**

Terminé

Note de 0,00 sur 1,50

[Q11] (1.5 pt)

Un système temps réel gère *3* tâches **indépendantes périodiques** *(T1, T2* et *T3)* avec respectivement des périodes de *(x ms, x ms* et 20 *ms*). Supposez que les tâches *(T1, T2* et *T3)* aient besoin, au maximum, respectivement de *(4, 3* et *6 ms)* de temps processeur pour réaliser leurs traitements périodiques.

Quelle est la plus petite valeur de *x* pour laquelle il est possible d'ordonnancer ces trois tâches, selon un ordonnancement préemptif à priorités ? Justifiez votre réponse.

On suppose que l'ordonnancement utilisé est RMA (priorités inversement proportionnelles aux périodes).

Les tâches peuvent être ordonnées si :

(4/x) + (3/x) + (6/20) <= 3 * (2^(1/3) - 1). On estime 2^(1/3) entre 1.2 et 1.5

Avec x = 1ms, On a 4 + 3 + 0.3 = 7.3

3 * (1.5 - 1) = 3*0.5 = 1.5 << 7.3

Avec x = 2ms, 2 + 1.5 + 0.3 = 3.8 > 1.5

Avec x = 3ms, 7 / 3 + 0.3 = 2.63333 > 1.5

Avec x = 4ms, 1 + 0.75 + 0.3 = 2.05 > 1.5

(4+3)/x + 0.3 <= 1.5

(4+3)/x <= 1.2

7 <= 1.2x

x >= 7/1.2

7/1.2 ~= 5.8

Il faut donc au moins x = 6ms pour qu'on puisse ordonnancer les tâches.

Mais comme 2^(1/3) pourrait être plus proche de 1.3, on prend x = 7ms.

Commentaire :

Réponse incorrecte.

Réponse attendue :

Il faut que $\sum(c_i/d_i) \leq 1$ (c'est le cas le plus général).

Donc : $4/x+3/x+6/20=1$ et donc $x=10$.