Chapitre 7 : Analyse

7.1 Un programme comporte 6 fonctions, main, A, B, C, D et E. L'outil gprof calcule le nombre d'appels de chaque fonction, avec leur provenance, et prend à intervalle de temps régulier un échantillon du compteur de programme. Pour un programme, il fournit les informations suivantes sur chaque fonction (nombre d'appels avec leur appelant; nombre d'échantillons du compteur de programme) : main (1 appel, pas d'appelant; 0 échantillon), A (5 appels, main:3, B:2; 4 échantillons), B (5 appels, main:5; 2 échantillons), C (6 appels, A:6; 6 échantillons), D (8 appels, A:5, B:3, 16 échantillons), E (8 appels, B:8; 12 échantillons). Pour chaque fonction, donnez le pourcentage du temps passé dans la fonction elle-même (self) et le pourcentage du temps passé dans la fonction et celles appelées récursivement (self+childs). Sur plusieurs exécutions avec les mêmes entrées (le programme n'utilise pas de variable aléatoire), est-ce que les nombres d'appels et d'échantillons peuvent varier? Quelle est la fiabilité des temps que vous avez calculés (self et self+childs)?

7.2 Un programme comme Oprofile ou Perf est utilisé pour prendre des échantillons du compteur de programme à intervalle de temps régulier. Que peut-on calculer à partir de ces échantillons? Si les échantillons sont plutôt récoltés basés sur des interruptions générées par les compteurs de performance (e.g., à tous les 100000 décomptes), par exemple celui des fautes de cache de données L1, que peut-on calculer? Finalement, on décide de plutôt utiliser un outil comme CPU Profile qui échantillonne à intervalle régulier le contenu de la pile d'appel (la fonction courante et celles dans le chemin d'appel). Lorsqu'un échantillon se répète, ce qui arrive souvent, un facteur multiplicatif est ajouté plutôt que de sauver le même chemin d'appel deux fois dans le fichier de sortie. Voici les échantillons récoltés, calculez le temps passé dans chaque fonction, self et self+childs : (main, A) 3 fois, (main,A,C) 6 fois, (main,A,D) 6 fois, (main,B,A) 1 fois, (main,B) 2 fois, (main,B,D) 10 fois, (main,B,E) 12 fois.

7.3 La version usuelle de malloc/new a été remplacée par une version qui, avant d'appeler la fonction usuelle, prend un échantillon de la pile d'appel (chemin d'appel) et annote le nombre d'octets alloués afin de créer un outil semblable à Heap Profile. L'information récoltée est donc une liste de chemins d'appel avec pour chacun le nombre d'octets alloués. Peut-on faire exactement le même traitement que celui réalisé par CPU Profile? Dans ce cas, qu'est-ce que le total pour self et self+childs donnera? Si, afin de réduire le surcoût de la collecte d'information, on ne collecte un échantillon de pile d'appel que pour 1 invocation de malloc/new sur 10, que donnera l'information résultante? On aimerait en plus détecter les objets non libérés ou libérés 2 fois, comment pourrait-on effectuer

cette vérification efficacement si on intercepte en plus les appels aux fonctions free/delete?

7.4 Un outil comme Helgrind note tous les appels de synchronisation (e.g., prise et relâchement de verrou, barrière, opération atomique…) ainsi que les accès en mémoire de variables partagées (variables globales ou dans le monceau). Expliquez comment cet outil pourra distinguer les cas problématiques des cas corrects dans les situations suivantes : deux thread incrémentent une même variable i) sans protection, ii) avec une opération atomique, iii) avec un verrou; iv) plusieurs threads calculent chacun une partie différente d'un vecteur, attendent à un rendez-vous global et ensuite chaque thread accède l'ensemble du vecteur en lecture; v) les ajouts à une queue sont protégés par un verrou alors que les recherches ne sont pas protégées par des verrous en utilisant la technique RCU.

7.5 Quel serait un bon outil pour chacune des tâches suivantes : i) trouver une fuite de mémoire dans un programme, ii) détecter de la corruption de mémoire dans un programme, iii) optimiser le temps CPU moyen des fonctions d'un processus, iv) trouver un problème de fautes de cache, v) vérifier si un programme souffre de blocages dans le pipeline d'exécution (e.g., cycles perdus en raison d'une mauvaise prédiction des sauts conditionnels), vi) détecter un problème de course dans un logiciel multi-thread, vii) détecter un problème intermittent de latence indue dans une application répartie, viii) valider qu'une section de code est correcte quel que soit l'ordonnancement possible des instructions.